Trusted Platform Module Library

Part 1: Architecture

Family "2.0"

Level 00 Revision 01.59

November 8, 2019

Published

Contact: admin@trustedcomputinggroup.org

TCG Published

Copyright © TCG 2006-2020

Licenses and Notices

Copyright Licenses:

- Trusted Computing Group (TCG) grants to the user of the source code in this specification (the "Source Code") a worldwide, irrevocable, nonexclusive, royalty free, copyright license to reproduce, create derivative works, distribute, display and perform the Source Code and derivative works thereof, and to grant others the rights granted herein.
- The TCG grants to the user of the other parts of the specification (other than the Source Code) the rights to reproduce, distribute, display, and perform the specification solely for the purpose of developing products based on such documents.

Source Code Distribution Conditions:

- Redistributions of Source Code must retain the above copyright licenses, this list of conditions and the following disclaimers.
- Redistributions in binary form must reproduce the above copyright licenses, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.

Disclaimers:

- THE COPYRIGHT LICENSES SET FORTH ABOVE DO NOT REPRESENT ANY FORM OF LICENSE OR WAIVER, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, WITH RESPECT TO PATENT RIGHTS HELD BY TCG MEMBERS (OR OTHER THIRD PARTIES) THAT MAY BE NECESSARY TO IMPLEMENT THIS SPECIFICATION OR OTHERWISE. Contact TCG Administration (admin@trustedcomputinggroup.org) for information on specification licensing rights available through TCG membership agreements.
- THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO EXPRESS OR IMPLIED WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, ACCURACY, COMPLETENESS, OR NONINFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.
- Without limitation, TCG and its members and licensors disclaim all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification and to the implementation of this specification, and TCG disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this specification or any information herein.

Any marks and brands contained herein are the property of their respective owners.

Change History

Revision 98

Added parameter to MemoryMove(), MemoryCopy(), and MemoryConcat() to make sure that the data being moved will fix into the receiving buffer

Change the size of local 2B buffers so that they are sized to the sum of the sizes of the elements rather than any other mathematical construct. This forces the size of the local buffer to track any changes to the sizes of the input components rather than have some assumed relationship.

Made multiple changes to code to eliminate “dead” code (code that could not be reached by any perturbation of the inputs).

Removed the “+” from the handle parameter in TPM2_HMAC_Start().

Changed TPM_RC_BAD_TAG to 0x01e so that its value would match TPM_BADTAG from 1.2

Changed reference implementation so that it would only allow use of default exponent for creation of RSA keys. It will allow other exponents for imported keys.

Changed _cpri__GenerateKeyRSA() in CpriRSA.c so that it no longer reads outside the bounds of an array when getting a value to use for encrypting/decrypting with a key, generated from a seed.

Removed TPM_NV_INDEX entity name space.

Authorization check includes locality.

Revision 99

Added phEnableNV to make NV enable independent of the platform hierarchy enable.

Added TPM2_PolicyNvWritten to permit a policy based on whether or not NV has been written

Added TPM_PT_NV_BUFFER_MAX, the maximum data size in an NV write.

Added define for HCRTM PCR, platform specific

Return code when an NV hierarchy is disabled is TPM_RC_HANDLE.

TPM2_Shutdown state may be nullified on any subsequent command.

CTR mode increments the entire IV, not just 32 bits.

TPM2_PolicySecret cannot have a null authHandle.

Revision 101

Added Definitions for Endorsement Authorization, Owner Authorization, Platform Authorization.

An error may change TPM state under certain conditions.

A restricted signing key cannot have a scheme of TPM_ALG_NULL.

Added TPMS_EMPTY.

 Copyright © TCG 2006-2020

TPM2_Sign: The signing scheme hash algorithm determines the size of the hash to be signed. However, this may be removed in a future revision.

TPM2_PCR_Allocate may return an error if the allocation fails.

Revision 103

Added ISO/IEC references and forward.

Handle errors always return TPM_RC_HANDLE, not TPM_RC_HIERARCHY.

TPM_PCR_Allocate does not change allocation for a bank not listed.

For a policy ticket, if expiration is non-negative, a NULL ticket is returned.

Revision 105

Added *lockoutPolicy.*

Added vendor-specific handles.

Added detection of a clock discontinuity to tickets.

Reworked TPM2_Import description.

Revision 107

Some reworking of H-CRTM, D-RTM.

Some clarification of policy expiration.

Changed references to ISO/IEC standards.

Change PPS, EPS Clear flush resident transient and persistent objects.

Revision 109

Any field upgrade preserves state, not just the standard commands.

Added TPM 2.0 Part 1 description of vendor-specific authorization values.

Refined description of PCR interaction with H-CRTM, TPM2_Startup, and locality. _TPM2_Hash_Start indicates the start of an H-CRTM sequence, not DRTM.

A non-authorization session must have at least one of encrypt, decrypt, or audit set

A policy session timeout can only change to a shorter value.

Added defines for ECC curves and removed some redundant values in the Part B annex.

TPM2_Sign can use a symmetric key.

TPM2_NV_UndefineSpace fails if TPMA_NV_POLICY_DELETE is set.

 Copyright © TCG 2006-2020

Revision 111

TPM2_ContextSave encrypts just the TPM2B_CONTEXT_SENSITIVE structure.

TPM 2.0 Part 2 structures removed algorithms and added notation referring to algorithm registry.

HMAC commands cannot be used with a restricted key.

Revision 113

Clarified Auth Role for hierarchies and NV Index.

Added password check to authorization checks.

Indicated that handles returned by the TPM are TPM_HT_TRANSIENT (three places).

Revision 115

FIPS 186-4 note.

Return codes for tag requires vs. actual mismatch.

Revision 117

A trial session cannot use encrypt or decrypt

HMAC is optional when the HMAC key is the Empty Buffer. If present, it must be correct.

CFB uses *sessionValue* in the KDF, not *sessionKey*

FIPS-140 requires NV to be erased when an Index is deleted. NV data must be initialized on a first partial write.

TPM2_Create for a keyed hash object must have TPM_ALG_NULL if *sign* and *decrypt* are both SET or CLEAR.

For an unrestricted HMAC key, if both the key and parameter have a non-NULL scheme, they must match.

Revision 119

Defined transient object and made the use of object and sequence object more consistent.

Refined the description of an exclusive audit session, the definition of auditReset, and its relationship to the audit attribute.

Explained that the TPM clock must be accurate even if there is no reliable external clock.

Updated the informative algorithm ID table.

TPM2_HMAC and TPM2_HMAC_Start return code change.

All signing commands, including attestation commands, return TPM_RC_KEY for a non-signing key.

TPM2_SetCommandCodeAuditStatus is not audited when used to change the algorithm.

 Copyright © TCG 2006-2020

Trial policy sessions check authorizations.

DA protection does apply to TPM_RH_LOCKOUT.

Revision 121

continueAuthSession is ignored for a password session.

Reworked NV attributes to accommodate more NV types. Defined TPM_NT.

For a hybrid counter Index, the first write always writes through to NV memory.

Added ECC point padding description.

Unmarshaling routines return error code, not bool. Detailed CommandDispatcher parameters. Unmarshal flag set means null is permitted.

The algorithm ID table in this specification is informative.

Context gap must be 2^^n-1.

Handle type 0x03 is for saved sessions, not active session.

Timeout is of length TPM2B_DIGEST, not UINT64.

nullProof can be used in a ticket.

TPM2_EncryptDecrypt uses an unrestricted key. The sign attribute is used as an encrypt attribute. A non-null mode cannot be overridden.

A TPM2_PolicySecret being satisfied by a policy requires a password or auth value. The object must permit password or HMAC authorization.

TPM2_PolicyNV is an immediate assertion.

Revision 122

NULL password can have continue set or clear.

Sign attribute becomes encrypt attribute for a symmetric cipher object.

Saved context metadata is normative. Encrypted data is vendor specific.

TPMU_SYM_MODE, TPMS_SCHEME_XOR selector permits NULL.

If the session requires a policy session, returns TPM_RC_AUTH_TYPE.

TPM2_NV_Certify returns TPM_RC_NV_UNINITIALIZED if unwritten even if size is zero.

Revision 123

Advised that callers should not use NV read public to calculate the Name.

Removed advice that FIPS may require an *authValue* size of half the hash algorithm digest size.

Clarified that *nonceTPM* is only used once in an HMAC calculation when the session is being used for both encrypt and decrypt.

Clarified that *authValue* is an Empty Buffer if a session is not an authorization session.

Clarified that *sessionValue* for authorization sessions that are encrypt or decrypt sessions is *sessionKey* || *authValue* regardless of binding.

Clarified that nameAlg is the *authPolicy* hash algorithm.

Structure definition lower limits apply to TPM inputs. Upper limits refer to inputs and outputs.

The year and day of year can indicate an errata date.

TPM_RC_NONCE is returned for a nonce value mismatch.

TPMS ALGORITHM_DETAIL_ECC kdf can be TPM_ALG_NULL.

TPMS_CONTEXT *savedHandle* indicates the context type.

If a handle in handle area references a session and the session is not present, returns TPM_RC_REFERENCE_H0 + N.

Clarified that the size of an encrypted parameter can be zero.

TPM2_Startup can result in the PCR update counter non-zero because of PCR resets.

For RSA salt key, the size of an encrypted salt must be the same as the size of the public modulus.

TPM2_ECDH_KeyGen requires *restricted* CLEAR and *decrypt* SET.

TPM2_Commit does not require the *sign* attribute.

TPM_PolicyOR extends the digest into a Zero Digest PolicyDigest. It does not replace the digest.

TPM2_PolicyPCR with a trial policy may use the TPM PCR if the caller does provide PCR settings.

TPM2_PolicyNV, TPM2_PolicyCounterTimer, TPM2_NV_Certify, can return TPM_RC_VALUE if the offset is greater than the data size.

Indicated that the reference implementation can do compare operations on a structure using a cast to a byte array, so unmarshaling code must initialize input buffers.

Revision 124

This revision begins to implement the NV PIN Index type. The information is incomplete and subject to change. It is included as a work in progress rather than create two forks to the specification.

Clarified that TPM2B_DATA is the size of a TPMT_HA but is not required to contain an algorithm ID.

Clarified that time can be set to zero at _TPM_Init or TPM2_Startup.

TPM2_StartAuthSession rejects a symmetric salt key.

Revision 125

 Copyright © TCG 2006-2020

Continued specifying NV PIN Index. The information is complete but not reviewed and still subject to significant changes.

Session-based encryption should support XOR, but a block cipher is platform specific.

Added TPM_PT_MODES for FIPS and other indications. Added TPMA_MODES.

Clarified the TPMA_STARTUP_CLEAR attribute (enable flags) settings on the various startup types.

_PRIVATE structure - changed from TPMT_SENSITIVE to TPM2B_SENSITIVE.

Revision 126

Reworded the PIN Index and rewrap text.

Added restrictions on *unique* input for TPM2_Create and TPM2_CreatePrimary.Removed obsolete TPM_CC_PP_FIRST and TPM_CC_PP_LAST.

Revision 127

Removed symmetric salt.

Revision 128

sensitiveDataOrigin is set for an asymmetric object.

Clarified that only the template *unique* field may be altered when an object is created.

A PIN index can be used in TPM2_PolicySecret if read or write locked.

ehProof is changed on TPM2_Clear.

TPM2_SetPrimaryPolicy requires a policy length consistent with the hash algorithm.

Revision 130

Augmented section 27.1 “Object Creation / Introduction” by adding the table “Creation Commands” and a description of that table.

Augmented section 27.6.1 “Entropy Creation / Introduction” by adding the table “Deriving Cryptographic Values” and a description of that table.

Added TPM2_PolicyTemplate(), TPM2_CreateLoaded(), TPMI_DH_PARENT.

Revision 131

Added TPM2_PolicyAuthorizeNV(), TPM2_EncryptDecrypt2().

Noted that TPM2_Create() may require transient resources.

TPM2_Clear() increments the *pcrUpdateCounter*, permitting a policy that can be invalidated on TPM2_Clear().

TPM_PT_NV_BUFFER_MAX returns the maximum size for NV read and NV certify as well as NV write,

 Copyright © TCG 2006-2020

Noted that TPMA_NV_POLICY_DELETE with a policy that cannot be satisfied defines an Index that can never be deleted.

TPM2_NV_Read ignores *offset* for bits and counter indexes.

Revision 132

Reworked Part 4 for refactored crypto code merge.

Added application note on audit alternative.

Added command code for PolicyAuthorizeNV and EncryptDecrypt2.

Added getcapability TPM_CAP_AUTH_POLICIES for hierarchy policies, and new structure TPMS_TAGGED_POLICY.

Offset is ignored when reading counter and bits NV indexes.

ReadClock can have audit session.

Revision 133

Added additional option to ticket expiration, and *timeEpoch*.

TPM2B_PRIVATE always has authorization value padded.

Clarified GPIO inputs and outputs.

EC Schnorr computation changes.

Salt always uses OAEP.

KDF must reject weak keys.

Revision 134

TPM2_Create for a *fixedParent* storage key only requires the symmetric algorithm of the parent and child to match.

Policy ticket creation also digests the *timeEpoch*.

Revision 135

Weak symmetric keys will not be generated and cannot be loaded.

OAEP uses the object's scheme. If the object's scheme is TPM_ALG_NULL, uses the objects Name algorithm.

GPIO input and output settings are platform or vendor specific.

Added a TPM2_Create, etc. reference code error check if data objects have *sensitiveDataOrigin* SET. The normative text was correct.

Revision 135 June 20

Modified the ECDAA signature calculation

Revision 136

Added PolicyAuthorize definition.

Noted that weak symmetric keys are not permitted.

OAEP uses the key's scheme unless it is NULL

Modifications to the ECDAA sign operation.

Parents use CFB mode, and cannot have a NULL symmetric algorithm

The salt key scheme must be NULL or OAEP.

Revision 137

Updated the interaction between nonceTPM and expiration.

data may be a non - Empty Buffer when a primary key is created.

TPM2_PolicySecret() referencing a PIN Pass Index returns a NULL ticket.

TPM2_SelfTest returns TPM_RC_FAILURE on failure.

phEnableNV is set on TPM Reset or TPM Restart

TPM2_Create and TPM2_CreatePrimary input is actually TPM2_PUBLIC even though the parameter says TPM2_TEMPLATE.

TPM2_PolicySecret for PIN and non-PIN Index clarifications.

TPM2_PolicyNV, TPM2_NV_Read, TPM2_NV_Certify may ignore offset parameter.

TPM2_NV_GlobalWriteLock, TPM2_NV_ReadLock may write NV.

Part 4 added SelfTest.h, Simulator_fp.h, removed CryptEccData.c,

Part 4 updated TPM2B structure sample.

Revision 138

Added back expiration comment that timeout cannot become smaller.

Explained the result of TPM_CAP_AUTH_POLICIES.

Removed obsolete CommandDispatcher.h and HandleProcess.h.

Revision 139

Revision 140

Added Attached Component (AC Send) description, structures, and functions.

TPM2_ECC_Parameters() may zero pad results.

TPM2_DictionaryAttackParameters does not reset failedTries.

Revision 141

Clarified that the KDFa 0x00 byte is only explicitly added if Label is not present or if it is not NULL-terminated.

Clarified that recoveryTIme may be tracked through a shutdown.

Added TDES annex explaining parity generation.

Revision 142

Code merge with 141.

Revision 143

Clarified HMAC key calculation for bound policy session with and without TPM2_PolicyAuthValue. Similar clarification for encrypted policy session.

Added the TPM2_MAC commands and merged with TPM2_HMAC commands. Added TPMI_ALG_MAC_SCHEME.

Changed TPM2B_TIMEOUT back to a UINT64.

TPM2_FlushContext for sessions ignores the upper byte of the handle.

Revision 144

Minor updates for TPM2_MAC.

Added TPMI_ALG_CIPHER_MODE, used for EncryptDecrypt.

Salt key must be a decrypt key.

seedValue is the size of the *nameAlg* digest.

Revision 145

More informative explanation. No normative changes.

Revision 146

Typos and fonts. No normative changes

Revision 147

Field upgrade should preserve the TPM vendor provisioned EKs.

Salt can only use asymmetric key encryption.

Alternative implementation of *failedTries* on non-orderly shutdown.

Added description of entropy usage for derived objects.

Alternate implementations for NV counter index initialization.

TPM_PT_NV_COUNTERS_MAX - zero value indicates no specified maximum.

Added TPMI_DH_SAVED for handle values that can be used in TPM2_ContextSave or TPM2_FlushContext.

TPMS_SCHEME_XOR cannot have a NULL hash algorithm

TPM2_PolicyTemplate() error codes if command is sent twice or if *cpHash* is already set.

CryptSym.h added to Part 4.

Revision 148

Reworked the attestation key certification to indicate that an encrypted challenge response is a more likely use case than an encrypted certificate.

Field upgrade should not affect TPM2_CreatePrimary() outputs under certain conditions.

The reset of the TIme circuit is related to TPM power, not TPM_Init.

MAX_SYM_DATA 128 changed from shall to should.

sign and decrypt both CLEAR or SET and scheme not TPM_ALG_NULL returns TPM_RC_SCHEME.

TPM2_PCR_Allocate() takes effect at _TPM_Init(), not TPM2_Startup().

Clarified in the text (the code was correct) that TPM2_PolicyDuplicationSelect() Names do not include the size.

The TPM may enter Failure mode if TPM2_Startup() is not TPM_SU_CLEAR after an algorithm set change that affects PCR banks. It was previously not a may.

After a field upgrade, preserving seeds, etc. was changed from shall to should.

Revision 149

Part 1 added phEnableNV to STATE_CLEAR_DATA, clearCount increments on TPM Restart, not TPM Resume

Noted that TPM2_EventSequenceComplete() always returns all hashes.

Noted that TPM2_PCR_Allocate() requirement for TPM_SU_CLEAR only applies until after the next _TPM_Init.

Part 4: For code merge: Added KdfTestData.h. Deleted BnEccData.c. Changed CryptDataEcc.c to CryptEccData.c

Revision 150

Added some notes about the interaction between audit and parameter encryption. Clarified that the audit digest is a single hash of *cpHash* and *rpHash*.

The random commit value has to be at least equal to the security strength of the signing key. KDFa for the commit calculation uses *vendorAlg*, not *nameAlg*.

+ decoration only applies to command parameters, not response parameters.

TPM2_Startup() does not clear the written bit for an orderly counter Index.

Removed CryptHashData.h.

Revision 151

TPM2_Hash() and TPM2_SequenceComplete() creates a ticket, with an Empty digest if in the NULL hierarchy.

TPM2_VerifySignature() returns a ticket with an Empty digest if the key is in the NULL hierarchy.

Updated ECC key generation and point padding in the Part 1 annex.

The TPM_PT_PS_REVISION value is platform specific.

Moved implementation specific description of the *Clock Safe* flag to an example.

Clarified that the getcapability returning TPML_PCR_SELECTION must return a selection for allocated banks but can return additional selections.

Revision 152

Added a first draft of TPM2_CertifyX509().

Revision 153

C.5 ECC Key Generation changed d to c and G to Q.

TPM2B_PRIVATE_KEY_RSA is permitted to be larger for *fixedTPM* keys. the TPM2B_PRIVATE structure in TPM2_Create() and TPM2_Load() may contain five CRT primes (instead of one).

Assign TPM_CC_CertifyX509 command code, x509Sign attribute, TPMA_X509_KEY_USAGE, Add TPM2_CertifyX509 description, parameters, and actions.

Define NV digest attestation structure, TPMS_NV_DIGEST_CERTIFY_INFO, and added certifying an NV digest to TPM2_NV_Certify.

Revision 154

Clarify that *label* in KDFa is an octet stream and the conditions for the KDFa zero byte.

Clarify the required size of an object sensitive area *seedValue* for TPM generated and imported objects.

Clarify that the L parameter in OAEP is a byte stream with the last byte zero, not a null terminated string.

Add an Annex with a Library Profile Guide.

Clarify that TPM_PT_NV_BUFFER_MAX applies to NV extend or NV certify.

The TPMA_X509_KEY_USAGE keyAgreeement and encipherOnly attributes require the decrypt attribute.

Explain that most of TPM2_SetAlgorithmSet is vendor-dependent.

Explain that the initialization of the list of commands requiring physical presence is platform-specific.

Revision 155

 Copyright © TCG 2006-2020

Part 2 removed , S AND <IO> from several table titles.

TPM_ECC_CURVE add + to TPM_ECC_NONE

Part 4 added files for X.509 support: OIDS.h, MinMax.h, and AC support: AC_spt.c

Changed Implementation.h to TpmProfile.h and added a pointer in TpmBuildSwitches.h to preset the values.

Revision 156

Added the ACT feature.

Explained that the TPM2_CertifyX509 *partialCertificate* and *addedToCertificate* are a DER encoded SEQUENCEs. Explained the encoding of the TPMA_OBJECT element. Noted that *tbsDigest* is returned as a debugging aid. Changed *qualifyingData* to *reserved* and that it must be an Empty Buffer.

Added a requirement that, if a command resets PCR in multiple banks, the PCR Update Counter must be incremented only once. If a command causes PCR in multiple banks to change, the PCR Update Counter must be incremented once for each bank.

Revision 157

Added the ACT code.

Revision 158

Minor updates to the ACT description.

Revision 159

Added several missing source code files to Part 4.

Reversed the TPMA_X509_KEY_USAGE bit map.

Acknowledgements

The writing of a specification, particularly a security specification, takes many hours for both development and review. This specification is no exception with roughly 100 individuals involved in the process. The TCG would like to acknowledge the contribution of those individuals (listed below) and the companies who allowed them to volunteer their time to the development of this specification.

The TCG would like to acknowledge the special contribution of David Wooten in the development of the TPM 2.0 architecture and documentation of this specification. We also acknowledge the generosity of Microsoft in contributing the code in this specification, written by David Wooten, Jiajing Zhu, and Paul England.

Special thanks are due to David Challener, David Wooten, Julian Hammersley, Graeme Proudler, and Ari Singer who served as Chair of the TPM Working Group at different times during the development of this specification.

The TCG would also like to give special thanks to David Grawrock, David Wooten, and Ken Goldman, who were the editors of this specification.

Contributors:

Loic Duflot; ANSSI
Frederic Guihery; AMOSSYS
Ralf Findeisen; AMD
Julian Hammersley; AMD
Dean Liberty; AMD
Ron Perez; AMD
Emily Ratliff; AMD
Gary Simpson; AMD
Gongyuan Zhuang; AMD
John Mersh; ARM Ltd.
Kerry Maletsky; Atmel
Randy Mummert; Atmel
Ronnie Thomas; Atmel
Douglas Allen; Broadcom
Chares Qi; Broadcom
Daniel Nowack; BSI
Florian Samson; BSI
Bill Lattin; Certicom
Matt Harvey; CESG
Paul Waller; CESG
Bob Bell; Cisco
Bill Jacobs; Cisco
Rafael Montalvo; Cisco
Frank Mosberry; Dell
Amy Nelson; Dell
Ari Singer; DMI
Sigrid Gürgens; Fraunhofer SIT
Andreas Fuchs: Fraunhofer SIT
Carsten Rudolph; Fraunhofer SIT
Carline Covey; Freescale Semiconductor
Ira McDonald; High North
Vali Ali; Hewlett Packard
Liqun Chen; Hewlett Packard
Carey Huscroft; Hewlett Packard
Wael Ibrahim; Hewlett Packard
Graeme Proudler; Hewlett Packard
Ken Goldman; IBM
Hans Brandl; Infineon
Hubert Braunwarth; Infineon
Ga-Wai Chin; Infineon
Roland Ebrecht; Infineon
Markus Gueller; Infineon
Ralph Hamm; Infineon
Georg Rankl; Infineon
Will Arthur; Intel
Ernie Brickell; Intel
Alex Eydelberg; Intel
David Grawrock; Intel
Jiangtao Li; Intel
David Riss; Intel
Ned Smith; Intel
Claire Vishik; Intel
Monty Wiseman; Intel
Igor Slutsker; Intel
Liran Perez; Intel
Zecharye Galitzky; Intel
Joshua Su; ITE
David Challener; Johns Hopkins APL
Huang Qian; Lenovo
Ronald Aigner; Microsoft
Jing De Jong-Chen; Microsoft
Shon Eizenhoefer; Microsoft
Carl Ellison; Microsoft
Paul England; Microsoft
Leonard Janke; Microsoft
Richard Korry; Microsoft
Jork Loeser; Microsoft
Andrey Marochko; Microsoft
Jim Morgan; Microsoft
Dennis Mattoon; Microsoft
Himanshu Raj; Microsoft
David Robinson; Microsoft
Rob Spiger; Microsoft
Stefan Thom; Microsoft
Mark Williams; Microsoft
David Wooten; Microsoft
Jiajing Zhu; Microsoft
Luis Samenta; MIT
Ariel Segall; MITRE
Nataly Kremer; M-Systems Flash
Andrew Regenscheid; NIST
Qin Fan; Nationz
Jay Liang; Nationz
Xin Liu; Nationz
Jan-Erik Ekberg; Nokia
Michael Cox; NTRU
Nick Howgrave-Graham; NTRU
William Whyte; NTRU
Leooid Asriel; Nuvoton
Dan Morav; Nuvoton
Erez Naory; Nuvoton
Oren Tanami; Nuvoton
Dennis Huage; NVIDIA
Whllys Ingersoll; Oracle
Scott Rotondo; Oracle
Timothy Markey; Phoenix
Anders Rundgren; PrimeKey Solutions
Laszlo Elteto; Safenet
Michael Willet; Seagate
Olivier Collart; STMicroelectronics
Miroslav Dusek; STMicroelectronics
Jan Smrcek; STMicroelectronics
Mohamed Tabet; STMicroelectronics
Paul Sangster; Symantec
Jerome Quevremont; Thales
Mark Ryan; University of Birmingham
Mike Boyle; US Department of Defense
Stanley Potter; US Department of Defense
Sandi Roddy; US Department of Defense
Adrian Stanger; US Department of Defense
Kelvin Li; VIA
Nick Bone; Vodafone
Mihran Dars; Wave Systems
Thomas Hardjono; Wave Systems
Greg Kazmierczak; Wave Systems
Len Veil; Wave Systems

 Copyright © TCG 2006-2020

CONTENTS

1 Scope ... 1
2 Specification Organization ... 2
3 Normative references ... 3
4 Terms and definitions ... 4
5 Symbols and Abbreviated Terms ... 14
5.1 Symbols ... 14
5.2 Abbreviations ... 14
6 Compliance ... 17
7 Conventions ... 18
7.1 Bit and Octet Numbering and Order ... 18
7.2 Sized Buffer References ... 18
7.3 Numbers ... 18
8 Changes from Previous Versions ... 20
9 Trusted Platforms ... 21
9.1 Trust ... 21
9.2 Trust Concepts ... 21
9.2.1 Trusted Building Block ... 21
9.2.2 Trusted Computing Base ... 21
9.2.3 Trust Boundaries ... 21
9.2.4 Transitive Trust ... 22
9.2.5 Trust Authority ... 22
9.3 Trusted Platform Module ... 23
9.4 Roots of Trust ... 23
9.4.1 Root of Trust for Measurement (RTM) ... 24
9.4.2 Root of Trust for Storage (RTS) ... 24
9.4.3 Root of Trust for Reporting (RTR) ... 24
9.5 Basic Trusted Platform Features ... 25
9.5.1 Introduction ... 25
9.5.2 Certification ... 25
9.5.3 Attestation and Authentication ... 26
9.5.4 Protected Location ... 29
9.5.5 Integrity Measurement and Reporting ... 29
10 TPM Protections ... 31
10.1 Introduction ... 31
10.2 Protection of Protected Capabilities ... 31
10.3 Protection of Shielded Locations ... 31
10.4 Exceptions and Clarifications ... 31
11 TPM Architecture ... 33
11.1 Introduction ... 33
11.2 TPM Command Processing Overview ... 33
11.3 I/O Buffer ... 37
11.4 Cryptography Subsystem ... 37

 Copyright © TCG 2006-2020

11.4.1 Introduction 37
11.4.2 Symmetric Block Cipher MAC Algorithms 37
11.4.3 Hash Functions 37
11.4.4 HMAC Algorithm 38
11.4.5 Asymmetric Operations 38
11.4.6 Signature Operations 39
11.4.7 Symmetric Encryption 41
11.4.8 Extend 42
11.4.9 Key Generation 43
11.4.10 Key Derivation Function 43
11.4.11 Random Number Generator (RNG) Module 47
11.4.12 Algorithms 49

11.5 Authorization Subsystem 50
11.6 Random Access Memory 50

11.6.1 Introduction 50
11.6.2 Platform Configuration Registers (PCR) 50
11.6.3 Object Store 51
11.6.4 Session Store 52
11.6.5 Size Requirements 52

11.7 Non-Volatile (NV) Memory 52
11.8 Power Detection Module 53

12 TPM Operational States 54

12.1 Introduction 54
12.2 Basic TPM Operational States 54

12.2.1 Power-off State 54
12.2.2 Initialization State 54
12.2.3 Startup State 55
12.2.4 Shutdown State 57
12.2.5 Startup Alternatives 58

12.3 Self-Test Modes 59
12.4 Failure Mode 60
12.5 Field Upgrade 61

12.5.1 Introduction 61
12.5.2 Field Upgrade Mode 61
12.5.3 Preserved TPM State 64
12.5.4 Field Upgrade Implementation Options 65

13 TPM Control Domains 66

13.1 Introduction 66
13.2 Controls 66
13.3 Platform Controls 67
13.4 Owner Controls 68
13.5 Privacy Administrator Controls 68
13.6 Primary Seed Authorizations 69
13.7 Lockout Control 69
13.8 TPM Ownership 70

13.8.1 Taking Ownership 70

 Copyright © TCG 2006-2020

13.8.2 Releasing Ownership 70

14 Primary Seeds 72

14.1 Introduction 72
14.2 Rationale 72
14.3 Primary Seed Properties 73

14.3.1 Introduction 73
14.3.2 Endorsement Primary Seed (EPS) 74
14.3.3 Platform Primary Seed (PPS) 74
14.3.4 Storage Primary Seed (SPS) 75
14.3.5 The Null Seed 75

14.4 Hierarchy Proofs 75

15 TPM Handles 77

15.1 Introduction 77
15.2 PCR Handles (MSO=00_{16}) 77
15.3 NV Index Handles (MSO=01_{16}) 77
15.4 Session Handles (MSO=02_{16} and 03_{16}) 77
15.5 Permanent Resource Handles (MSO=40_{16}) 78
15.6 Transient Object Handles (MSO=80_{16}) 78
15.7 Persistent Object Handles (MSO=81_{16}) 79

16 Names 80
17 PCR Operations 81

17.1 Initializing PCR 81
17.2 Extend of a PCR 81
17.3 Using Extend with PCR Banks 81
17.4 Recording Events 82
17.5 Selecting Multiple PCR 82
17.6 Reporting on PCR 83

17.6.1 Reading PCR 83
17.6.2 Attesting to PCR 83

17.7 PCR Authorizations 84

17.7.1 PCR Not in a Set 84
17.7.2 Authorization Set 84
17.7.3 Policy Set 85
17.7.4 Order of Checking 85

17.8 PCR Allocation 85
17.9 PCR Change Tracking 86
17.10 Other Uses for PCR 86

18 TPM Command/Response Structure 87

18.1 Introduction 87
18.2 Command/Response Header Fields 88

18.2.1 tag 89
18.2.2 commandSize/responseSize 89
18.2.3 commandCode 89
18.2.4 responseCode 89

18.3 Handles 89
18.4 Parameters 90
18.5 *authorizationSize*/*parameterSize* 90
18.6 Authorization Area 91
18.6.1 Introduction 91
18.6.2 Authorization Structure 92
18.6.3 Session Handles 93
18.6.4 Session Attributes (*sessionAttributes*) 93
18.7 Command Parameter Hash (*cpHash*) 95
18.8 Response Parameter Hash (*rpHash*) 96
18.9 Command Example 96
18.10 Response Example 98
19 Authorizations and Acknowledgments 99
19.1 Introduction 99
19.2 Authorization Roles 99
19.3 Physical Presence Authorization 100
19.4 Password Authorizations 101
19.5 Sessions 102
19.6 Session-Based Authorizations 102
19.6.1 Introduction 102
19.6.2 Authorization Session Formats 103
19.6.3 Session Nonces 103
19.6.4 Authorization Values 105
19.6.5 HMAC Computation 105
19.6.6 Note on Use of Nonces in HMAC Computations 107
19.6.7 Starting an Authorization Session 107
19.6.8 *sessionKey* Creation 108
19.6.9 Unbound and Unsalted Session Key Generation 108
19.6.10 Bound Session Key Generation 109
19.6.11 Salted Session Key Generation 112
19.6.12 Salted and Bound Session Key Generation 113
19.6.13 Encryption of *salt* 114
19.6.14 Caution on use of Unsalted Authorization Sessions 114
19.6.15 No HMAC Authorization 115
19.6.16 Authorization Selection Logic for Objects 115
19.6.17 Authorization Session Termination 116
19.7 Enhanced Authorization 116
19.7.1 Introduction 116
19.7.2 Policy Assertion 117
19.7.3 Policy AND 117
19.7.4 Policy OR 119
19.7.5 Order of Evaluation 121
19.7.6 Policy Session Creation 121
19.7.7 Policy Assertions (Policy Commands) 122
19.7.8 Policy Session Context Values 125
19.7.9 Policy Example 127
19.7.10 Trial Policy 127
19.7.11 Modification of Policies 127

 Copyright © TCG 2006-2020

19.7.12 TPM2_PolicySigned(), TPM2_PolicySecret(), and TPM2_PolicyTicket() ... 129
19.7.13 Use of TPM for authPolicy Computation ... 131
19.7.14 Trial Policy Session ... 131
19.7.15 Use of TPM2_PolicySigned() and TPM2_PolicySecret() without *nonceTPM* ... 132
19.8 Dictionary Attack Protection ... 132
19.8.1 Introduction ... 132
19.8.2 Lockout Mode Configuration Parameters ... 133
19.8.3 Lockout Mode ... 134
19.8.4 Recovering from Lockout Mode ... 134
19.8.5 Authorization Failures Involving *lockoutAuth* ... 134
19.8.6 Non-orderly Shutdown ... 135
19.8.7 Justification for Lockout Due to Session Binding ... 135
19.8.8 Sample Configurations for Lockout Parameters ... 136
20 Audit Session ... 137
20.1 Introduction ... 137
20.2 Exclusive Audit Sessions ... 138
20.3 Command Gating Based on Exclusivity ... 138
20.4 Audit Session Reporting ... 138
20.5 Audit Establishment Failures ... 139
20.6 Audit Alternative ... 139
21 Session-based encryption ... 140
21.1 Introduction ... 140
21.2 XOR Parameter Obfuscation ... 141
21.3 CFB Mode Parameter Encryption ... 141
22 Protected Storage ... 143
22.1 Introduction ... 143
22.2 Object Protections ... 143
22.3 Protection Values ... 143
22.4 Symmetric Encryption ... 144
22.5 Integrity ... 144
23 Protected Storage Hierarchy ... 147
23.1 Introduction ... 147
23.2 Hierarchical Relationship between Objects ... 147
23.3 Duplication ... 148
23.3.1 Definition ... 148
23.3.2 Protections ... 149
23.3.3 Rewrap ... 154
23.4 Duplication Group ... 157
23.5 Protection Group ... 158
23.6 Summary of Hierarchy Attributes ... 159
23.7 Primary Seed Hierarchies ... 160
24 Credential Protection ... 161
24.1 Introduction ... 161
24.2 Protocol ... 161

 Copyright © TCG 2006-2020

24.3 Protection of Credential 161
24.4 Symmetric Encrypt 162
24.5 HMAC 162
24.6 Summary of Protection Process 163

25 Object Attributes 164

25.1 Base Attributes 164

25.1.1 Introduction 164
25.1.2 *Restricted* Attribute 164
25.1.3 *Sign* Attribute 164
25.1.4 *Decrypt* Attribute 164
25.1.5 Uses 166

25.2 Other Attributes 167

25.2.1 fixedTPM and fixedParent 167
25.2.2 stClear 167
25.2.3 sensitiveDataOrigin 168
25.2.4 userWithAuth 168
25.2.5 adminWithPolicy 168
25.2.6 noDA 169
25.2.7 encryptedDuplication 169

26 Object Structure Elements 170

26.1 Introduction 170
26.2 Public Area 170
26.3 Sensitive Area 170
26.4 Private Area 171
26.5 Qualified Name 172
26.6 Sensitive Area Encryption 172
26.7 Sensitive Area Integrity 172

27 Object Creation 174

27.1 Introduction 174
27.2 Public Area Template 175

27.2.1 Introduction 175
27.2.2 type 175
27.2.3 nameAlg 175
27.2.4 objectAttributes 175
27.2.5 authPolicy 176
27.2.6 parameters 176
27.2.7 unique 176

27.3 Sensitive Values 176

27.3.1 Overview 176
27.3.2 userAuth 176
27.3.3 data 176

27.4 Creation PCR 177
27.5 Public Area Creation 177

27.5.1 Introduction 177

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

27.5.2 type, nameAlg, objectAttributes, authPolicy, and parameters ... 177
27.5.3 unique ... 177
27.6 Creation Entropy ... 178
27.6.1 Introduction ... 178
27.6.2 Entropy for Ordinary Objects ... 179
27.6.3 Entropy for Primary Objects ... 179
27.7 Sensitive Area Creation ... 179
27.7.1 Introduction ... 179
27.7.2 type ... 180
27.7.3 authValue ... 180
27.7.4 *seedValue* ... 180
27.7.5 sensitive ... 181
27.8 Creation Data and Ticket ... 182
27.9 Creation Resources ... 182
28 Object Derivation ... 183
28.1 Introduction ... 183
28.2 Derivation Parameters ... 183
28.3 Public Area Template ... 183
28.4 Entropy for Derived Objects ... 184
28.4.1 Conceptual Description ... 184
28.4.2 Implementation Alternatives ... 185
28.5 Derivation Process ... 185
29 Object Loading ... 186
29.1 Introduction ... 186
29.2 Load of an Ordinary Object ... 186
29.3 Public-only Load ... 186
29.4 External Object Load ... 187
30 Context Management ... 188
30.1 Introduction ... 188
30.2 Context Data ... 189
30.2.1 Introduction ... 189
30.2.2 Sequence Number ... 189
30.2.3 Handle ... 190
30.2.4 Hierarchy ... 191
30.3 Context Protections ... 191
30.3.1 Context Confidentiality Protection ... 191
30.3.2 Context Integrity Protection ... 192
30.4 Object Context Management ... 193
30.5 Session Context Management ... 193
30.6 Eviction ... 194
30.7 Incidental Use of Object Slots ... 195
31 Attestation ... 196

 Copyright © TCG 2006-2020

31.1 Introduction 196
31.2 Standard Attestation Structure 196
31.3 Privacy 197
31.4 Qualifying Data 197
31.5 Anonymous Signing 197
31.6 X.509 Certificate Signing 197

32 Cryptographic Support Functions 200

32.1 Introduction 200
32.2 Hash 200
32.3 HMAC 200
32.4 Hash, MAC, and Event Sequences 200

32.4.1 Introduction 200
32.4.2 Hash Sequence 201
32.4.3 Event Sequence 201
32.4.4 HMAC Sequence 201
32.4.5 Sequence Contexts 202

32.5 Symmetric Encryption 202
32.6 Asymmetric Encryption and Signature Operations 202

33 Locality 203
34 Hardware Core Root of Trust Measurement (H-CRTM) Event Sequence 204

34.1 Introduction 204
34.2 Dynamic Root of Trust Measurement 204
34.3 H-CRTM before TPM2_Startup() and TPM2_Startup() without H-CRTM 205

35 Command Audit 206
36 Timing Components 208

36.1 Introduction 208
36.2 Time 209
36.3 Clock 209

36.3.1 Introduction 209
36.3.2 *Clock* Implementation 210
36.3.3 Orderly Shutdown of *Clock* 210
36.3.4 *Clock* Initialization at TPM2_Startup() 211
36.3.5 Setting *Clock* 211
36.3.6 *Clock* Periodicity 212

36.4 resetCount 212
36.5 restartCount 213
36.6 Note on the Accuracy and Reliability of *Clock* 213
36.7 Privacy Aspects of Clock 214

37 NV Memory 216

37.1 Introduction 216
37.2 NV Indices 216

37.2.1 Definition 216
37.2.2 NV Index Allocation 217
37.2.3 NV Index Deletion 218
37.2.4 High-Endurance (Hybrid) Indices 218

 Copyright © TCG 2006-2020

37.2.5 Reading an NV Index 220
37.2.6 Updating an Index 220
37.2.7 NV Index in a Policy 224
37.2.8 PIN Index Considerations 225
37.3 Owner and Platform Evict Objects 226
37.4 State Saved by TPM2_Shutdown() 227
37.4.1 Background 227
37.4.2 NV Orderly Data 227
37.4.3 NV Clear Data 227
37.4.4 NV Reset Data 228
37.5 Persistent NV Data 229
37.6 NV Rate Limiting 231
37.7 NV Other Considerations 232
37.7.1 Power Interruption 232
37.7.2 External NV 232
37.7.3 PCR in NV 233
38 Multi-Tasking 234
39 Errors and Response Codes 235
39.1 Error Reporting 235
39.2 TPM State After an Error 235
39.3 Resource Exhaustion Warnings 235
39.3.1 Introduction 235
39.3.2 Transient Resources 235
39.3.3 Temporary Resources 236
39.4 Response Code Details 236
40 General Purpose I/O 238
41 Minimums 239
41.1 Introduction 239
41.2 Authorization Sessions 239
41.3 Transient Objects 239
41.4 NV Counters and Bit Fields 239
42 Attached Components 240
42.1 Introduction 240
42.1.1 Purpose 240
42.1.2 Concept 240
42.2 TPM2_AC_Send() 240
42.3 Send Object Types 241
42.4 Send Object Attributes 241
42.5 Attached Component Authorization 241
42.6 Attached Component Object Management 242
42.6.1 Discovery 242
42.6.2 Setup 242
42.6.3 Sending 242
42.7 Power States 243

 Copyright © TCG 2006-2020

42.8 Attached Component Format 243

43 Authenticated Countdown Timer (ACT) 244

43.1 Introduction 244
43.2 Description 244
43.3 Typical Use 244
43.4 Failure Mode 245
43.5 Field Upgrade 245
43.6 Typical ACT authPolicy 246

Annex A (informative) Policy Examples 247

A.1 Introduction 247
A.2 TPM 1.2 Compatible Authorization 247

Annex B (normative/informative) RSA 249

B.1 Introduction 249
B.2 RSAEP 250
B.3 RSADP 250
B.4 RSAES_OAEP 250
B.5 RSAES_PKCSV1_5 250
B.6 RSASSA_PKCS1v1_5 250
B.7 RSASSA_PSS 251
B.8 RSA Key Generation 252

B.8.1 Background 252
B.8.2 Large Prime Generation 252
B.8.3 RSA Key Generation Algorithm 253

B.9 RSA Cryptographic Primitives 253

B.9.1 Introduction 253
B.9.2 TPM2_RSA_Encrypt() 253
B.9.3 TPM2_RSA_Decrypt() 254

B.10 Secret Sharing 254

B.10.1 Overview 254
B.10.2 RSA Encryption of Salt 254
B.10.3 RSA Secret Sharing for Duplication 254
B.10.4 RSA Secret Sharing for Credentials 255

Annex C (normative/informative) ECC 256

C.1 Introduction 256
C.2 Split Operations 256

C.2.1 Introduction 256
C.2.2 Commit Random Value 256
C.2.3 TPM2_Commit() 257
C.2.4 TPM2_EC_Ephemeral() 258
C.2.5 Recovering the Private Ephemeral Key 259

C.3 ECC-Based Secret Sharing 259
C.4 EC Signing 259

C.4.1 ECDSA 259
C.4.2 ECDAA 259

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

C.4.3 EC Schnorr ... 261
C.5 ECC Key Generation ... 263
C.6 Secret Sharing ... 263
C.6.1 ECDH ... 263
C.6.2 ECDH Encryption of Salt ... 264
C.6.3 ECC Secret Sharing for Duplication ... 264
C.6.4 ECC Secret Sharing for Credentials ... 264
C.7 ECC Primitive Operations ... 264
C.7.1 Introduction ... 264
C.7.2 TPM2_ECDH_KeyGen() ... 264
C.7.3 TPM2_ECDH_ZGen() ... 264
C.7.4 Two-phase Key Exchange ... 265
C.8 ECC Point Padding ... 266
Annex D (normative/informative) Support for SMx Family of Algorithms ... 268
D.1 Introduction ... 268
D.2 SM2 ... 268
D.2.1 Introduction ... 268
D.2.2 SM2 Digital Signature Algorithm ... 269
D.2.3 SM2 Key Exchange ... 271
D.3 SM3 ... 272
D.4 SM4 ... 272
Annex E (normative/informative) TDES ... 273
E.1 TDES Key Parity Generation ... 273
Annex F (informative) Library Profile Guide ... 274
F.1 Introduction ... 274
F.2 Platform Specific Constants ... 274
F.3 PCR ... 274
F.4 Algorithms ... 274
F.5 Commands ... 274
F.6 Buffers ... 275
F.7 NV Storage ... 275
F.8 Sessions and Objects ... 275
F.9 Physical Presence ... 275
F.10 Dictionary Attack Lockout ... 275
F.11 Self Test ... 275
F.12 ACT ... 275

Tables

Table 1 — Block Cipher Parameters ... 41
Table 2 — Hierarchy Control Setting Combinations ... 67
Table 3 — Equations for Computing Entity Names ... 80
Table 4 — Separators ... 88
Table 5 — *Tag* Values ... 89
Table 6 — Use of Authorization/Session Blocks ... 92
Table 7 — Description of *sessionAttributes* ... 93
Table 8 — Command Layout for Example Command ... 97
Table 9 — Example Command Showing *authorizationSize* ... 97
Table 10 — Response Layout for Example Command ... 98
Table 11 — Example Response Showing *parameterSize* ... 98
Table 12 — Password Authorization of Command ... 101
Table 13 — Password Acknowledgment in Response ... 101
Table 14 — Session-Based Authorization of Command ... 103
Table 15 — Session-Based Acknowledgment in Response ... 103
Table 16 — Schematic of TPM2_StartAuthSession Command ... 107
Table 17 — Handle Parameters for TPM2_StartAuthSession ... 108
Table 18 — Format to Start Unbounded, Unsalted Session ... 109
Table 19 — Format to Start Bound Session ... 111
Table 20 — Format to Start Salted Session ... 112
Table 21 — Format to Start Salted and Bound Session ... 113
Table 22 — Mapping of Hierarchy Attributes ... 159
Table 23 — Allowed Hierarchy Settings ... 160
Table 24 — Mapping of Functional Attributes ... 166
Table 25 — TPM 1.2 Correspondence ... 167
Table 26 — Public Area Parameters ... 170
Table 27 — Sensitive Area Parameters ... 171
Table 28 — Creation Commands ... 174
Table 29 — Deriving Object Entropy ... 179
Table 30 — Standard Attestation Structure ... 196
Table 31 — Contents of the ORDERLY_DATA Structure ... 227
Table 32 — Contents of the STATE_CLEAR_DATA Structure ... 228
Table 33 — Contents of the STATE_RESET_DATA Structure ... 228
Table 34 — Contents of the PERSISTENT_DATA Structure ... 229

 Copyright © TCG 2006-2020

Figures

Figure 1 — Attestation Hierarchy ... 27
Figure 2 — Architectural Overview ... 33
Figure 3 — Command Execution Flow ... 37
Figure 4 — Random Number Generation ... 48
Figure 5 — TPM Startup Sequences ... 57
Figure 6 — On-Demand Self-Test ... 59
Figure 7 — Failure Mode Behavior ... 61
Figure 8 — Resuming Field Upgrade Mode after _TPM_Init ... 63
Figure 9 — Field Upgrade Mode ... 64
Figure 10 — Command Structure ... 88
Figure 11 — Response Structure ... 88
Figure 12 — Command/Response Header Structure ... 88
Figure 13 — Authorization Layout for Command ... 92
Figure 14 — Authorization Layout for Response ... 93
Figure 15 — A 12-input OR Policy ... 121
Figure 16 — Use of TPM2_PolicyAuthorize() to Avoid PCR Brittleness ... 128
Figure 17 — Creating a Private Structure ... 146
Figure 18 — Symmetric Protection of Hierarchy ... 148
Figure 19 — Duplication Process with Inner and Outer Wrapper ... 152
Figure 20 — Duplication Process with Outer Wrapper and No Inner Wrapper ... 153
Figure 21 — Duplication Process with Inner Wrapper and TPM_RH_NULL as NP ... 154
Figure 22 — Duplication Process with no Inner Wrapper and TPM_RH_NULL as NP ... 154
Figure 23 — Key Recovery Process ... 155
Figure 24 — Duplication Groups ... 158
Figure 25 — Protection Groups ... 159
Figure 26 — Creating a Identity Structure ... 163
Figure 27 — Response Code Evaluation ... 237

 Copyright © TCG 2006-2020

Trusted Platform Module Library

Part 1: Architecture

1 Scope

This specification defines the Trusted Platform Module (TPM) a device that enables trust in computing platforms in general. It is broken into parts to make the role of each part clear. All parts are required in order to constitute a complete standard

For a complete definition of all requirements necessary to build a TPM, the designer will need to use the appropriate platform-specific specification to understand all of the requirements for a TPM in a specific application or make appropriate choices as an implementer.

Those wishing to create a TPM need to be aware that this specification does not provide a complete picture of the options and commands necessary to implement a TPM. To implement a TPM the designer needs to refer to the relevant platform-specific specification to understand the options and settings required for a TPM in a specific type of platform or make appropriate choices as an implementer.

EXAMPLE The number of platform configuration registers and their attributes are not defined in this specification. Those values would be specified by a platform specific specification or alternatively determined by an implementer.

2 Specification Organization

This specification contains four parts, as follows. In normative clauses, text labeled NOTE or EXAMPLE are informative, non-normative. Text in Part 2 table columns Description or Comments are informative, non-normative.

TPM 2.0 Part 1: Architecture

TPM 2.0 Part 1 contains a narrative description of the properties, functions, and methods of a TPM. Unless otherwise noted, this narrative description is *informative*. TPM 2.0 Part 1 contains descriptions of some of the data manipulation routines that are used by this specification. The normative behavior for these routines is in C code in TPM 2.0 Part 3 and TPM 2.0 Part 4. Algorithms and processes described in this TPM 2.0 Part 1 may be made normative by reference from TPM 2.0 Part 2, TPM 2.0 Part 3, or TPM 2.0 Part 4.

TPM 2.0 Part 2: Structures

TPM 2.0 Part 2 contains a *normative* description of the constants, data types, structures, and unions for the TPM interface. Unless otherwise noted: (1) all tables and C code in TPM 2.0 Part 2 are normative, and (2) normative content in TPM 2.0 Part 2 takes precedence over any other part of this specification.

TPM 2.0 Part 3: Commands

TPM 2.0 Part 3 contains: (1) a *normative* description of commands, (2) tables describing the command and response formats, and (3) C code that illustrates the actions performed by a TPM. Within TPM 2.0 Part 3, command and response tables have the highest precedence, followed by the C code, followed by the narrative description of the command. TPM 2.0 Part 3 is subordinate to TPM 2.0 Part 2.

A TPM need not be implemented using the C code in TPM 2.0 Part 3. However, any implementation should provide equivalent or, in most cases, identical results as observed at the TPM interface or demonstrated through evaluation.

TPM 2.0 Part 4: Supporting Routines

TPM 2.0 Part 4 presents C code that describes the algorithms and methods used by the command code in TPM 2.0 Part 3. The code in TPM 2.0 Part 4 augments Parts 2 and 3 to provide a complete description of a TPM, including the supporting framework for the code that performs the command actions.

Any TPM 2.0 Part 4 code may be replaced by code that provides similar results when interfacing to the action code in TPM 2.0 Part 3. The behavior of TPM 2.0 Part 4 code not included in an annex is *normative*, as observed at the interfaces with TPM 2.0 Part 3 code. Code in an annex is provided for completeness, that is, to allow a full implementation of the specification.

NOTE This specification does not provide code for lower-level cryptographic algorithms and use of external libraries is required for a complete implementation.

Extensive modification of the code provided in TPM 2.0 Part 4 annexes is expected for any TPM implementation. Modifications are required in order to interface the TPM code with actual TPM hardware rather than the simulation framework provided. In addition, modifications of the code in TPM 2.0 Part 4 annexes would be necessary in order to meet the needs of applicable evaluation regimes.

 Copyright © TCG 2006-2020

3 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IETF RFC 8017, Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.2

NIST SP800-56A, *Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised)*

NIST SP800-108, *Recommendation for Key Derivation Using Pseudorandom Functions (revised)*

FIPS PUB 186-3, *Digital Signature Standard (DSS)*

ISO/IEC 9797-2, Information technology -- Security techniques -- Message Authentication Codes (MACs) -- Part 2: Mechanisms using a dedicated hash-function

IEEE Std 1363™-2000, *Standard Specifications for Public Key Cryptography*

IEEE Std 1363a™-2004 (Amendment to IEEE Std 1363™-2000), IEEE *Standard Specifications for Public Key Cryptography- Amendment 1: Additional Techniques*

ISO/IEC 10116:2006, *Information technology — Security techniques — Modes of operation for an n*-bit *block cipher*

GM/T 0003.1-2012: *Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves Part 1: General*

GM/T 0003.2-2012: *Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves Part 2: Digital Signature Algorithm*

GM/T 0003.3-2012: *Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves Part 3: Key Exchange Protocol*

GM/T 0003.5-2012: *Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves Part 5: Parameter definition*

GM/T 0004-2012: *SM3 Cryptographic Hash Algorithm*

GM/T 0002-2012: *SM4 Block Cipher Algorithm*

ISO/IEC 10118-3, Information technology — Security techniques — Hash-functions — Part 3: Dedicated hash functions

ISO/IEC 14888-3, Information technology -- Security techniques -- Digital signature with appendix -- Part 3: Discrete logarithm based mechanisms

ISO/IEC 15946-1, Information technology — Security techniques — Cryptographic techniques based on elliptic curves — Part 1: General

ISO/IEC 18033-3, Information technology — Security techniques — Encryption algorithms — Part 3: Block ciphers

TCG Algorithm Registry

 Copyright © TCG 2006-2020

4 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

4.1
“ATH”
sequence of four octets of data containing 41 54 48 00_{16} that is used as a label in a KDF

NOTE See 11.4.10.2 for justification for the terminating octet of 00_{16}.

4.2
“CFB”
sequence of four octets containing 43 46 42 00_{16} that is used as a label in a KDF

NOTE See 11.4.10.2 for justification for the terminating octet of 00_{16}.

4.3
“DUPLICATE”
sequence of 10 octets containing 44 55 50 4C 49 43 41 54 45 00_{16} that is used as a label in a KDF

NOTE See 11.4.10.2 for justification for the terminating octet of 00_{16}.

4.4
“IDENTITY”
sequence of nine octets containing 49 44 45 4E 54 49 54 59 00_{16} that is used as a label in a KDF

NOTE See 11.4.10.2 for justification for the terminating octet of 00_{16}.

4.5
“OBFUSCATE”
sequence of 10 octets containing 4F 42 46 55 53 43 41 54 45 00_{16} that is used as a label in a KDF

NOTE See 11.4.10.2 for justification for the terminating octet of 00_{16}.

4.6
“SECRET”
sequence of seven octets containing 53 45 43 52 45 54 00_{16} that is used as a label in a KDF

NOTE See 11.4.10.2 for justification for the terminating octet of 00_{16}.

4.7
“STORAGE”
sequence of eight octets containing 53 54 4F 52 41 47 45 00_{16} that is used as a label in a KDF

NOTE See 11.4.10.2 for justification for the terminating octet of 00_{16}.

4.8
“XOR”
sequence of four octets containing 58 4F 52 00_{16} that is used as a label in a KDF

NOTE See 11.4.10.2 for justification for the terminating octet of 00_{16}.

 Copyright © TCG 2006-2020

4.9
ancestor
<object loaded in a TPM> Storage Key that was required to have been loaded prior to loading an object

4.10
authValue
octet string containing a value that is used for access authorization. The value is used as a password or to derive a key for an HMAC calculation.

4.11
authPolicy
digest value produced by an execution of policy commands and used for access authorization

4.12
bound
authValue of the Object is not included in the HMAC authorization for the authorization session

4.13
canonical form
data structure in the format used for transport to and from the TPM (see 4.36)

4.14
CLEAR
bit with a value of zero (0), or the action of causing a bit to have a value of zero (0)

4.15
command
discrete TPM function that is exposed externally and recognizable by a TPM's command processor; also, the values sent to the TPM to indicate the operation to be performed

4.16
commandCode
numeric identifier of the operation to be performed by a TPM

4.17
context
collection of data that provides qualifying information about a data object to differentiate it from others of the same type or to differentiate one version of a data object from another

4.18
cpHash
hash of the command code, Object names, and parameters of a command

4.19
Derivation Parent
loadable key used to derive other keys; a TPM_ALG_KEYEDHASH Parent Key

4.20
descendant
<Storage Key> Object whose loading is conditional on a specific Storage Key having been previously loaded

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

4.21
digest
result of a hash operation

4.22
duplicate
allowing a Protected Object created by a TPM to be used on a different TPM

4.23
ECDH
Diffie-Hellman secure secret sharing process using elliptic curve operations

4.24
entity
a hierarchy, PCR, object, or NV Index in a TPM shielded location

4.25
Ephemeral Key
key created as part of a protocol that is not used again after the protocol is complete

4.26
Empty Auth
Empty Buffer used as an authorization value

4.27
Empty Buffer
sized array with no data; indicated by a size field of zero followed by an array containing no elements

4.28
Empty Digest
Empty Buffer used as a digest

4.29
Empty Point
ECC point with Empty Buffers for both the x and y coordinates

4.30
Empty Policy
Empty Buffer used when a policy value is required; as a *policyValue*, an Empty Buffer will satisfy no policy

NOTE No policy can be satisfied by an Empty Policy because an Empty Policy has zero length but a *policyDigest* is the size of a hash digest and a digest is never zero length.

4.31
Endorsement Authorization
authorization using either *endorsementAuth* or *endorsementPolicy*

 Copyright © TCG 2006-2020

4.32
Extend
Extended
operation that replaces the current value of a digest with the hash of a buffer constructed by concatenating new data (normally a digest) to the current value of the digest (see 11.4.8)

4.33
External Object
Object that may be loaded into a TPM without being a member of a specific hierarchy

4.34
Failure mode
mode in which the TPM returns TPM_RC_FAILURE in response to all commands except TPM2_GetTestResult() or TPM2_GetCapability()

4.35
import
operation that allows a Protected Object not created by a TPM to be incorporated into a hierarchy of the TPM

4.36
internal form
data structure using a layout that is specific to an implementation that may or may not be the same as the canonical form

4.37
Lockout Authorization
Authorization using either *lockoutAuth* or *lockoutPolicy*

4.38
LSB0
little-endian
the least-significant octet of a datum is at byte offset 0

4.39
MSB0
big-endian
the most-significant octet of a datum is at byte offset 0

4.40
LSb0

the least-significant bit of a datum is assigned the bit number of 0

4.41
MSb0
the most-significant bit of a datum is assigned the bit number of 0

4.42
non-volatile
data that is retained even when power is removed

 Copyright © TCG 2006-2020

4.43
NULL
context-sensitive value that, when applied to a pointer, is a system-defined value indicating that the pointer does not reference data; and, when applied to a structure identified by an algorithm identifier, is the TPM_ALG_NULL value indicating that no additional data is present

4.44
NULL Password
NULL Auth
authorization where the authorization value is the Empty Buffer, resulting in an authorization that is a sequence of 9 octets containing either 40 00 00 09 00 00 00 00_{16} or 40 00 00 09 00 00 01 00 00_{16}

4.45
NULL Signature
signature with the TPM_ALG_NULL signature scheme that contains no data

4.46
NULL-terminated
sequence of non-zero values followed by a value containing zero; most often a NULL-terminated string where the values are ASCII-encoded octets

4.47
NULL Ticket
ticket structure with *tag* set to a value that is correct for the context, *hierarchy* is TPM_RH_NULL, and *digest* is an Empty Buffer

4.48
NV Index
Index
user defined non-volatile shielded location

4.49
Object
key or data that has a public portion and, optionally, a sensitive portion; and which is a member of a hierarchy

NOTE An NV Index is not an object.

4.50
octet
eight bits of data

NOTE On most modern computers, this is the smallest addressable unit of data.

4.51
orderly shutdown
when the TPM has completed TPM2_Shutdown() before power to the TPM is removed or _TPM_Init is asserted

4.52
ordinary key
key produced with a seed taken from the TPM RNG

cf. Primary Key

4.53
Owner Authorization
authorization using either *ownerAuth* or *ownerPolicy*

4.54
Parent Key
any object with the *decrypt* and *restricted* attributes SET and the *sign* attribute CLEAR

NOTE There are two types of parent keys: Storage Parent and Derivation Parent.

4.55
PCR
one or more platform configuration registers each containing a digest

4.56
PCR.alg
hash algorithm associated with a specific PCR

4.57
PCR bank
collection of PCR identified by a hash algorithm, with each PCR in the bank containing a digest computed using the bank identifier's hash algorithm

4.58
PCR.digest
digest value associated with a specific PCR

4.59
Permanent Entity
TPM resource with an architecturally defined handle that does not change

Note The value of a Permanent Entity may change

4.60
Persistent Entity
TPM resource created by a Protected Capability that persists in TPM memory across power cycles and TPM resets

4.61
Platform Authorization
authorization using either *platformAuth* or *platformPolicy*

4.62
policyDigest
digest uniquely representing an ordered set of policy commands and operands; used to determine if a policy authorizing an action has been satisfied

4.63
policySession→cpHash
policy session context value that, if not the Empty Buffer, is the *cpHash* value that the authorized command is required to have for the authorization to be valid

4.64
PolicyAuthorize Command
either TPM2_PolicyAuthorize() or TPM2_PolicyAuthorizeNV()

4.65
platform firmware
code added to the platform by its manufacturer that is needed for booting and proper platform operation

NOTE Commonly, but not exclusively, referred to as BIOS or UEFI or SMM code

4.66
Primary Key
key derived from a Primary Seed that is associated with the hierarchy of the Primary Seed

cf. ordinary key

4.67
Primary Object
Primary Key or a data blob with a sensitive area that is encrypted using a symmetric key derived from the public area of the object and a Primary Seed

4.68
private area
encrypted and integrity protected blob that contains the sensitive area of an object

4.69
Primary Seed
large random value contained within a TPM from which Primary Keys and Primary Objects are derived

4.70
Protected Capability
operation performed by the TPM on data in a Shielded Location in response to a command sent to the TPM

4.71
Protected Object
object with an encrypted sensitive portion, the sensitive portion of which the TPM will only decrypt when it is in a Shielded Location

4.72
RAM
memory that may be accessed in any order and which has no endurance limitations

 Copyright © TCG 2006-2020

4.73
reset interval
period between two successive TPM Resets and the interval during which the *resetCount* is not changed

4.74
response
values returned by the TPM when it completes processing of a command

4.75
Resume PCR
platform configuration register with a value that is preserved over a TPM Resume sequence

4.76
Root of Trust
component that must always behave in the expected manner because its misbehavior cannot be detected

NOTE The complete set of Roots of Trust has at least the minimum set of functions to enable a description of the platform characteristics that affect the trustworthiness of the platform.

4.77
rpHash
hash of the response code and the parameters of a response

4.78
Sealed Data Object
encrypted, user-defined, data blob that is associated with a hierarchy and loaded using TPM2_Load() or TPM2_CreatePrimary()

4.79
sensitive area
contain the confidential or secret parts of an object that are required to be encrypted and integrity protected when not in a Shielded Location on a TPM

4.80
sequence object
transient data structure used to hold hash state that has a handle and may be context swapped

NOTE See clause 30

4.81
session
transient TPM structure that maintains the state associated with a sequence of authorizations or an audit digest

4.82
SET
bit with a value of one (1), or the action of causing a bit to have a value of one (1)

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

4.83
Shielded Location
location on a TPM that contains data that is shielded from access by any entity other than the TPM and which may be operated on only by a Protected Capability

4.84
Shutdown(CLEAR)
abbreviated form of the command TPM2_Shutdown() with the *startupType* parameter set to TPM_SU_CLEAR

4.85
Shutdown(STATE)
abbreviated form of the command TPM2_Shutdown() with the *startupType* parameter set to TPM_SU_STATE

4.86
sizeof(x)
operator that returns the number of octets in the operand 'x'

4.87
Startup(CLEAR)
abbreviated form of the command TPM2_Startup() with the *startupType* parameter set to TPM_SU_CLEAR

4.88
Startup(STATE)
abbreviated form of the command TPM2_Startup with the *startupType* parameter set to TPM_SU_STATE

4.89
Storage Key
key used to provide integrity and confidentiality protection for descendant keys that are stored off of the TPM

4.90
Storage Parent
Storage Key that is acting as a parent key

4.91
Temporary Object
Objects that become unusable after a TPM Reset and that may not be converted into Persistent Objects

4.92
temporary resource
data object created during the execution of a command that does not persist in TPM memory after the command completes

4.93
TPM_GENERATED_VALUE
32-bit number (FF 54 43 47_{16}) that is used to tag structures that are generated by a TPM

 Copyright © TCG 2006-2020

4.94
TPM Reset
resetting of all TPM internal state to default values due to Startup(CLEAR)

4.95
TPM Resource Manager
TRM
software executing on a system with a TPM that ensures that the resources necessary to execute TPM commands are present in the TPM

4.96
TPM Restart
Startup(CLEAR) that initializes all PCR but preserves most other TPM state from the previous Shutdown(STATE)

4.97
TPM Resume
Startup(STATE) that initializes some PCR but preserves most TPM state from the previous Shutdown(STATE)

4.98
transient object
object or sequence object that may be explicitly loaded and unloaded from TPM memory by the TRM; cleared from TPM memory when the TPM is initialized (TPM2_Startup())

4.99
transient resource
object, sequence object, or session that may be explicitly loaded and unloaded from TPM memory by the TRM; cleared from TPM memory when the TPM is initialized (TPM2_Startup())

4.100
Trusted Platform Module
TPM
implementation of this specification

4.101
user-installable software
any software that may be installed on a platform other than platform firmware

4.102
volatile data
data that is lost when power is removed

4.103
Zero Digest
non-zero-length digest with all octets set to zero

5 Symbols and Abbreviated Terms

5.1 Symbols

For the purposes of this document, the following symbol definitions apply unless the text is in the `Courier` font.

A || B	concatenation of B to A
$\lceil x \rceil$	the smallest integer not less than x
$\lfloor x \rfloor$	the largest integer not greater than x
A ≔ B	assignment of the results of the expression on the right (B) to the parameter on the left
A = B	equivalence (A is the same as B)
{ A }	an optional element
A ⊕ B	bitwise exclusive OR of elements
A & B	logical AND of elements
A | B	the logical OR of elements
{A | B}	selection of elements
{A : B}	an inclusive range of elements between A and B
<A, B, … >	an ordered list of elements (a tuple)
0…0	a context-sensitive number of octets of zero
F()	denotes a function **F**
F(*p* == *x*)	denotes a function or TPM command **F** with parameter *p* set to value *x*
length(*x*)	denotes a function that returns the number of significant bits in an integer value *x*
H()	denotes the hash function
[*n*]*P*	multiplication of point *P* by the integer value *n*
A • B	multiplication of two integer values A and B
A→B	denotes a reference to element B within structure A
A mod B	A modulus B

Text in the **`Courier`** font indicates code written according to the C language standard.

5.2 Abbreviations

For the purposes of this document, the following abbreviations apply.

Abbreviation	Description
TPM	Prefix for an indication passed from the system interface of the TPM to a Protected Capability defined in this specification
AK	Attestation Key
BIOS	Basic Input/Output System
CA	Certificate Authority
CFB	Cipher Feedback mode

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

Abbreviation	Description
CPU	Central Processing Unit
CRTM	Core Root of Trust for Measurement
CTR	Counter mode
D-RTM	dynamic RTM
DA	dictionary attack
DoS	Denial of Service
DRBG	Deterministic Random Bit Generator
DSA	Digital Signature Algorithm
EA	Enhanced Authorization
EAL	evaluated assurance level
ECDAA	ECC-based Direct Anonymous Attestation
ECDH	Elliptic Curve Diffie-Hellman
EK	Endorsement Key
EPS	Endorsement Primary Seed
FIPS	Federal Information Processing Standard
FUM	Field Upgrade mode
GPIO	General Purpose I/O
HMAC	Hash Message Authentication Code
I/O	Input/Output
IV	Initialization Vector
KDF	key derivation function
KVT	known value test
LPC	Low Pin Count
LSb	Least Significant bit
LSO	Least Significant Octet
MSb	Most Significant bit
MSO	Most Significant Octet
NIST	National Institute of Standards and Technology
NP	new parent
NV	non-volatile
NVRAM	Non-Volatile Random Access Memory
OAEP	Optimal Asymmetric Encryption Padding
OEM	Original Equipment Manufacturer
OIAP	Object-Independent Authorization Protocol
OID	Object Identifier in ASN.1 format
OSAP	Object-Specific Authorization Protocol
PCR	platform configuration register(s)

Abbreviation	Description
POST	Power On Self-Test
PP	Physical Presence
PPS	Platform Primary Seed
PRF	Pseudo-Random Function
PRNG	Pseudo-Random Number Generator
PSS	Probabilistic Signature Scheme
QN	Qualified Name
RNG	Random Number Generator
RSA	Rivest, Shamir and Adleman
RTM	Root of Trust for Measurement
RTR	Root of Trust for Reporting
RTS	Root of Trust for Storage
S-RTM	Static RTM
SHA	Secure Hash Algorithm
SMAC	Symmetric block cipher Message Authentication Code
SMM	System Management Mode
SPS	Storage Primary Seed
SRK	Storage Root Key
TBB	Trusted Building Block
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TPM	Trusted Platform Module
TPM2_	Prefix for a command defined in this specification
TSS	TCG Software Stack
UEFI	Unified Extensible Firmware Interface

 Copyright © TCG 2006-2020

6 Compliance

Unless the TPM 2.0 Part 3 general description of a command indicates that the command is mandatory, a compliant TPM need not implement the command. However, if implemented, the command is required to have the behavior defined in TPM 2.0 Part 3. A platform-specific specification will indicate the commands from this specification that are required to be implemented in order to be compliant with that platform-specific specification.

The code in this specification is a reference implementation that describes required TPM behavior as observed from the TPM interface. The C-code may be reorganized or rewritten in any desired implementation language and remain compatible with this specification as long as the observable behavior is equivalent.

Even though the code in the reference implementation has undergone extensive testing, it is likely that some errors exist and one or more of those errors could lead to a TPM failure or exploit. Regardless of any other statement about normative behavior, one should not assume that a TPM exploit or failure is an intended behavior. It is not necessary to reproduce such a behavior in order to be compliant with this specification.

NOTE Please report bugs in the reference code to the TCG (admin@trustedcomputinggroup.org) so that the reference code may be brought into compliance with the specification.

The response codes in the specification are normative. An implementation performing a check prescribed by this specification is required to return the indicated error if the check fails. The order in which checks are performed is not normative. This means that a command with multiple errors could return different response codes on different TPMs. However, the response code returned is required to be the normative response code used to indicate the specific failure.

Capacities and algorithms of a TPM implementation may vary from the reference implementation; in this case, the same error would not occur in the same situation (such as, a TPM implementation with more memory may be able to satisfy a request where the reference implementation would have returned an error). However, these differences should not cause a different response code to be returned when the nature of the error is the same as in the reference implementation.

TPM 2.0 Part 4 of the specification contains major subsystems that may change for each instance of a TPM. For example, the NV subsystem of the reference implementation is not representative of the actual implementation of most physical NV implementations but is a crude analog. When the subsystem is rewritten, an equivalent interface should be provided, and the errors returned are required to match those of the reference implementation.

NOTE A constraint on the design of the TPM was the process of compliance-testing of different TPM implementations. If a TPM implementation has modularity similar to the reference implementation, then TPM tests that assume a modular design will be able to produce reliable test results on each TPM implementation.

The reference implementation uses static and stack-based allocation of resources and does not do allocations on a heap. However, a TPM implementation may use heap-based memory management in which case some error conditions and codes will differ. These differences are limited, and the allowed response codes and error conditions are defined in 39.3.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

7 Conventions

7.1 Bit and Octet Numbering and Order

An integer value is considered to be an array of one or more octets. The octet at offset zero within the array is the most significant octet (MSO) of the integer. Bit number 0 of that integer is its least significant bit and is the least significant bit in the last octet in the array.

EXAMPLE A 32-bit integer is an array of four octets; the MSO is at offset [0], and the most significant bit is bit number 31. Bit zero of this 32-bit integer is the least significant bit in the octet at offset [3] in the array.

NOTE 1 Array indexing is zero-based.

NOTE 2 This definition does not match the "network bit order" used in many IETF documents, such as RFC 4034. In those documents, the most significant bit of a datum has the lowest bit number. It is conventional practice to send that bit first when using a serial network protocol, and the bits are numbered in the order in which they are sent. This specification numbers bits according to the power of two to which they correspond within a datum. This numbering corresponds to the normal convention for bit numbering in hardware registers that hold integer values rather than fixed-point numbers.

NOTE 3 The TPM uses MSB0, LSb0 numbering.

The first listed member of a structure is at the lowest offset within the structure and the last listed member is at the highest offset within the structure.

For a character string (letters delimited by ""), the first character of the string contains the MSO.

7.2 Sized Buffer References

The specification makes extensive use of a data structure called a *sized buffer*. A sized buffer has a size field followed by an array of octets equal in number to the value in the size field.

The structure will have an identifying name. When the specification references the size field of the structure, the structure name is followed by ".size" (a period followed by the word "size"). When the specification references the octet array of the structure, the structure name is followed by ".buffer" (a period followed by the word "buffer").

7.3 Numbers

Numbers are decimal unless a different radix is indicated.

Unless the number appears in a table intended to be machine readable, the radix is a subscript following the digits of the number. Only radix values of 2 and 16 are used in this specification.

Radix 16 (hexadecimal) numbers have a space separator between groups of two hexadecimal digits.

EXAMPLE 1 $40\ FF\ 12\ 34_{16}$

Radix 2 (binary) numbers use a space separator between groups of four binary digits.

EXAMPLE 2 $0100\ 1110\ 0001_{2}$

The number of digits indicates the number of bits in the representation.

EXAMPLE 3 20_{16} is a hexadecimal number that contains exactly 8 bits and has a decimal value of 32.

EXAMPLE 4 $10\ 0000_{2}$ is a binary number that contains exactly 6 bits and has a decimal value of 32.

EXAMPLE 5 $0\ 20_{16}$ is a hexadecimal number that contains exactly 12 bits and has a decimal value of 32.

A number in a machine-readable table may use the “0x” prefix to denote a base 16 number. In this format, the number of digits is not always indicative of the number of bits in the representation.

EXAMPLE 6 0x20 is a hexadecimal number with a value of 32, and the number of bits is determined by the context.

8 Changes from Previous Versions

This version of the TPM specification introduces these additional features to the TPM family:

- Definition of an interface that allows variability of underlying cryptographic algorithms – TPM 1.2 is constrained by its data structures to using RSA and SHA1. The TPM 2.0 structure and interface defines support for a wide range of hash and asymmetric algorithms along with limited support for use of various block, symmetric ciphers. Of particular note is the addition of support for the elliptic curve (ECC) family of asymmetric algorithms.
- Unification of authorization methods – TPM 1.2 has different schemes to authorize the use, delegated use, and migration of objects. This 2.0 specification provides a uniform framework for using authorization capabilities, so they may be combined in unique ways to provide more flexibility.
- Expansion of authorization methods – TPM 2.0 allows authorization with clear-text passwords and Hash Message Authentication Code (HMAC). It also allows construction of an arbitrarily complex authorization policy for an object using multiple authorization qualifiers.
- Dedicated BIOS support – TPM 2.0 adds a Storage hierarchy controlled by platform firmware, letting the OEM benefit from the cryptographic capabilities of the TPM regardless of the support provided to the OS.
- Simplified control model – TPM 2.0 needs no special provisioning process to be useful to applications. Although objects on which the TPM operates may have limitations, all commands are available all the time. This lets application developers rely on TPM capabilities being available whenever a TPM is present.

A TPM compatible with this specification need not be compatible with previous TPM specifications.

This specification defines the operations a TPM performs and the structures used for communication between the TPM and the host system. It does not define an electrical interface to the TPM, nor does it specify which subset of TPM 2.0 commands and resources are required for a specific platform. Please refer to platform-specific TPM specifications for this information.

 Copyright © TCG 2006-2020

9 Trusted Platforms

9.1 Trust

In the context of Trusted Computing Group (TCG) specifications, “trust” is meant to convey an expectation of behavior. However, predictable behavior does not necessarily constitute behavior that is worthy of trust. For example, we expect that a bank will behave like a bank, and we expect that a thief will behave like a thief.

In order to determine the expected behavior of a platform, it is necessary to determine its identity as it relates to the platform behavior. Physically different platforms may have identical behavior. If they are constructed of components (hardware and software) that have identical behavior, then their trust properties should be the same.

The TCG defines schemes for establishing trust in a platform that are based on identifying its hardware and software components. The Trusted Platform Module (TPM) provides methods for collecting and reporting these identities. A TPM used in a computer system reports on the hardware and software in a way that allows determination of expected behavior and, from that expectation, establishment of trust.

9.2 Trust Concepts

9.2.1 Trusted Building Block

A trusted building block (TBB) is a component or collection of components required to instantiate a Root of Trust. Typically, platform-specific, a TBB is part of a Root of Trust that does not have Shielded Locations.

One example of a TBB is the combination of the CRTM, the connection between CRTM storage and a motherboard, the path between CRTM storage and the CPU, the connection between the TPM and a motherboard, and the path between the CPU and the TPM. This combination comprises the Root of Trust for Reporting (RTR).

A TBB is a component that is expected to behave in a way that does not compromise the goals of trusted platforms.

9.2.2 Trusted Computing Base

A trusted computing base (TCB) is the collection of system resources (hardware and software) that is responsible for maintaining the security policy of the system. An important attribute of a TCB is that it be able to prevent itself from being compromised by any hardware or software that is not part of the TCB.

The TPM is not the trusted computing base of a system. Rather, a TPM is a component that allows an independent entity to determine if the TCB has been compromised. In some uses, the TPM can help prevent the system from starting if the TCB cannot be properly instantiated.

9.2.3 Trust Boundaries

The combination of TBB and Roots of Trust form a trust boundary, within which measurement, storage, and reporting may be accomplished for a minimal configuration. In systems that are more complex, it may be necessary for the CRTM to establish trust in other code, by making measurements of that other code and recording the measurement in a PCR. If the CRTM transfers control to that other code regardless of the measurement, then the trust boundary is expanded. If the CRTM will not run that code unless its

 Copyright © TCG 2006-2020

measurement is the expected value, the trust boundary remains the same because the measured code is an expected extension of the CRTM.

9.2.4 Transitive Trust

Transitive trust is a process whereby the Roots of Trust establish the trustworthiness of an executable function, and trust in that function is then used to establish the trustworthiness of the next executable function.

Transitive trust may be accomplished either by: (1) knowing that a function enforces a trust policy before it allows a subsequent function to take control of the TCB, or (2) using measurements of subsequent functions so that an independent evaluation may establish the trust. The TPM may support either of these methods.

9.2.5 Trust Authority

When the RTM begins to execute the CRTM, the entity that may vouch for the correctness of the TBB is the entity that created the TBB. For typical systems, this is the platform manufacturer. In other words, the manufacturer is the authority on what constitutes a valid TBB, and its reputation is what allows someone to trust a given TBB.

As the system transitions to code outside the CRTM, the transitive trust chain is maintained by measurement of that code. If execution of that code is conditional on its measurement, then the authority for that code remains unchanged. That is, if the platform manufacturer's CRTM does not run code outside the CRTM unless that code has a specific measured value, then the platform manufacturer remains the trust authority regardless of who provided that code.

In modern architectures, where firmware and software components come from many different suppliers, it is often not feasible for platform manufacturers to know the signers of all code that runs on a platform. Therefore, they may not remain the authority on platform state for very long. The measurements recorded in the RTS then determine the chain of authority for the current system state.

Two different methods allow evaluation of the trust authority for a platform.

1) Code is measured (hashed), and its value is recorded in the RTS. If the code is run regardless of its measurement, then the authority for the trust is the digest of the code reported by the RTR. That is, the measurements speak for themselves, and the verifier needs either to have knowledge of the measurements that constitute trustworthy code or knowledge of the measurements that indicates malicious or vulnerable code.
2) Code is signed so that the identity of the authority for the code is known. If this identity is recorded in the RTS, the evaluation can be changed. Instead of being based on knowing the digest of the code, it can be based on identities of the signers of the code.

Because trusted sources of code may sometimes produce code with security vulnerabilities, support for revocation is often required. To allow revocation of specific code modules, it is often necessary to use a hybrid solution where both authorities and details are recorded. This simplifies the process of determining whether a module from a specific vendor has been revoked.

NOTE If the code is measured (hashed) and not signed, it is harder to know if a specific measurement is valid unless there is a centralized database of all known digests of revoked code. When the identity of the authority is known, one can contact the vendor to determine if it has revoked code with a given hash.

 Copyright © TCG 2006-2020

9.3 Trusted Platform Module

A TPM is a system component that has state that is separate from the system on which it reports (the host system). The only interaction between the TPM and the host system is through the interface defined in this specification.

TPMs are implemented on physical resources, either directly or indirectly. A TPM may be constructed using physical resources that are permanently and exclusively dedicated to the TPM, and/or using physical resources that are temporarily assigned to the TPM. All of a TPM's physical resources may be located within the same physical boundary, or different physical resources may be within different physical boundaries.

Some TPMs are implemented as single-chip components that are attached to systems (typically, a PC) using a low-performance interface (such as, Low Pin Count, or LPC). The TPM component has a processor, RAM, ROM, and Flash memory. The only interaction with these TPMs is through the LPC bus. The host system cannot directly change the values in TPM memory other than through the I/O buffer that is part of the interface.

Another reasonable implementation of a TPM is to have the code run on the host processor while the processor is in a special execution mode. For these TPMs, parts of system memory are partitioned by hardware so that the memory used by the TPM is not accessible by the host processor unless it is in this special mode. Further, when the host processor switches modes, it always begins execution at specific entry points. This version of a TPM would have many of the same attributes as the stand-alone component in that the only way for the host to cause the TPM to modify its internal state is with well-defined interfaces. There are several different schemes for achieving this mode switching including System Management Mode, Trust Zone™, and processor virtualization.

Definition of the interaction between the host and the TPM is the primary objective of this specification. Prescribed commands instruct the TPM to perform prescribed actions on data held with the TPM. A primary purpose of these commands is to allow determination of the trust state of a platform. The ability of a TPM to accomplish its objective depends on the proper implementation of Roots of Trust.

9.4 Roots of Trust

TCG-defined methods rely on Roots of Trust. These are system elements that must be trusted because misbehavior is not detectable. The set of roots required by the TCG provides the minimum functionality necessary to describe characteristics that affect a platform's trustworthiness.

While it is not possible to determine if a Root of Trust is behaving properly, it is possible to know how roots are implemented. Certificates provide assurances that the root has been implemented in a way that renders it trustworthy. For example, a certificate may identify the manufacturer and evaluated assurance level (EAL) of a TPM. This certification provides confidence in the Roots of Trust implemented in the TPM. In addition, a certificate from a platform manufacturer may provide assurance that the TPM was properly installed on a machine that is compliant with TCG specifications so that the Root of Trust provided by the platform may be trusted (see 9.5.2 for more information on certification).

The TCG requires three Roots of Trust in a trusted platform:

- Root of Trust for Measurement (RTM),
- Root of Trust for Storage (RTS), and
- Root of Trust for Reporting (RTR).

Trust in the Roots of Trust can be achieved through a variety of means but is anticipated to include technical evaluation by competent experts.

 Copyright © TCG 2006-2020

9.4.1 Root of Trust for Measurement (RTM)

The RTM sends integrity-relevant information (measurements) to the RTS. Typically, the RTM is the CPU controlled by the Core Root of Trust for Measurement (CRTM). The CRTM is the first set of instructions executed when a new chain of trust is established. When a system is reset, the CPU begins executing the CRTM. The CRTM then sends values that indicate its identity to the RTS. This establishes the starting point for a chain of trust (see 9.5.5 for a more detailed description of integrity measurement).

9.4.2 Root of Trust for Storage (RTS)

The TPM memory is shielded from access by any entity other than the TPM. Because the TPM can be trusted to prevent inappropriate access to its memory, the TPM can act as an RTS.

Some of the information in TPM memory locations is not sensitive and the TPM does not protect it from disclosure. An example of non-sensitive data is the current contents of a platform configuration register (PCR) containing a digest. Other information is sensitive and the TPM does not allow access to the information without proper authority. An example of sensitive data in a Shielded Location is the private part of an asymmetric key.

Sometimes, the TPM uses the contents of one Shielded Location to gate access to another Shielded Location. For example, access to (use of) a private key for signing may be conditioned on PCR having specific values.

9.4.3 Root of Trust for Reporting (RTR)

9.4.3.1 Description

The RTR reports on the contents of the RTS. An RTR report is typically a digitally signed digest of the contents of selected values within a TPM.

NOTE Not all Shielded Locations are directly accessible. For example, the values of the private part of keys and authorizations are in Shielded Locations on which the TPM will not report.

The values on which the RTR reports typically are

- evidence of a platform configuration in PCR (such as, TPM2_Quote()),
- audit logs (such as, TPM2_GetCommandAuditDIgest ()), and
- key properties (such as, TPM2_Certify()).

The interaction between the RTR and RTS is critical. The design and implementation of this interaction should mitigate tampering that would prevent accurate reporting by the RTR. An instantiation of the RTS and RTR will

- be resistant to all forms of software attack and to the forms of physical attack implied by the TPM's Protection Profile, and
- supply an accurate digest of all sequences of presented integrity metrics.

9.4.3.2 Identity of the RTR

The TPM contains cryptographically verifiable identities for the RTR. The identity is in the form of asymmetric aliases (Endorsement Keys or EKs) derived from a common seed. Each seed value and its aliases should be statistically unique to a TPM. That is, the probability of two TPMs having the same EK should be insignificant.

 Copyright © TCG 2006-2020

The seed may be used to generate multiple asymmetric keys, all of which would represent the same TPM and RTR.

9.4.3.3 RTR Binding to a Platform

The TPM reports on the state of the platform by quoting the PCR values. For assurance that these PCR values accurately reflect that state, it is necessary to establish the binding between the RTR and the platform. A Platform Certificate can provide proof of this binding. The Platform Certificate is assurance from the certifying authority of the physical binding between the platform (the RTM) and the RTR.

9.4.3.4 Platform Identity and Privacy Considerations

The uniqueness of an EK and its cryptographic verifiability raises the issue of whether direct use of that identity could result in aggregation of activity logs. Analysis of the aggregated activity could reveal personal information that a user of a platform would not otherwise approve for distribution to the aggregators.

To counter undesired aggregation, TCG encourages the use of domain-specific signing keys and restrictions on the use of an EK. The Privacy Administrator controls use of an EK, including the process of binding another key to the EK.

NOTE Privacy Administrator's control of the EK differs from Owner control of the RTS providing separation of the security and identity uses of the TPM.

Unless the EK is certified by a trusted entity, its trust and privacy properties are no different from any other asymmetric key that can be generated by pure software methods. Therefore, by itself, the public portion of the EK is not privacy sensitive.

9.5 Basic Trusted Platform Features

9.5.1 Introduction

At a minimum, a trusted platform provides the three Roots of Trust described previously. All three roots use certification and attestation to provide evidence of the accuracy of information. A trusted platform will also offer Protected Locations (see 10.3) for the keys and data objects entrusted to it. Finally, a trusted platform may provide integrity measurement to ensure the trustworthiness of a platform by logging changes to platform state; this is done by recording logged entries in PCR for later validation as being correct and unaltered. These basic TPM concepts are now described in detail.

9.5.2 Certification

The nominal method of establishing trust in a key is with a certificate indicating that the processes used for creating and protecting the key meets necessary security criteria. A certificate may be provided by shipping the TPM with an embedded key (that is, an Endorsement Key) along with a Certificate of Authenticity for the EK. The EK and its certificate may be used to associate credentials (certificates) with other TPM keys; this process is described in 9.5.3.3. When a certified key has attributes that let it sign TPM-created data, it may attest to the TPM-resident record of platform characteristics that affect the integrity (trustworthiness) of a platform.

NOTE The EK does not have to be installed when the TPM is shipped. At the factory, an EK may be generated from the Endorsement Seed and a Certificate of Authenticity created for that EK. The EK does not have to be permanently installed in the TPM. When the TPM is in possession of a customer, the customer may, at their discretion, have the TPM use the Endorsement Seed and recreate the EK for which they have a Certificate of Authenticity.

 Copyright © TCG 2006-2020

9.5.3 Attestation and Authentication

9.5.3.1 Types of Attestation

Trusted platforms employ a hierarchy of attestations:

1) An external entity attests to a TPM in order to vouch that the TPM is genuine and complies with this TPM specification. This attestation takes the form of an asymmetric key embedded in a genuine TPM, plus a credential that vouches for the public key of that pair.

NOTE 1 A credential that is used to vouch for the embedded asymmetric key is commonly called an "Endorsement Certificate."

2) An external entity attests to a platform in order to vouch that the platform contains a Root-of-Trust-for-Measurement, a genuine TPM, plus a trusted path between the RTM and the TPM. This attestation takes the form of a credential that vouches for information including the public key of the asymmetric key pair in the TPM.

NOTE 2 A credential used to vouch for the platform is commonly called a "Platform Certificate."

3) An external entity called an "Attestation CA" attests to an asymmetric key pair in a TPM in order to vouch that a key is protected by an unidentified but genuine TPM and has particular properties. This attestation takes the form of a credential that vouches for information including the public key of the key pair. An Attestation CA typically relies upon attestations of type 1 and 2 in order to produce attestation of type 3.

NOTE 3 The credential created by the CA is commonly called an "Attestation Key Certificate."

4) A trusted platform attests to an asymmetric key pair in order to vouch that a key pair is protected by a genuine but unidentified TPM and has particular properties. This attestation takes the form of a signature signed by the platform's TPM over information that describes the key pair, using an attestation-key protected by the TPM, plus attestation of type 3 that vouches for that attestation key.

NOTE 4 This type of attestation is done using TPM2_Certify().

5) A trusted platform attests to a measurement in order to vouch that a particular software/firmware state exists in a platform. This attestation takes the form of a signature over a software/firmware measurement in a PCR using an attestation key protected by the TPM, plus attestation of type 3 or 4 for that attestation key.

NOTE 5 This is type of attestation is commonly called a "quote" and is done with TPM2_Quote().

6) An external entity attests to a software/firmware measurement in order to vouch for particular software/firmware. This attestation takes the form of a credential that vouches for information including the value of a measurement and the state it represents.

NOTE 6 This is commonly called "third-party certification."

Attestation of types 3 and 4 entail the use of a key to sign the contents of Shielded Locations. An Attestation Key (AK) is a particular type of signing key that has a restriction on its use, in order to prevent forgery (the signing of external data that has the same format as genuine attestation data). The restriction is that an AK may be used only to sign a digest that the TPM has created. If an AK is known to be protected by a TPM (by virtue of attestation of type 3 or 4), it may be relied on to report accurately on Shielded Location content, and not sign externally provided data that appears to be valid and TPM-produced but is not.

 Copyright © TCG 2006-2020

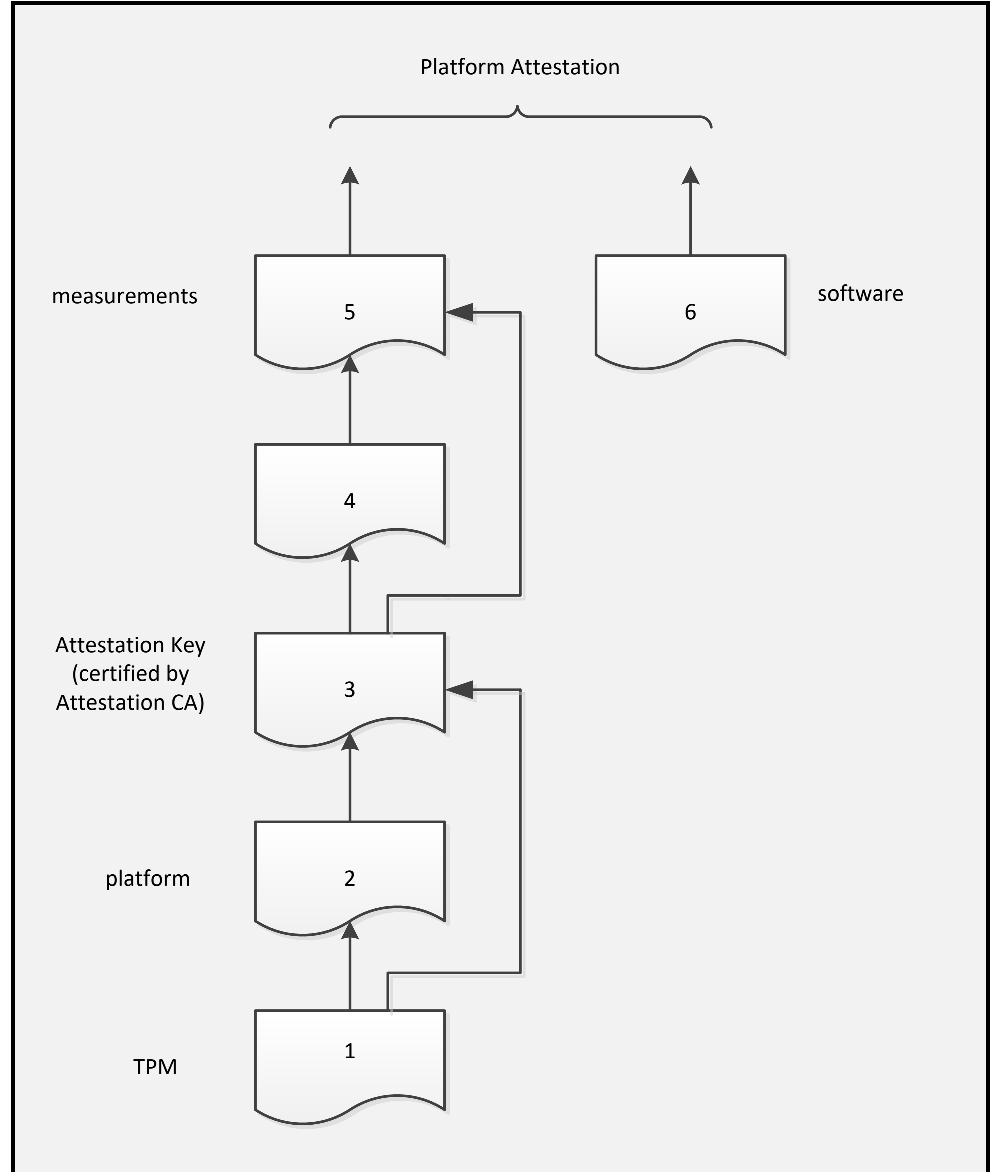


Figure 1 — Attestation Hierarchy

9.5.3.2 Attestation Keys

When the TPM creates a message to sign from internal TPM state (such as, in TPM2_Quote()), a special value (TPM_GENERATED_VALUE) is used as the message header. A TPM-generated message always begins with this value.

When the TPM digests an externally provided message, it checks the first few octets of the message to ensure that they do not have the same value as TPM_GENERATED_VALUE. When the digest is complete, the TPM produces a ticket that indicates the message did not start with TPM_GENERATED_VALUE. When an AK is used to sign the digest, the caller provides the ticket so that the TPM can determine that the message used to create the digest was not a possible forgery of TPM attestation data.

NOTE The digest in the ticket must match the digest being presented to the AK for signing.

EXAMPLE If an attacker produced a message block that was identical to a TPM-generated quote, that message block would start with TPM_GENERATED_VALUE to indicate that it is a proper TPM quote. When the TPM performs a digest of this block, it notes that the first octets are the same as TPM_GENERATED_VALUE. It will not generate the ticket indicating that the message is safe to sign, so an AK may not be used to sign this digest. Similarly, an entity checking an attestation made by an AK must verify that the message signed begins with TPM_GENERATED_VALUE in order to verify the message is indeed a TPM-generated quote.

Values signed by an AK may be assured to reflect TPM state, but AKs may also be used for general signing purposes.

An AK does not have much value to a remote challenger if the AK cannot be associated with the platform that it represents. This association is made using the identity certification process.

9.5.3.3 Attestation Key Identity Certification

Any TPM user that can create a key on a TPM can create a restricted-use signing key. The key creator may then ask a third party, such as an attestation Certificate Authority (CA), to provide a certificate for it. The attestation CA may request that the caller provide some evidence that the key being certified is a TPM-resident key.

Evidence of TPM residency may be provided using a previously generated certificate for another key on the same TPM. An EK or Platform Certificate may provide this evidence.

NOTE 1 There is no requirement that certificates come only from an attestation CA. The method described above is an example of a scheme that may be used when privacy is required.

If a certified key may sign, it may be used to certify that some other object is resident on the same TPM. This allows the new AK to be linked to a certified key. A CA may use the certification from the TPM to produce a traditional certificate for the new key.

If the certified key is a decryption key and may not sign, then an alternative method is used to allow the new key or data object to be reliably certified. For this alternative certification, the identity of the Object to be certified and a certificate for the decryption key (such as, an EK) are provided to the CA. From the certificate, the CA determines the public key for the decryption key. The CA then produces a challenge for the Object to be certified and encrypts the challenge with the certified key. The encrypted challenge is given to the TPM containing both the certified decryption key and the key to be certified.

The challenge is protected using methods that are dependent on the type of the certified decryption key. The general method is described in clause 24. Additional methods appropriate to RSA keys are described in B.10.4 and additional methods appropriate to ECC key are in C.6.4. The protection process produces an encrypted blob, an HMAC over the blob, and a secret value that can only be recovered by the certified decryption key.

TPM2_ActivateCredential() is used to access the challenge. The TPM recovers the secret value and uses it to generate the keys necessary to decrypt and validate the HMAC and encrypted blob. If the challenge is recovered successfully, and the key being certified by the credential is loaded on the TPM, then the challenge is returned to the caller, and then provided to the CA. After the CA validates the challenge, it can issue the certificate for the key

NOTE 3 The protection process used for the challenge is almost identical to the process used for key import. In order to make sure that there is no misuse of the encrypted structures, an application-specific value is used in the key recovery process. In the case of a challenge, the label "IDENTITY" is used in the KDF that generates the keys (symmetric and HMAC) from the seed value.

TPM2_ActivateCredential() can operate on any Object. The choice of attributes for an Object to be certified is at the discretion of the CA. Because a unique identifier for the Object is included in the integrity

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

hash, the TPM enforces the challenge's accessibility only if the Object matches the criteria set by the CA as expressed in the object identifier.

9.5.4 Protected Location

When the sensitive portion of an object is not held in a Shielded Location on the TPM, it is encrypted. When encrypted, but not on the TPM, it is not protected from deletion, but it is protected from disclosure of its sensitive portions. Wherever it is stored, it is in a Protected Location.

Objects in long-term protected storage need to be loaded into the TPM for use. The application that created the objects manages their movement from long-term storage to the TPM.

Since a TPM has limited memory, it may be unable to hold all objects required by all applications simultaneously. The TPM supports swapping of object contexts by a TPM Resource Manager (TRM) so that the TPM can service these multiple applications. The object contexts are encrypted before being returned to the TRM by the TPM. If the object is needed later, the TRM can reload the context into the TPM providing a cache-like behavior.

Encryption of Protected Locations uses multiple seeds and keys that never leave the TPM. One of these is the Context Key. It is a symmetric key used to encrypt data when it is temporarily swapped out of the TPM so that a different working set of objects may be loaded. Other sensitive values that never leave the TPM are the Primary Seeds. These seeds are the root of the storage hierarchies that protect objects that are retained by applications. A Primary Seed is a random number used to generate protection keys for other objects; these objects may be Storage Keys that contain protection keys that are then used to protect still more objects.

Primary Seeds may be changed, and when they are changed, the objects they protected will no longer be usable. For example, the Storage Primary Seed (SPS) creates the Storage hierarchy for owner-related data, and that seed changes when the owner changes.

9.5.5 Integrity Measurement and Reporting

The Core Root of Trust for Measurement (CRTM) is the starting point of measurement. This process makes the initial measurements of the platform that are Extended into PCR in the TPM. For measurements to be meaningful, the executing code needs to control the environment in which it is running, so that the values recorded in the TPM are representative of the initial trust state of the platform.

A power-on reset creates an environment in which the platform is in a known initial state, with the main CPU running code from some well-defined initial location. Since that code has exclusive control of the platform at that time, it may make measurements of the platform from firmware. From these initial measurements, a chain of trust may be established. Because this chain of trust is created once when the platform is reset, no change of the initial trust state is possible, so it is called a static RTM (S-RTM).

An alternative method of initializing the platform is available on some processor architectures. It lets the CPU act as the CRTM and apply protections to portions of memory it measures. This process lets a new chain of trust start without rebooting the platform. Because the RTM may be re-established dynamically, this method is called dynamic RTM (D-RTM). Both S-RTM and D-RTM may take a system in an unknown state and return it to a known state. The D-RTM has the advantage of not requiring the system to be rebooted.

An integrity measurement is a value that represents a possible change in the trust state of the platform. The measured object may be anything of meaning but is often

- a data value,

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

- the hash of code or data, or
- an indication of the signer of some code or data.

The RTM (usually, code running on the CPU) makes these measurements and records them in RTS using Extend. The Extend process (see 17.2) allows the TPM to accumulate an indefinite number of measurements in a relatively small amount of memory.

The digest of an arbitrary set of integrity measurements is statistically unique, and an evaluator might know the values representing particular sequences of measurements. To handle cases where PCR values are not well known, the RTM keeps a log of individual measurements. The PCR values may be used to determine the accuracy of the log, and log entries may be evaluated individually to determine if the change in system state indicated by the event is acceptable.

Implementers play a role in determining how event data is partitioned. TCG's platform-specific specifications provide additional insight into specifying platform configuration and representation as well as anticipated consumers of measurement data.

Integrity reporting is the process of attesting to integrity measurements recorded in a PCR. The philosophy behind integrity measurement, logging, and reporting is that a platform may enter any state possible — including undesirable or insecure states — but is required to accurately report those states. An independent process may evaluate the integrity states and determine an appropriate response.

 Copyright © TCG 2006-2020

10 TPM Protections

10.1 Introduction

This part of the specification describes the protections provided by the Trusted Platform Module. This clause describes the properties of selected capabilities and selected data locations for a TPM that has been evaluated according to a Protection Profile and a TPM that has not been modified by physical means.

TPM protections are based on the concepts of Protected Capabilities and Protected Objects. A Protected Capability is an operation that must be performed correctly for a TPM to be trusted. A Protected Object is data (including keys) that must be protected for a TPM operation to be trusted. Protected Objects in the TPM reside in Shielded Locations; the TPM may manipulate the contents of Shielded Locations only by using Protected Capabilities. Protected Objects outside Shielded Locations have their integrity and confidentiality protected cryptographically.

Since a Protected Object may reside outside of Shielded Location protections, the definition of “access” to a Protected Object denotes disclosure of its contents, not modification. Such objects are not protected against loss or tampering. However, before loading a Shielded Location with an outside object, the TPM will use a secure hash function to validate that the object was properly protected and not altered. If the integrity check fails, the TPM returns an error and does not load the object.

The only operations on Shielded Locations of a TPM are the Protected Capabilities defined in this specification and the vendor-specific operations that meet the requirements of 10.4.

10.2 Protection of Protected Capabilities

A Protected Capability may be modified only by other Protected Capabilities in the same TPM. Thus, the process of updating TPM firmware is required to be a Protected Capability.

10.3 Protection of Shielded Locations

As noted, access to any data on a TPM requires use of a Protected Capability. Therefore, all information on a TPM is in a Shielded Location. The contents of a Shielded Location are not disclosed unless the disclosure is intended by the definition of the Protected Capability. A TPM is not allowed to export data from a Shielded Location other than by using a Protected Capability.

NOTE Data in an I/O buffer that can be modified by the host is not “on” the TPM, even though the I/O buffer may be shielded from access while the TPM is processing a command or generating a response.

10.4 Exceptions and Clarifications

Vendor-specific operations may access and modify Shielded Locations on a TPM under the following circumstances.

- A vendor-specific operation may use the standard TPM authorization mechanism.
- A vendor-specific capability may read any TPM-resident structure that is not required to be in a Shielded Location at all times if the usage of that structure is authorized per the structure’s authorization mechanism.

EXAMPLE A vendor-specific command may use the public portion of a key. If the key is a user key, no authorization would be required.

 Copyright © TCG 2006-2020

NOTE Among other things, the exception above enables access to a Shielded Location, so the structure's access authorization may be checked.

- Vendor-specific operations may use a sequence of Protected Capabilities.
- Vendor-specific operations may use the standard TPM command interface or use a vendor-defined interface.

These clarifications serve to approve specific legitimate interpretations of the requirements.

- A vendor-specific operation that takes advantage of exceptions and clarifications to the "protection" requirements should be defined as part of the security target of the TPM. Such a vendor-specific command or capability should be evaluated to determine whether it meets Platform-specific TPM and System Security Targets.
- If a TPM stores vendor-specific cipher-text that is protected against subversion to the same or greater extent as internal TPM-resources stored outside the TPM with TCG-defined methods, then that cipher-text does not require protection from physical attack. If the TPM stores only vendor-specific cipher-text that does not require protection from physical attack, that location may be excluded from analysis when determining whether the TPM complies with the "physical protection" requirements specified by TCG.
- If a TPM uses external memory for non-volatile storage of TPM state (including seeds and proof values), movement of the TPM state to and from the NV memory constitutes a vendor-defined operation that is allowed by this specification. The protection profile of that TPM should include a description of the protections of that data to ensure confidentiality and integrity of the data and to mitigate against rollback attacks.

 Copyright © TCG 2006-2020

11 TPM Architecture

11.1 Introduction

This clause describes the overall operation of the TPM and the functional units required for its operation. The major elements of the architecture are shown in Figure 2.

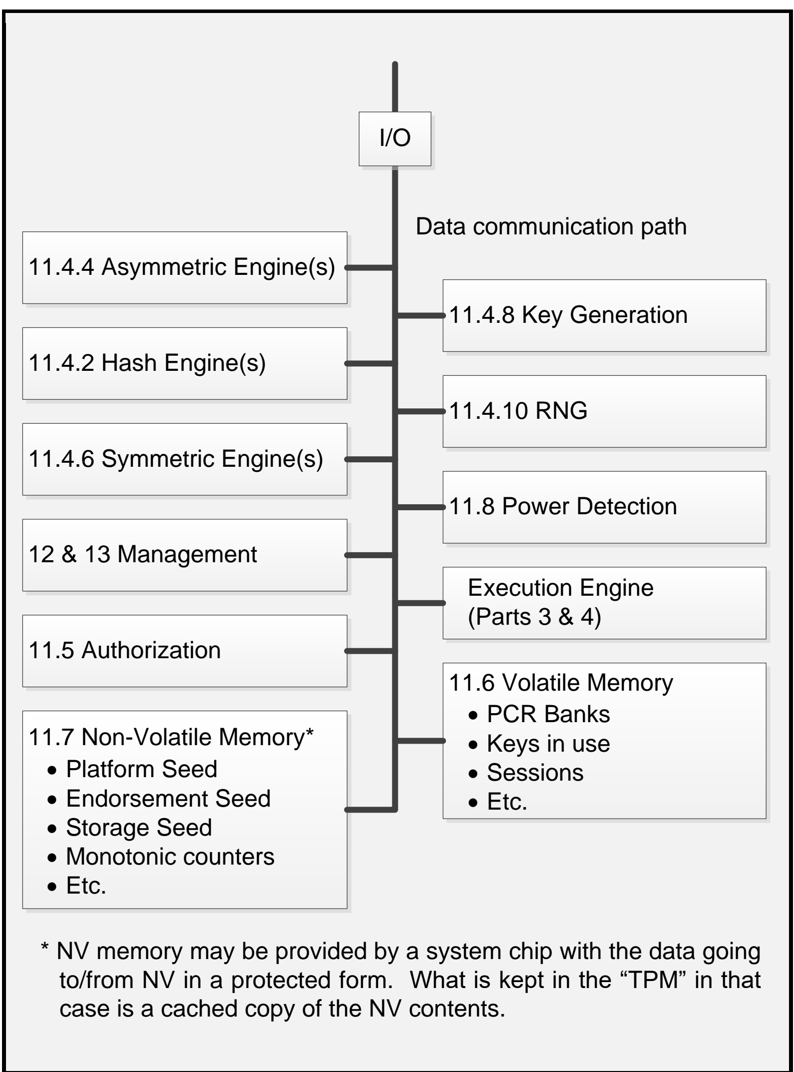


Figure 2 — Architectural Overview

11.2 TPM Command Processing Overview

Figure 3 is a high-level flow diagram for a TPM command. The figure shows only the normal flow for a command that executes successfully. The tabs on a box indicate the name of the module performing the operation. Additional details for each of the modules shown in Figure 3 are in this clause and in clauses dedicated to those modules.

The partitioning of functions in Figure 3 is illustrative and not normative.

 Copyright © TCG 2006-2020

The flow assumes that the command has been placed in an input buffer that is accessible to the Execute Command module (this name is used because of its similarity to the ExecCommand() function in the reference code that performs the functions illustrated here).

NOTE 1 The mechanism for getting the command into the TPM buffer and providing the command-available indication is specific to each physical interface and is defined in interface-specific documents.

The command structure includes a standard header (see 18) that Execute Command validates. It then determines if the command requires access to any Shielded Location that is identified by a handle. If so, it calls the Handle module to verify that the handle references the right type of resource for the command and that the resource is currently loaded on the TPM.

When control returns to Execute Command, it checks the *tag* parameter in the command header to determine if authorization values are provided. If so, Authorizations is called to validate that each of the authorizations is correct. The authorizations are associated with a handle value, so the authorization is specific to a particular entity.

After validating the authorizations, Execute Command calls Command Dispatch to unmarshal the remaining command parameters and validate that the required parameters of the required type are present. All parameters are validated to meet the requirements of its data type as defined in TPM 2.0 Part 2 even if the parameter will subsequently be discarded because of optional behavior of a command.

After unmarshaling the parameters, Command Dispatch calls the command-specific library function to execute the specific command. Additional parameter checking may be required in the command-specific actions.

The command processing is structured so that changes to the TPM state do not occur until the TPM can validate that the command parameters are correct and that the resources necessary to complete the command are available. Only then will it make irreversible changes to the TPM state. This structuring ensures that when the TPM returns an error, the TPM will be in the same state as before command actions modified the data in any Shielded Location.

NOTE 2 Requiring that the TPM retain its state minimizes the interference between applications and helps prevent system instability due to careless use of the TPM by applications.

There are several classes of operations that return an error but may change TPM state.

- An authorization failure may update the dictionary attack mechanism.
- The self test mechanism has state (for example, which algorithms have been tested) that is considered to be different from the command execution state. Changes to this state may occur regardless of the command return code. For example, an implicit self test invoked to test an algorithm required by the command may mark the algorithm as tested.
- If a self test fails, the TPM will go into Failure Mode.

When the command actions are complete, the Command Dispatch marshals response parameters into the output buffer. If the command had authorizations, Acknowledge is called to construct acknowledge session values for the response.

If the command encounters an error, the response packet will contain a code that is characteristic of the error and, when possible, an indication of whether the error was associated with a handle, an authorization session, or a command parameter. No additional qualifying data is present. In most cases, the error code and parameter location value suffice to isolate the problem.

NOTE 3 In the case of a self-test failure, the TPM response code is not sufficient to diagnose the problem. Therefore, a reporting scheme is provided so that the failure cause can be read. However, error report contents vary by vendor and are not standardized. There is thus no need to standardize self-test response codes because no standard remediation is possible for most self-test failures.

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

After constructing the response, including acknowledge sessions, the TPM indicates to the interface that the response is ready to be returned.

The TPM command/response structure is described in Clause 17.10 (see clause 19 for a description of the methods for creating the values that authorize use of a TPM Shielded Location and clause 39 for response code formatting information).

During the processing of these commands, the TPM uses other modules that the following parts of this clause will describe.

 Copyright © TCG 2006-2020

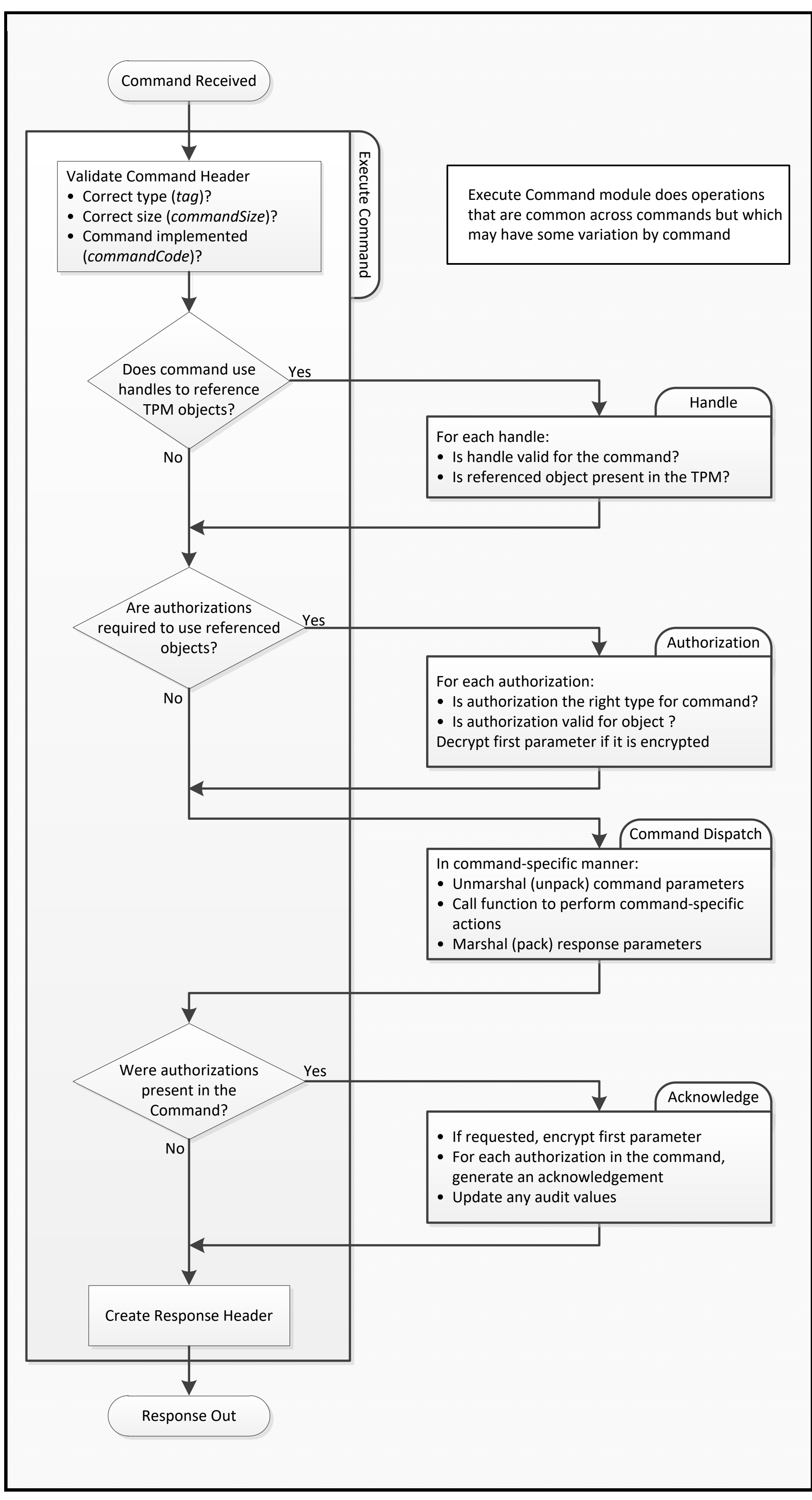

Command Received
Execute Command
Validate Command Header
• Correct type (tag)?
• Correct size (commandSize)?
• Command implemented (commandCode)?
Execute Command module does operations that are common across commands but which may have some variation by command
Does command use handles to reference TPM objects?
Yes
No
Handle
For each handle:
• Is handle valid for the command?
• Is referenced object present in the TPM?
Are authorizations required to use referenced objects?
Yes
No
Authorization
For each authorization:
• Is authorization the right type for command?
• Is authorization valid for object ?
Decrypt first parameter if it is encrypted
Command Dispatch
In command-specific manner:
• Unmarshal (unpack) command parameters
• Call function to perform command-specific actions
• Marshal (pack) response parameters
Were authorizations present in the Command?
Yes
No
Acknowledge
• If requested, encrypt first parameter
• For each authorization in the command, generate an acknowledgement
• Update any audit values
Create Response Header
Response Out

Figure 3 — Command Execution Flow

11.3 I/O Buffer

The I/O buffer is the communications area between a TPM and the host system. The system places command data in the I/O buffer and retrieves response data from the buffer.

A description of the physical processes used to move I/O buffer data to/from the system is beyond the scope of this specification. Platform-specific working groups within the TCG produce the specifications for the physical interfaces to the TPM on their platforms. Those specifications detail the interactions between system software and the TPM I/O buffer.

There is no requirement that the I/O buffer be physically isolated from other parts of the system. It can be a shared memory. However, when processing of a command begins, the implementation must ensure that the TPM is using the correct values. For example, if the TPM performs a hash of the command data as part of the authorization processing, the TPM needs to protect the validated command data from modification. That is, before the data is validated, it is required to be protected from modification. Before the data is modified, it is required to be in a Shielded Location.

11.4 Cryptography Subsystem

11.4.1 Introduction

The Cryptography subsystem implements the TPM's cryptographic functions. It may be called by the Command Parsing module, the Authorization Subsystem, or the Command Execution module. The TPM employs conventional cryptographic operations in conventional ways. These operations include

- hash functions,
- asymmetric encryption and decryption,
- asymmetric signing and signature verification,
- symmetric encryption and decryption,
- symmetric signing (HMAC and SMAC) and signature verification, and
- key generation.

The remainder of this clause describes some algorithms usually found in a TPM to show how they are handled. These descriptions illustrate, but do not limit, the choice of available algorithms.

11.4.2 Symmetric Block Cipher MAC Algorithms

The TPM may implement Symmetric Block Cipher Message Authentication Code (SMAC).

An SMAC is a form of symmetric signature over some data using a symmetric block cipher algorithm. It provides assurance that protected data was not modified and that it came from an entity with access to a key value. To have usefulness in protecting data, the key value needs to be a secret or a shared secret.

11.4.3 Hash Functions

Hash functions may be used directly by external software or as the side effect of many TPM operations. The TPM uses hashing to provide integrity checking and authentication as well as one-way functions, as needed (such as, KDF).

A TPM should implement an approved hash algorithm that has approximately the same security strength as its strongest asymmetric algorithm.

EXAMPLE An ECC with a 384-bit key has a security strength of 192 bits. SHA384, with 192 bits of security, would meet the preceding requirement above.

NOTE The TCG may create sets of algorithms that do not have the same security strength for the hash and asymmetric algorithms.

A hash function will be denoted by $\mathbf{H}_{algorithm}()$ with the algorithm subscript indicating the hash algorithm or the parameter that contains the hash algorithm identifier. In some cases, the algorithm subscript is missing, in which case the algorithm will be determined by context.

The Command Dispatch module will use the hash function when validating certain types of authorizations. Hash functions are also used in support of other operations in the TPM such as PCR Extend.

11.4.4 HMAC Algorithm

The TPM implements the Hash Message Authentication Code (HMAC) algorithm described in ISO/IEC 9797-2.

An HMAC is a form of symmetric signature over some data. It provides assurance that protected data was not modified and that it came from an entity with access to a key value. To have usefulness in protecting data, the key value needs to be a secret or a shared secret.

ISO/IEC 9797-2 defines the HMAC operation as:

$$\mathbf{HMAC}(K, text) = \mathbf{H}((\bar{K} \oplus OPAD) \mid\mid \mathbf{H}((\bar{K} \oplus IPAD) \mid\mid text)) \quad (1)$$

(See ISO/IEC 9797-2 for a description of parameters.)

Performing the HMAC computation requires selection of a hash algorithm. This specification modifies the notation from ISO/IEC 9797-2 to be:

$$\mathbf{HMAC}_{hashAlg}(K, text) \quad (2)$$

If the algorithm subscript is not present, the hash algorithm is implied by the context.

The Command Dispatch module may use the HMAC function to validate an authorization. The HMAC function may be used by the Command Execution module in support of its operations.

11.4.5 Asymmetric Operations

A TPM uses asymmetric algorithms for attestation, identification, and secret sharing. A TPM may support any asymmetric algorithm to which the TCG has assigned an identifier. An asymmetric algorithm identifier will indicate a family of algorithms and methods that are used with that algorithm.

The methods for using an asymmetric algorithm are found in algorithm-specific annexes to this TPM 2.0 Part 1. Currently, the only supported asymmetric algorithms are RSA (described in Annex B) and ECC using prime curves (described in Annex B).

A TPM is required to implement at least one asymmetric algorithm.

 Copyright © TCG 2006-2020

11.4.6 Signature Operations

11.4.6.1 Signing

The TPM may sign using either an asymmetric or a symmetric algorithm. The method of signing depends on the type of the key. For an asymmetric algorithm, the methods of signing are dependent on the algorithm (RSA or ECC). For symmetric signatures, HMAC and SMAC signing schemes are defined. If a key may be used for signing, then it will have the *sign* attribute.

NOTE 1 The signing schemes for RSA are described in B.6 (RSASSA_PKCS1v1_5) and B.7 (RSASSA_PSS). The signing schemes for ECC are described in C.4 (EC Signing).

NOTE 2 Symmetric signing (HMAC and SMAC) may only be performed with unrestricted signing keys.

A key with a sign attribute may also have a restriction on the contents of the message that can be signed with the key. When a key has this restriction, the TPM will not use the key to sign message digests that the TPM did not compute.

Any attestation message produced by a TPM will have a header (TPM_GENERATED_VALUE) to identify the data as being produced within a TPM. If a restricted key is used to sign this data, then a relying party can have assurance that the message data came from a TPM.

To allow a restricted key to sign an externally generated message, the TPM is used to produce the message digest. When the TPM computes the digest, it will validate that the message does not begin with TPM_GENERATED_VALUE. If it does, then the TPM will not produce the special certification (a ticket) that indicates that the digest was produced by the TPM and is safe to sign with a restricted key.

A key designated as a signing key may be used in any command that uses a signing key. For some commands, the signing scheme may be specified in the command. Not all schemes are valid for all keys, and the TPM generates an error if the scheme is not allowed with the indicated key type.

EXAMPLE 1 The RSASSA-PKCS1-v1_5 signing scheme is not valid with an ECC key.

EXAMPLE 2 A key that has the "restricted" attribute may only be used with one signing scheme. If it is limited to be used with RSASSA-PSS, it may not be used with RSASSA-PKCS1-v1_5.

A restricted signing key is required to have a signing scheme specified in the key definition and that is the only signing scheme that is allowed to be used with the key. For an unrestricted key, the key definition may contain a signing scheme selection, or the signing scheme may be determined when the key is used. To defer the signing scheme selection, the key would be created with TPM_ALG_NULL as the signing scheme selection.

11.4.6.2 Signature Verification

TPM2_VerifySignature() validates a signature over a digest. The command takes a handle of a public key, a digest, and a block that contains the signature over the digest.

The TPM validates that the signature scheme is compatible with the selected key. In general, the TPM will be able to validate any signature over a digest that it could have produced.

If the signature is valid, the TPM will produce a ticket.

11.4.6.3 Tickets

A ticket is an HMAC signature that uses a proof value as the HMAC key.

 Copyright © TCG 2006-2020

NOTE Hierarchy proof values are described in detail in14.4.

The TPM uses tickets for two purposes:

- re-signing data. After checking an asymmetric signature, the TPM re-signs the digest using a TPM symmetric key. The TPM can later re-verify a signature without having to load the asymmetric key; and
- expanding state memory. When hashing an external message, the TPM has some state that indicates the message did not start with TPM_GENERATED_VALUE. This state information cannot be retained indefinitely in the TPM. A ticket allows this state to be stored off of the TPM in a way that is easy for the TPM to validate. When a digest is later presented to the TPM to be signed, the ticket is provided allowing the TPM to validate that the digest to be signed is safe to sign.

The proof value used for a ticket will minimally have a number of bits equal to the size of the digest produced by the hash algorithm.

EXAMPLE A proof value of 256 bits is required for a SHA256 ticket.

There are five different ticket types:

1) TPMT_TK_CREATION – this ticket type is produced when an object is created (TPM2_Create() or TPM2_CreatePrimary()). The ticket is used in TPM2_CertifyCreation() so that the TPM can certify that it created a specific object and the environmental parameters (PCR) that were extant when the object was created. This avoids having the digest of the creation data be a permanent part of an object's data structure.
2) TPMT_TK_VERIFIED – this ticket type is produced by TPM2_VerifySignature() and used by TPM2_PolicyAuthorize(). If a signature is signed by an asymmetric key, the signature verification might be time consuming. If the same authorization is going to be used many times (such as an authorization for TPM2_PolicyAuthorize()), there is a performance advantage to having the asymmetric authorization converted so that it uses symmetric cryptography which is usually faster. This ticket is the symmetric equivalent authorization.
3) TPMT_TK_AUTH – this ticket is produced by TPM2_PolicySigned() or TPM2_PolicySecret() and used in TPM2_PolicyTicket(). A policy authorization can be tied to a specific policy session or allowed to be used in any policy. When it can be used in any policy, it has a time at which it expires (which can be some arbitrary time in the future). The long-lived authorization may be given in TPM2_PolicySigned()/TPM2_PolicySecret() and a ticket is produced that is used to verify the authorization parameters (what was authorized) and the time in the future when the authorization expires. This ticket is then processed by TPM2_PolicyTicket() and, until the ticket expires, will have the same effect on the *policyDigest* computation as the original authorization.

NOTE If produced by TPM2_PolicySigned(), the ticket will use the TPM_ST_AUTH_SIGNED structure tag and if produced by TPM2_PolicySecret(), the ticket will use the TPM_ST_AUTH_SECRET structure tag. TPM2_PolicyTicket() will use this tag to indicate which command code to use (TPM_CC_PolicySigned/TPM_CC_PolicySecret) when extending *policyDigest*.

4) TPMT_TK_HASHCHECK – This ticket is used to indicate that a digest of external data is safe to sign using a restricted signing key. A restricted signing key may only sign a digest that was produced by the TPM. If the digest was produced from externally provided data, there needs to be an indication that the data did not start with the same first octets as are used for data that is generated within the TPM. This prevents "forgeries" of attestation data. This ticket is used to provide the evidence that the data used in the digest was checked by the TPM and is safe to sign. Assuming that the external data is "safe", this type of ticket is produced by TPM2_Hash() or TPM2_SequenceComplete() and used by TPM2_Sign().

 Copyright © TCG 2006-2020

5) NULL Ticket – A NULL Ticket is produced when a response has a ticket, but no ticket is produced. An example is TPM2_PolicySecret() with an expiration time of zero. It does not produce a ticket because, since the expiration time was zero, the authorization expires immediately. In this case, the TPM will return a NULL Ticket. A NULL Ticket may also be used as an input parameter when the command requires a ticket, but no ticket data is available.

11.4.7 Symmetric Encryption

The TPM uses symmetric encryption to encrypt some command parameters (typically, authentication information) and to encrypt Protected Objects stored outside it. Cipher Feedback mode (CFB) is the only block cipher mode required by this specification.

Any symmetric block cipher supported by a TPM may be used for parameter encryption. However weak keys are not permitted to be used. Additionally, a TPM should support XOR obfuscation, which is a hash-based stream cipher. XOR obfuscation may be used only for confidential parameter passing.

NOTE XOR allows an application to have confidential and integrity-protected interactions with only one algorithm in common with the TPM (a hash).

When paired with an asymmetric key — as in an ECC decrypting key — a symmetric key is required to have as many bits of security strength as the asymmetric key with which it is paired.

EXAMPLE 1 SP800-57 classifies 2048-bit RSA as providing 112 bits of security. AES with 128- or 256-bit keys provides adequate symmetric security for pairing with a 2048-bit RSA key.

EXAMPLE 2 A prime-modulus ECC key has a security strength that is half the size of the prime modulus. AES with 128- or 256-bit keys is suitable for pairing with a 256-bit ECC key, but AES with 128-bit keys is not recommended for pairing with a 384-bit ECC key.

When a symmetric key is used for data encryption, the encrypted data has an HMAC. This HMAC is checked before the data is decrypted. Verification that the decrypted data is properly associated with the symmetric key is intended to make it more difficult to perform power analysis. To defeat the protections, it would be necessary to defeat two different families of protection rather than one as would exist if the integrity protection were applied to the clear text rather than the cipher text.

11.4.7.1 Block Cipher Modes

The block cipher modes referenced in this specification are defined in ISO/IEC 10116:2006. That specification allows parameterization of most of the modes. In a TPM implementation, the parameters are fixed, as defined in Table 1.

Table 1 — Block Cipher Parameters

Mode	Common Name	Parameter	Comments
CTR	Counter	j = n	size of the plaintext variable
OFB	Output Feedback	j = n	size of the plaintext variable
CBC	Cipher-block Chaining	m = 1	interleave factor
CFB	Cipher-feedback	r = n	size of feedback buffer
		k = n	size of feedback variable
		j = n	size of plaintext variable
ECB	Electronic Code Book	none	
NOTE *n* is the input block size of the cipher.			

 Copyright © TCG 2006-2020

11.4.7.2 Cipher Feedback (CFB) Mode

CFB is used when a symmetric block cipher is chosen as the encryption algorithm associated with a session. When used for parameter encryption, the key and Initialization Vector (IV) are derived from a per-session key so that reuse of the same key and IV is statistically unlikely.

NOTE ISO/IEC 10116 use the term Start Variable instead of Initialization Vector (IV).

CFB is also used for symmetric encryption of the sensitive area of an object when the object is not stored in a Shielded Location. When used in this way, the key and IV are derived from a secret. In some cases, the IV is set to zero.

11.4.7.3 XOR Obfuscation

XOR obfuscation resembles Counter mode (CTR) block encryption, but it uses a KDF as the pseudo-random function instead of a symmetric block cipher.

XOR obfuscation reduces to one (a hash) the number of algorithms that a caller needs in common with the TPM in order to use the TPM with some level of confidentiality and authentication.

This specification's XOR scheme differs from that used in TPM 1.2: it uses a different formulation for input into the hash function.

When this specification calls for use of XOR obfuscation, it uses a function reference. The function prototype is:

$$\mathbf{XOR}(data, hashAlg, key, contextU, contextV) \quad (3)$$

where

data	a variable-sized buffer containing the data to be obfuscated
hashAlg	the hash algorithm to be used in the KDF
key	a variable-sized value containing a secret key
contextU	a variable-sized value used to qualify one of the parties to the operation (often a nonce value)
contextV	a variable-sized value used to qualify one of the parties to the operation (often a nonce value)

The **XOR**() function uses the *hash, key, contextU*, and *contextV* parameters in a call to **KDFa**() to produce a *mask* value:

$$mask \coloneqq \mathbf{KDFa}\ (hashAlg, key, \text{“XOR”}, contextU, contextV, data.size \bullet 8) \quad (4)$$

NOTE The “XOR” value is defined in 4.8.

The octets of *mask* are then XOR'd with the octets of *data.buffer*.

11.4.8 Extend

The Extend operation is used to make incremental updates to a digest value. It is useful for updating PCR, auditing, and constructing policy. Extend uses a hash function to combine new data with an existing digest. Its notation is:

 Copyright © TCG 2006-2020

$$digest_{new} \coloneqq \mathbf{H}_{hashAlg}(digest_{old} \,||\, data_{new}) \quad (5)$$

where

$digest_{new}$	the value of the digest (such as, a PCR) after the Extend operation
$\mathbf{H}_{hashAlg}$	the hash function using a context-specific algorithm (such as, the hash algorithm associated with a specific bank of PCR)
$digest_{old}$	the value of the digest before the Extend operation
$data_{new}$	a variable number of octets of data that are to be hashed with the initial value of $digest_{old}$ to produce Extend results

The Extend operation may also apply to an NV Index that has the TPMA_NV_EXTEND attribute.

11.4.9 Key Generation

Key generation produces two different types of keys. The first, an ordinary key, is produced using the random number generator (RNG) to seed the computation. The result of the computation is a secret key value kept in a Shielded Location.

The second type, a Primary Key, is derived from a seed value, not the RNG directly. The RNG usually generates the seed that is persistently stored on the TPM. Generation of a Primary Key from a seed is based on use of an approved key derivation function (KDF). The KDF from SP800-108 is widely used in this specification.

This specification places no upper limit on the time allowed to generate a key. Platform-specific specifications may limit the time for generating various key types.

Depending on the application, the TPM may generate a key by

- using bits from the RNG, or
- deriving the key from another secret value.

There are many ways to generate keys; these methods are described in detail in each clause where generation of a key is required.

11.4.10 Key Derivation Function

11.4.10.1 Introduction

The TPM uses a hash-based function to generate keys for multiple purposes. This specification uses two different schemes: one for ECDH and one for all other uses of a KDF.

The ECDH KDF is from SP800-56A. The Counter mode KDF, from SP800-108, uses HMAC as the pseudo-random function (PRF). It is referred to in the specification as **KDFa**().

11.4.10.2 KDFa()

With the exception of ECDH, **KDFa**() is used in all cases where a KDF is required. **KDFa**() uses Counter mode from SP800-108, with HMAC as the PRF.

As defined in SP800-108, the inner loop for building the key stream is:

$$K(i) \coloneqq \mathbf{HMAC}\,(K_I, [i]_2 \,||\, Label \,||\, 00_{16} \,||\, Context \,||\, [L]_2) \tag{6}$$

where

$K(i)$	the ith iteration of the KDF inner loop
HMAC()	the HMAC algorithm using an approved hash algorithm
K_I	the secret key material
$[i]_2$	a 32-bit counter that starts at 1 and increments on each iteration
$Label$	a octet stream indicating the use of the key produced by this KDF
00_{16}	Added only if *Label* is not present or if the last octet of *Label* is not zero.
$Context$	a binary string containing information relating to the derived keying material
$[L]_2$	a 32-bit value indicating the number of bits to be returned from the KDF

NOTE 1 Equation (6) is not **KDFa**(). **KDFa**() is the function call defined below.

As shown in equation (6), there is an octet of zero that separates *Label* from *Context*. In SP800-108, *Label* is a sequence of octets that may or may not have a final octet that is zero. If *Label* is not present, a zero octet is added. If *Label* is present and the last octet is not zero, a zero octet is added.

After each iteration, the HMAC digest data is concatenated to the previously produced value until the size of the concatenated string is at least as large as the requested value. The string is then truncated to the desired size (which causes the loss of some of the most recently added bits), and the value is returned.

When this specification calls for use of this KDF, it uses a function reference to **KDFa**(). The function prototype is:

$$\mathbf{KDFa}\,(hashAlg, key, label, contextU, contextV, bits) \tag{7}$$

where

hashAlg	a TPM_ALG_ID to be used in the HMAC in the KDF
key	a variable-sized value used as K_I
label	a variable-sized octet stream used as Label
contextU	a variable-sized value concatenated with *contextV* to create the *Context* parameter used in equation (6) above
contextV	a variable-sized value concatenated to *contextU* to create the *Context* parameter used in equation (6) above
bits	a 32-bit value used as $[L]_2$; and is the number of bits returned by the function

The values of *contextU* and *contextV* are passed as sized buffers and only the buffer data is used to construct the *Context* parameter used in equation (6) above. The size fields of *contextU* and *contextV* are not included in the computation. That is:

$$Context \coloneqq contextU.buffer \,||\, contextV.buffer \tag{8}$$

 Copyright © TCG 2006-2020

The 32-bit value of $bits$ is in TPM canonical form, with the least significant bits of the value in the highest numbered octet.

The implied return from this function is a sequence of octets with a length equal to (*bits* + 7) / 8. If *bits* is not an even multiple of 8, then the returned value occupies the least significant bits of the returned octet array, and the additional, high-order bits in the 0th octet are CLEAR. **The unused bits of the most significant octet (MSO) are masked off and not shifted.**

EXAMPLE If **KDFa**() were used to produce a 521-bit ECC private key, the returned value would occupy 66 octets, with the upper 7 bits of the octet at offset zero set to 0.

11.4.10.3 KDFe for ECDH

Producing a symmetric encryption key for an ECC-protected object uses “One-Pass Diffie-Hellman, C(1, 1, ECC CDH)” from SP800-56A, 6.2.2.2. The KDF used is the “Concatenation Key Derivation Function (Approved Alternative 1)”. The inner loop of that KDF uses:

$$digest_i \coloneqq \mathbf{H}(counter \mid\mid Z \mid\mid OtherInfo) \tag{9}$$

where

$digest_i$	the digest generated on the ith iteration of the loop (i starts at 1)
$\mathbf{H}()$	an approved hash function
$counter$	a 32-bit counter that is initialized to 1 and incremented on each iteration
Z	the X coordinate of the product of a public ECC key and a different private ECC key
$OtherInfo$	a collection of qualifying data for the KDF defined below

The 32-bit $counter$ value is included in TPM canonical form, with the least-significant bit of the counter in the highest numbered octet.

After each iteration, $digest_i$ is concatenated to the previously produced digests (MSO of $digest_i$ follows the LSO of $digest_{i-1}$). The number of iterations is determined by the number of bits to be produced and the size of the digest produced by the hash function. In the returned octet string, the MSO of the returned value is the MSO of $digest_1$.

In SP800-56A, *OtherInfo* is specified as:

$$OtherInfo \coloneqq AlgorithmID \mid\mid PartyUInfo \mid\mid PartyVInfo \;\{\mid\mid SuppPubInfo\}\; \{\mid\mid SuppPrivInfo\} \tag{10}$$

where

$AlgorithmID$	a bit string that indicates how the derived keying material will be parsed and for which algorithm(s)
$PartyUInfo$	public information contributed by party U (the initiator)
$PartyVInfo$	public information contributed by party V (the responder)
$SuppPubInfo$	public information known to both U and V (optional)
$SuppPrivInfo$	private (secret) information known to both U and V (optional)

This specification requires that $OtherInfo$ be constructed as:

 Copyright © TCG 2006-2020

$$OtherInfo \coloneqq Use \,||\, PartyUInfo.buffer \,||\, PartyVInfo.buffer \tag{11}$$

where

Use	a null-terminated string indicating the use of the key (e.g., "DUPLICATE", "IDENTITY", "SECRET", etc.) (see clause 4 for the definition of these values). This field satisfies the requirements of SP800-56A since the parsing of keying material is determined by the use.
PartyUInfo.buffer	the x-coordinate of the public point of an ephemeral key
PartyVInfo.buffer	the x-coordinate of the public point of a static TPM key

The x-coordinates of the public points are *sized buffers* (that is, integers indicating the size in octets of the buffer that follows). The buffer data is used in the KDF, but the size field is not.

When this specification calls for use of this KDF, it uses a function reference to **KDFe**(). The function prototype is:

$$\mathbf{KDFe}(hashAlg, Z, Use, PartyUInfo, PartyVInfo, bits) \tag{12}$$

where

hashAlg	the hash algorithm to be used as **H**() in equation (9) above
Z	the product of a public point and a private x-coordinate
Use	a null-terminated string indicating the use of the key (e.g., "DUPLICATE", "IDENTITY", "SECRET", etc.) (see clause 4 for the definition of these values).
PartyUInfo	the x-coordinate of the public point of an ephemeral key
PartyVInfo	the x-coordinate of the public point of a static TPM key
bits	a 32-bit value indicating the number of bits to be returned

The implied return from this function is an octet string containing $bits/8$ octets. If *bits* is not an even multiple of 8, the return value is the least-significant bits of the return value, and the additional high-order bits in the 0th octet are CLEAR. The unused bits of the MSO are masked off and not shifted.

NOTE The function prototype in (12) is not a C-language prototype but, rather, a prototype to illustrate the parameters of the KDF for specific uses. The C-language prototype will include an extra parameter that will be the buffer to receive the key material generated by the KDF.

11.4.10.4 Rejection of weak keys

Some algorithms have known weak keys. If such a key is generated, it must be discarded, and a new key generated by starting over with another iteration of the KDF. In the case of DES, there are 64 known weak or semi-weak keys. None of them are allowed. In the case of AES, at least one bit in the upper half of the key must be set. Again, if this is not true, the key must be discarded, and a new key generated by starting over with another iteration of the KDF.

11.4.11 Random Number Generator (RNG) Module

11.4.11.1 Source of Randomness

The RNG is the source of randomness in the TPM. The TPM uses random values for nonces, in key generation, and for randomness in signatures.

The RNG is a Protected Capability with no access control. It nominally consists of

- an entropy source and collector,
- a state register, and
- a mixing function (typically, an approved hash function).

The entropy collector collects entropy from entropy sources and removes bias. The collected entropy is then used to update the state register providing input to the mixing function to produce the random numbers.

The mixing function may be implemented with a pseudo-random number generator (a PRNG). A PRNG may produce numbers that are apparently random from a non-random input (such as, a counter). Combining an approved PRNG with an input that has considerably more entropy than a counter yields an RNG with properties no worse than the underlying PRNG and possibly much better.

The RNG should meet the certification requirements of the intended market.

The TPM should provide sufficient randomness for each use by an internal function. When accessed by an external call, it should be able to provide 32 octets of randomness. Larger requests may fail if insufficient randomness is available.

Each RNG access produces a new value regardless of the data's use. There is no distinction between accesses for internal versus external purposes.

11.4.11.2 Entropy Source and Collector

A TPM should have at least one internal source of entropy, and possibly more. These sources could include noise, clock variations, air movement, and other types of events.

As noted, the entropy collector is the process that collects the entropy from various sources and removes bias.

EXAMPLE If the entropy source has a bias of creating 60 percent 1s and only 40 percent 0s, then the collector design corrects the bias before sending the information to the state register.

The entropy source and collector should provide entropy to the state register in a manner that is not visible to an outside process or other TPM capability.

The entropy collector should regularly update the state register with additional, unbiased entropy.

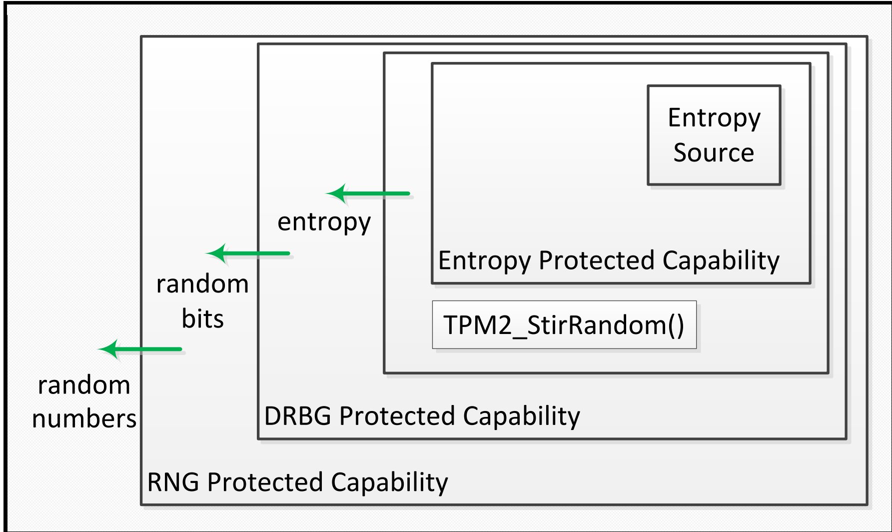


Figure 4 — Random Number Generation

Any Protected Capability that requires an unpredictable number obtains it from a Random Number Generator (RNG) Protected Capability in the same TPM. The RNG Protected Capability assembles random bits from a Deterministic Random Bit Generator (DRBG) Protected Capability in the same TPM. The DRBG Protected Capability obtains entropy from the entropy Protected Capability in the same TPM and the TPM2_StirRandom() Protected Capability can be used to add additional information. The entropy Protected Capability obtains entropy from an entropy source in the same TPM.

NOTE 1 The "additional information" added by TPM2_StirRandom() could be entropy gathered from other sources but the TPM has no way of determining if the value has any entropy or not. As a consequence, it is just deemed to be "additional information."

NOTE 2 The DRBG Protected Capability of a non-FIPS TPM consists of a DRBG mechanism that should comply with NIST Recommendation SP800-90 A, revised March 2012; except it does not comply with its Clause 11.

NOTE 3 The DRBG Protected Capability of a FIPS TPM consists of a DRBG mechanism that complies with NIST Recommendation SP800-90 A, revised March 2012.

The DRBG mechanism security should be at least as strong as the security strength of the strongest cryptographic algorithm implemented in the TPM.

The DRBG Protected Capability should be reseeded using entropy from the entropy Protected Capability when:

- a flag is SET indicating that reseeding is required;
- TPM2_StirRandom() is executed;
- after the TPM has failed a self-test; or
- before the SPS is replaced.

It may be reseeded at other times, as well.

NOTE 4 Each TPM may be seeded during TPM manufacture, via a manufacturer-specific method, using a personalization string for the DRBG that should be specific to the manufacturer and the type of TPM, plus a manufacturer-provided nonce that is specific to the individual TPM.

 Copyright © TCG 2006-2020

11.4.11.3 Nonce Creation

The RNG module provides the bits used in any TPM-generated nonce.

11.4.12 Algorithms

11.4.12.1 Algorithm Identifiers

The structures and commands in this specification are constructed with minimal reliance on algorithm defaults.

In most cases, an algorithm identifier identifies a family of algorithms followed by qualifiers. This differs from the TPM 1.2 version of the specification, which often included the key size in the algorithm identifier (TPM_ALG_AES128). This specification only uses the TPM 1.2 form of algorithm identifiers for hash algorithms.

Since this specification depends on being able to discern the hash output size from the algorithm ID, its hash algorithm identifiers imply a digest size.

EXAMPLE 1 Some of the hash algorithm identifiers are TPM_ALG_SHA256, TPM_ALG_SHA384, and TPM_ALG_SM3_256.

Algorithm identifiers for symmetric and asymmetric encryption identify the family, such as RSA, ECC, AES, etc. For these algorithms, supplementary information is required to define parameters.

EXAMPLE 2 Some family algorithm identifiers are TPM_ALG_ECC, TPM_ALG_RSA, TPM_ALG_SM4, and TPM_ALG_AES.

11.4.12.2 Algorithm Support

This specification does not require implementation of any specific set of algorithms. When determining algorithms or algorithm sets supported, implementers should carefully consider factors such as use cases, strength of function, interoperability, backward compatibility, algorithm diversity, etc. TCG recommends using TCG platform-specific specifications that reflect industry best practices.

NOTE 1 It is anticipated that support for TPM 1.2 compatibility will be retained unless support for the 1.2 algorithms (RSA 2048 and SHA1) would prevent that TPM from being sold.

TCG will specify sets of algorithms to be incorporated by various platform-specific specifications. Each set includes a minimum of one hash algorithm, one symmetric encryption algorithm with approved parameters, and one asymmetric encryption/signing algorithm with approved parameters. Without a complete set of algorithms, the TPM would be unable to support all necessary functions.

A TPM may support algorithms in addition to the required sets. These do not need to be part of any set. For example, the TPM may include an additional hash algorithm without including an additional asymmetric or symmetric algorithm.

It is possible, and very likely given the multitude of algorithms supported by the TPM, that key-size support will differ between TPM implementations. In addition, keys created by outside software may greatly increase the number of key sizes that are possible to load.

A TPM will not create or load an object that uses an algorithm that is not supported by the TPM. When creating an object, the TPM checks the template for the object being created and when loading an object, the TPM checks the public area of the object. In both cases, the TPM validates that it supports all of the indicated algorithms, parameters, and key sizes.

 Copyright © TCG 2006-2020

The strength of at least one algorithm set supported by a TPM should be at least 112 bits. Other algorithms and algorithm sets may be supported in any combination.

NOTE 2 A set's strength is normally determined by the number of bits in a key of the symmetric algorithm. An exception is Suite B, Top Secret, where the strength is considered to be 192 bits even though the symmetric algorithm has 256-bit keys.

If a TPM supports RSA, it should support a key size of 2048 bits or larger. Support for smaller key sizes is allowed but discouraged.

NOTE 3 Support for smaller keys is allowed so that legacy keys may continue to be supported. Use of key-sizes less than 1024 bits is strongly discouraged.

A platform-specific specification may mandate support for algorithms or algorithm sets. It may select only those algorithms for which the TCG has assigned algorithm identifiers.

A TPM may only implement algorithms that have a TCG-assigned algorithm ID.

11.5 Authorization Subsystem

The Authorization Subsystem is called by the Command Dispatch module at the beginning and end of command execution. Before the command may be executed, the Authorization Subsystem checks that proper authorization for use of each of the Shielded Locations has been provided.

Some commands access Shielded Locations that require no authorizations; access to some locations may require a single-factor authorization; and access to other Shielded Locations may require use of an authorization policy of arbitrary complexity.

The only cryptographic functions required by the Authorization Subsystem are hash and HMAC. An asymmetric algorithm may be required if TPM2_PolicySigned() is implemented.

The details of the different methods of authorization are provided in Clause 19.

11.6 Random Access Memory

11.6.1 Introduction

Random access memory (RAM) holds TPM transient data. Data in TPM RAM is allowed, but not required, to be lost when TPM power is removed. Because the values in TPM RAM may be lost, in this specification they are referred to as being volatile, even if the data loss is implementation-dependent.

When the specification refers to a value that has both volatile and non-volatile copies, they may be kept in a single location as long as that location has the properties of allowing random access and having unlimited endurance.

Not all values in TPM RAM are in Shielded Locations. A portion of TPM RAM contains the I/O buffer with properties that are described in 11.3.

11.6.2 Platform Configuration Registers (PCR)

PCR are Shielded Locations used to validate the contents of a log of measurements. The nominal behavior of a trusted platform is to maintain, in a log, a record of the events that affect the security state of the platform, at least through the boot process while it is establishing the TCB. When additions are made to the log, the TPM receives a copy of the log entry or the digest of data described by the log. The

data sent to the TPM is included in an accumulative hash in a PCR. The TPM may then provide an attestation of the value in the PCR, which, in turn, verifies the contents of the log.

It is possible for a single PCR to record all log entries. However, this would make it difficult to evaluate the different stages of platform evolution as it boots into the operating system. Normally, multiple PCR are provided in a TPM to allow simplification of the evaluation.

EXAMPLE 1 A TPM intended for a PC could have a PCR dedicated to recording measurements of the BIOS, a PCR dedicated to the boot ROM on add-in cards, and a PCR dedicated to the OS loader. The platform-specific specifications determine the number of PCR and their attributes in a TPM.

PCR may also be used to gate access to an object. If selected PCR do not have the required values, the TPM will not allow use of the object.

A TPM may maintain multiple banks of PCR. A PCR bank is a collection of PCR that are Extended with the same hash algorithm. PCR banks are identified by the hash algorithm used to Extend the PCR in that bank.

Multiple banks may handle situations where one hash algorithm is required for legacy or compatibility with one set of applications, while a different hash algorithm is required to meet the security needs of another application. Within a bank, all PCR updates use the same hash algorithm. Not all banks need to have the same number of PCR, but the attributes of all PCR with the same Index, other than hash algorithm, are the same in all banks.

EXAMPLE 2 If PCR[0] has an attribute that allows it to be reset by TPM2_PCR_Reset(), then that attribute applies to PCR[0] in all banks.

NOTE 1 Since banks may have different numbers of PCR, a PCR Index value may not be valid for all banks. The allocation of PCR may also be changed by TPM2_PCR_Allocate() using Platform Authorization. Changing the PCR allocation does not change the attributes of the PCR.

The contents of a PCR may be modified or reported. The two ways to modify a PCR are to reset it or Extend it. Reporting on a PCR may be accomplished through simple reading, inclusion in an attestation, or inclusion in a policy.

Although listed in this clause, PCR need not be maintained in RAM. They may be kept in non-volatile memory. If kept in non-volatile memory, consideration must be made for the possible impact on TPM performance during the critical boot phase, when many measurements are recorded.

A TPM is required to implement a PCR bank for each supported algorithm. However, a PCR bank may be defined such that it contains no PCR.

NOTE 2 The requirement that a PCR bank be implemented for each hash algorithm allows the unmarshaling to be based on the implemented algorithms rather than the implemented PCR.

The TPM may support Resume PCR that retain their state across a TPM Resume sequence but are set to their default initial value on TPM Reset or TPM Restart.

11.6.3 Object Store

TPM RAM holds keys and data that are loaded into the TPM from external memory. In most cases, an object may not be used or modified unless it was first loaded into TPM RAM with one of the object load commands: (TPM2_Load(), TPM2_CreatePrimary(), TPM2_LoadExternal(), or TPM2_ContextLoad()).

NOTE TPM2_Create() does not automatically load the object. After creation, the object needs to be explicitly loaded with TPM2_Load(), to load both the public and private portions, or with TPM2_LoadExternal() to load just the public portion.

 Copyright © TCG 2006-2020

The structure used for keys may be generalized for use on data objects if the access properties used for keys are suitable for access to these objects.

EXAMPLE A data blob may be defined so that access requires that some set of PCR has defined values, or an authorization value may be needed for access. Such a data blob, called a Sealed Data Object, is managed in the same way that a key is managed. That is, the Sealed Data Object should be loaded before being accessed, and the loaded blob may be context saved.

The TPM operates on other structures that are passed as parameters in specific commands. These structures are transient and are not stored in the TPM as identifiable entities after the command has completed.

Items loaded in the TPM are given handles to let them be referenced in subsequent commands.

11.6.4 Session Store

The TPM uses sessions to control a sequence of operations. A session may audit actions, provide authorizations for actions, or encrypt parameters passed in commands.

A session may be created as needed using one of the session creation commands. The session is assigned a handle at that time.

A TPM may be designed so that the RAM used for sessions is from a memory pool shared with the object store. It may also be designed so that the session store and object store are separated and dedicated.

11.6.5 Size Requirements

Random access memory (RAM) should be large enough to handle the transient state, sessions, and objects needed for completion of any implemented command. The minimums for the worst-case command in this specification are:

- two loaded entities (two keys, a key and a Sealed Data Object, or a hash/HMAC sequence and a key);
- three authorization sessions;
- an input buffer able to accommodate the largest command or an output buffer required for the largest possible response;

NOTE The largest command or response depends on the algorithms supported by the implementation.

- any vendor-defined state required for operation; and
- all defined PCR.

11.7 Non-Volatile (NV) Memory

The NV memory module stores persistent state associated with the TPM. Some NV memory is available for allocation and use by the platform and entities authorized by the TPM Owner.

TPM NV memory contains Shielded Locations and Shielded Location can only be accessed with Protected Capabilities.

If the specification is not explicit about storage of a parameter, that parameter may be in either RAM or NV, according to vendor preference.

If the NV memory of the TPM is subject to wear, then the TPM should detect whether the data being written to an NV memory location is the same as that currently stored and not perform the NV write if they are the same.

The OS or the platform may define a special NV data structure (an NV Index) in order to store persistent data values. NV memory may also be used persistently to store a loaded object. When a persistent object is referenced in a TPM command, the TPM may move that object into an object slot so it may be accessed more efficiently. The TRM needs to ensure that sufficient object memory RAM is available to allow this movement.

NOTE The movement occurs transparently.

A TPM capability indicates if the TPM is using Transient Object resources when a command references a persistent object. If so, the TRM needs to ensure that a Transient Object slot is available for each persistent object so referenced.

11.8 Power Detection Module

This module manages TPM power states in conjunction with platform power states.

All platform-specific TCG specifications that define the binding of the TPM to the platform should include a requirement that the TPM be notified of all power state changes.

The TPM supports only the ON and OFF power states. Any system power transition requiring the RTM to be reset also causes the TPM to be reset (_TPM_Init). Any system power transition that causes the TPM to be reset will also cause the RTM to be reset.

NOTE In most cases, the RTM will be a host CPU.

 Copyright © TCG 2006-2020

12 TPM Operational States

12.1 Introduction

This clause describes TPM operational states and state transitions.

12.2 Basic TPM Operational States

12.2.1 Power-off State

A hardware TPM is in the Power-off state when reset is being asserted or when no power is applied to the TPM. The TPM may internally generate reset by detecting low power or reset may be provided by an external source.

It is possible to transition to the Power-off state from any other state because power can be lost at any time.

NOTE Uncontrolled transitions to this state are not shown in diagrams/descriptions because they would add unnecessary clutter and provide no additional understanding.

12.2.2 Initialization State

The TPM is placed in its initialization state when it receives the _TPM_Init indication. _TPM_Init is provided in a platform-specific manner. For a hardware TPM, the _TPM_Init is normally signaled by the de-assertion of the TPM's reset signal. It may also be signaled by an interface protocol or setting. For a software implementation, _TPM_Init may be a dedicated procedure call.

Regardless of how it is generated, _TPM_Init should coincide with a reset of the Roots of Trust for Measurement for which the TPM is the Root of Trust for Reporting. For example, if the TPM is a component on the PC's motherboard, _TPM_Init should coincide with a reset of the processor and chipset. After _TPM_Init is indicated, the RTM should begin executing the Core Root of Trust for Measurement. It should not be possible to reset the TPM without resetting the RTM. It should not be possible to reset the RTM without resetting the TPM.

While in the Initialization state, the TPM performs basic initialization functions in preparation for accepting commands on the TPM interface. These functions are implementation dependent but, minimally, the TPM should perform validation of the TPM firmware necessary to execute the expected command. If the TPM is in Field Upgrade mode (FUM), the expected command is TPM2_FieldUpgradeData(). If the TPM is not in FUM, the expected command is TPM2_Startup().

After completing the initializations, the TPM waits for the next command and, if the command is not the expected first command, the TPM will return an error indicative of the mode. If the TPM returns an error, it will continue to wait for the expected first command.

NOTE 1 If the TPM is not in FUM, it returns TPM_RC_INITIALIZE. If the TPM is in FUM, it returns TPM_RC_UPGRADE.

NOTE 2 If TPM2_Startup()/TPM2_FieldUpgrateData() is not the first command to the TPM, it indicates failure of the system to properly enter the CRTM, and the reliability of TPM measurements may not be assured. While it is possible to define a special failure mode that prohibits just PCR-related operations, it is expected to be infrequent enough not to warrant such a mode and, as shown in Figure 5, the TPM does not enter Failure Mode, if the first command is not TPM2_Startup().

When the TPM receives TPM2_Startup(), it becomes operational and is able to process other commands.

 Copyright © TCG 2006-2020

NOTE 3 For compliance with other standards, such as FIPS 140, it is necessary for the TPM to validate the firmware associated with a command's execution before that command is executed. This includes the code associated with TPM2_Startup() and TPM2_FieldUpgadeData(). This validation may require use of a digital signature or message authentication code.

Occasionally, some TPM state may need to be retained over a power transition. This might occur if the platform is entering the Suspend state, where the preponderance of system state is retained. To allow the TPM to reflect this condition, system software may issue TPM2_Shutdown(TPM_SU_STATE) to the TPM.

TPM2_Shutdown() initiates an orderly shutdown of the TPM. The command's *startupType* parameter indicates the type of startup that is anticipated to follow and the type of data to be saved. For TPM2_Shutdown(TPM_SU_CLEAR), the amount of data saved to NV memory is relatively small, with considerably more information retained when TPM_SU_STATE is indicated.

12.2.3 Startup State

12.2.3.1 TPM2_Startup()

TPM2_Startup() transitions the TPM from the Initialization state to an Operational state. The command includes information from the platform to inform the TPM of the platform's operating state. TPM2_Startup() has two options: TPM_SU_CLEAR and TPM_SU_STATE. The operating state of a TPM after TPM2_Startup() is dependent on how the TPM was shut down and the selected startup option.

12.2.3.2 Startup Types

The following terms are used to refer to the different startup and shutdown operations:

- Startup(CLEAR) means TPM2_Startup(*startupType* == TPM_SU_CLEAR);
- Startup(STATE) means TPM2_Startup(*startupType* == TPM_SU_STATE);
- Shutdown(STATE) means TPM2_Shutdown(*startupType* == TPM_SU_STATE); and
- Shutdown(CLEAR) means TPM2_Shutdown(*startupType* == TPM_SU_CLEAR).

The combinations of Shutdown() and Startup() provide three unique methods of preparing the TPM for operation:

1) **TPM Reset** is a Startup(CLEAR) that follows a Shutdown(CLEAR), or a Startup(CLEAR) for which there was no preceding Shutdown() (that is, a disorderly shutdown). A TPM Reset is roughly analogous to a reboot of a platform. As with a reboot, most values are placed in a default initial state, but persistent values are retained. Any value that is not required by this specification to be kept in NV memory is reinitialized. In some cases, this means that values are cleared, in others it means that new random values are selected.
2) **TPM Restart** is a Startup(CLEAR) that follows a Shutdown(STATE). This indicates a system that is restoring the OS from non-volatile storage, sometimes called "hibernation". For a TPM Restart, the TPM restores values saved by the preceding Shutdown(STATE) except that all the PCR are set to their default initial state. This allows the TPM to record the boot sequence to ensure that the TCB is properly instantiated while allowing continued function of the restored OS.
3) **TPM Resume** is a Startup(STATE) that follows a Shutdown(STATE). This indicates a system that is restarting the OS from RAM memory, sometimes called "sleep." For sleep, the expectation is that the CRTM will perform the minimal actions required to make the system functional and then "return" to the running OS rather than rebooting it. TPM Resume restores all of the state that was saved by Shutdown(STATE), including those PCR that are designated as being preserved by Startup(STATE). PCR not designated as being preserved, are reset to their default initial state.

 Copyright © TCG 2006-2020

NOTE 1 The PCR are designated in a platform-specific specification.

If the TPM receives Startup(STATE) that was not preceded by Shutdown(STATE), then there is no state to restore and the TPM will return TPM_RC_VALUE. The CRTM is expected to take corrective action to prevent malicious software from manipulating the PCR values such that they would misrepresent the state of the platform. The CRTM would abort the Startup(State) and restart with Startup(CLEAR).

NOTE 2 The startup behavior defined by this specification is different than TPM 1.2 with respect to Startup(STATE). A TPM 1.2 device will enter Failure Mode if no state is available when the TPM receives Startup(STATE). This is not the case in this specification. It is up to the CRTM to take corrective action if it the TPM returns TPM_RC_VALUE in response to Startup(STATE).

The TPM is required to validate the integrity of any NV values before those values are used before that state is used. This includes the state saved by TPM2_Shutdown(STATE)(see 12.2.4). When the TPM determines that some NV value required for proper TPM operation is not valid, the TPM will enter Failure Mode.

It is not specified when the validation of state specific to TPM Resume is to be checked. This gives implementation options that may be specified by a platform-specific specification or determined by the vendor.

The startup sequences are illustrated in Figure 5.

 Copyright © TCG 2006-2020

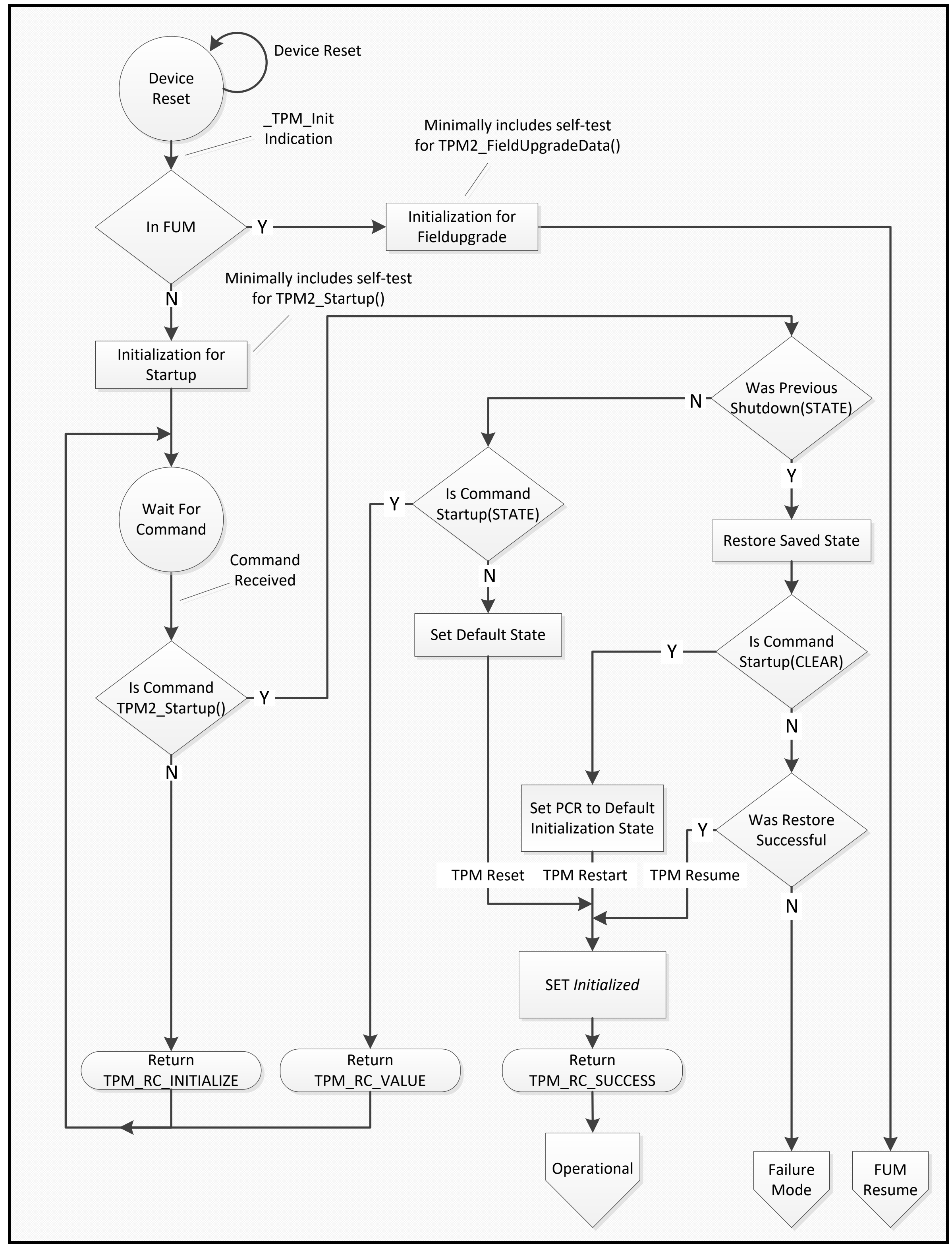


Figure 5 — TPM Startup Sequences

12.2.4 Shutdown State

TPM2_Shutdown() is used to prepare the TPM for loss of power. As with TPM2_Startup(), TPM2_Shutdown() has two options: TPM_SU_CLEAR and TPM_SU_STATE.

 Copyright © TCG 2006-2020

TPM2_Shutdown(TPM_SU_STATE) preserves the majority of the TPM operational state so that it may be restored on a subsequent TPM2_Startup(). TPM2_Shutdown(TPM_SU_CLEAR) preserves a minimal amount of state, mostly to ensure continuity of the TPM timing functions.

NOTE The timing functions are described in Clause 36.

The TPM preserves state data in NV memory. Data is copied from RAM into NV memory so that it is not lost when power is removed from the TPM. The amount of data copied to NV memory is largely implementation-dependent, but the specification indicates the state data that is required to be preserved. This state data is recovered in a subsequent TPM2_Startup(). The type of startup determines what parts of the saved state data is restored and what is discarded.

A shutdown is "orderly" if the TPM receives TPM2_Shutdown() before power is lost and if the state is not subsequently modified by a TPM command before the next _TPM_Init.

These commands will invalidate saved TPM state:

NOTE This is not an inclusive list:

- TPM2_Clear(), TPM2_ChangeEPS(), TPM2_ChangePPS() – these commands invalidate saved contexts in the hierarchy. TPM2_Clear() invalidates preserved contexts in both the storage and endorsement hierarchies.
- TPM2_ContextSave() – context variables are modified by context save. Saving a session context changes the session context ID and its tracking state (saved or in memory). Saving an object context changes the object context ID.
- TPM2_ContextLoad() for a session – the context ID and tracking state (in TPM or context saved) for each active session should be retained across a TPM Restart or TPM Resume sequence. Saving or loading a session context changes the context ID or its tracking state. Saving or loading an object context need not invalidate a preserved context.
- Any command that modifies a PCR – regardless of the implementation, any change to a Resume PCR will invalidate the saved state. If the TPM implements TPM2_PolicyPCR() and uses a PCR generation counter, any PCR modification will change this counter value.

EXAMPLE If a Shutdown(STATE) occurs but, prior to Startup(STATE), a TPM2_PCR_Event() is executed selecting a Resume PCR, then the preserved state is no longer valid, and Startup(STATE) is not valid until another Shutdown(STATE) occurs.

- Any command that modifies *Clock* or returns the value of *Clock*.

A TPM implementation may invalidate a preserved context on any command except TPM2_GetCapability().

12.2.5 Startup Alternatives

The description of the startup process above was given in terms of a command interface. In some systems, the TPM code is run in a special processor mode that provides the required isolation between the TPM state and any other program state. For these implementations, TPM2_Startup() may not be a command that is actually implemented. That is, the platform initialization may boot, validate the TPM code, and place the TPM in a state that is functionally equivalent to having run TPM2_Startup() on a discrete TPM component.

12.3 Self-Test Modes

If a command requires use of an untested algorithm or functional module, the TPM performs the test and then completes the command actions. When performing a self-test on demand, the TPM should test only those algorithms needed to complete the command (see Figure 6).

NOTE 1 It is preferable for the TPM to perform self-tests on untested algorithms and functional blocks as a background task to increase the likelihood that algorithms are tested before they are needed.

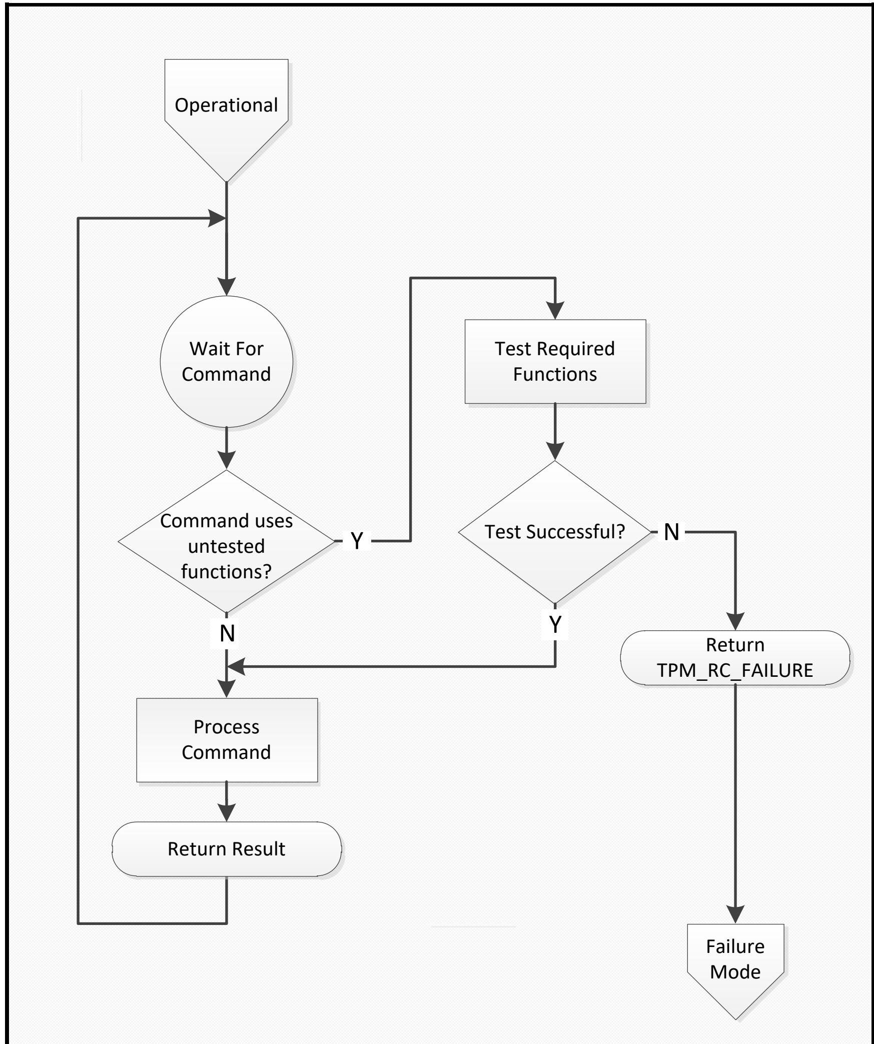


Figure 6 — On-Demand Self-Test

After sending TPM2_Startup(), the system may use either TPM2_SelfTest() or TPM2_IncrementalSelfTest() to cause the TPM to perform tests of untested algorithms. TPM2_SelfTest() may optionally cause the TPM to perform a full self-test of all algorithms and functional blocks. Once these commands are issued, the TPM returns TPM_RC_TESTING for any command that requires use of any testable function until all requested tests are completed.

NOTE 2 FIPS 140-2 requires that all power-on self-tests be complete before the TPM returns any value that depends on the results of a testable function. If compliance with FIPS 140-2 is required, any command that requires use of an untested function causes the TPM to operate as if TPM2_SelfTest(fullTest = NO) was received. The TPM returns TPM_RC_TESTING and continues to return TPM_RC_TESTING until all tests are complete. Alternatively, it may complete all tests and then complete the command. It may also return TPM_RC_NEEDS_TEST.

NOTE 3 Authenticated tests may be generated by attaching an audit session to TPM2_GetTestResult() and then using TPM2_GetSessionAuditDigest() to obtain the signature.

If any self-tests fail, the TPM goes into Failure mode and does not allow execution of any Protected Capabilities except TPM2_GetTestResult() and TPM2_GetCapability(). The TPM exits Failure mode when it receives _TPM_Init.

12.4 Failure Mode

If the TPM fails an internal test, it enters Failure mode. While in Failure mode, the TPM returns TPM_RC_FAILURE in response to any command except TPM2_GetTestResult() or TPM2_GetCapability() (see Figure 7). While in Failure mode, the TPM is only required to provide a limited number of property values. They are all in the set of TPM properties (TPM_CAP_TPM_PROPERTIES):

- TPM_PT_MANUFACTURER
- TPM_PT_VENDOR_STRING_1
- TPM_PT_VENDOR_STRING_2
- TPM_PT_VENDOR_STRING_3
- TPM_PT_VENDOR_STRING_4
- TPM_PT_VENDOR_TPM_TYPE
- TPM_PT_FIRMWARE_VERSION_1
- TPM_PT_FIRMWARE_VERSION_2

NOTE An implementation is allowed to return other property values.

 Copyright © TCG 2006-2020

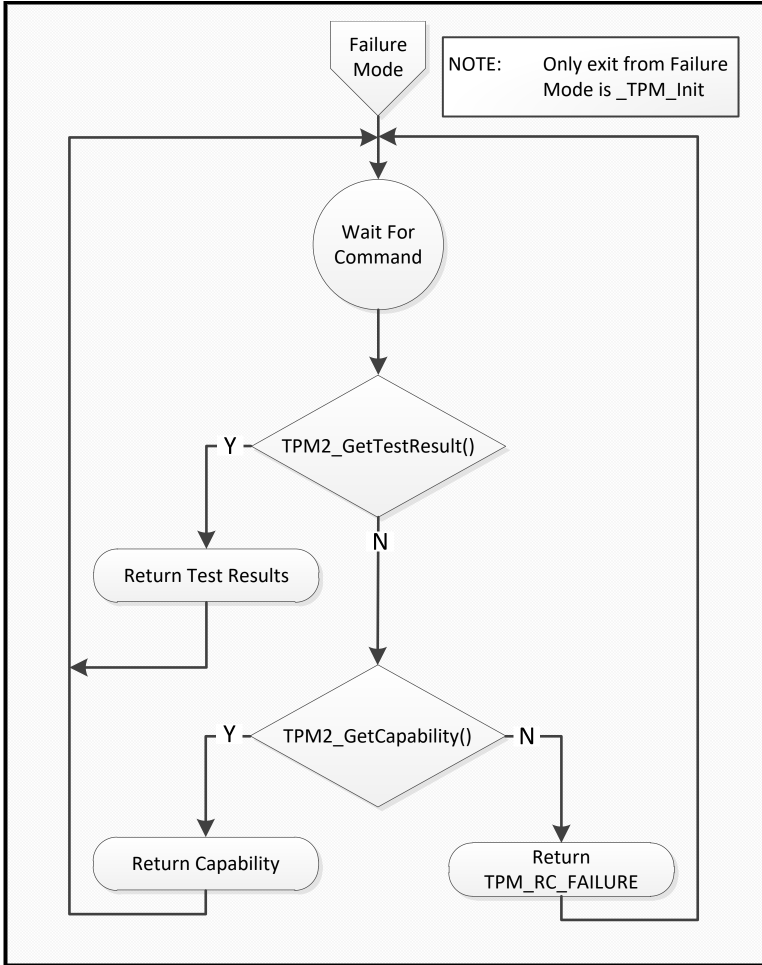


Figure 7 — Failure Mode Behavior

12.5 Field Upgrade

12.5.1 Introduction

This specification describes optional Protected Capabilities for upgrading the TPM firmware. The methods described in this specification would allow the upgrade process to be handled in a standard way on TPMs from multiple vendors. The methods described here should not be viewed as limiting the vendor's options for implementation of their own, vendor-specific methods for upgrading the TPM firmware. However, the field upgrade methods chosen by the vendor should not be less robust than the methods described in this specification. In particular, the authorizations for the upgrade should be the same as the field upgrade commands in this specification.

12.5.2 Field Upgrade Mode

This specification describes two optional upgrade methods: full and incremental. These terms do not refer to how much of the firmware in the TPM changes, but to how the upgrade is applied.

- For a full upgrade, the TPM stores in Shielded Locations all blocks of the firmware update. It makes no change to the executing firmware unless all the blocks are confirmed to be correct. The upgrade process may be interrupted or abandoned without affecting TPM functionality.

 Copyright © TCG 2006-2020

- For an incremental upgrade, firmware updates may be applied as each block is received. The TPM may not be fully functional if the upgrade process is abandoned.

The field upgrade process starts when the TPM receives a properly authorized TPM2_FieldUpgradeStart() (see Figure 6). That command contains the digest of a first block of the upgrade. If the next command is TPM2_FieldUpgradeData() and the digest of the data parameter (*fuData*) of the command matches the signed digest in TPM2_FieldUgradeStart(), the TPM accepts *fuData* as containing the upgrade data.

The TPM may buffer firmware update blocks and not change the firmware until its buffer is full. When a consequential change to the running firmware is made, the TPM enters Field Upgrade mode (FUM) and does not accept any command but TPM2_FieldUpgradeData() until the update is complete (see Figure 7). Before the TPM enters FUM

- it may accept other commands, and
- the update sequence may be abandoned by sending a zero-length upgrade data buffer. The TPM acknowledges that it has abandoned the field upgrade by returning TPM_ALG_NULL for *nextDigest*.

When the field upgrade process is complete, the TPM may either return to normal operation or enter a mode that requires _TPM_Init before normal TPM operations resume. The TPM vendor should determine if a reboot is required after the firmware update and cause the TPM to set the mode appropriately.

If the TPM is reset (_TPM_Init) while in FUM and the TPM is not able to revert to normal operation, three possibilities exist for recovery. The choice is determined by the digest of the first upgrade block provided to the TPM after _TPM_Init. The TPM may retain up to three digest values that it uses for comparison:

1) the digest of the first upgrade block of the current sequence to be used when the intent is to restart the current upgrade sequence from the start (called Digest C in Figure 8);
2) the digest of the first block of the firmware that was being replaced (called Digest P in Figure 8) to be used when the intent is to abort the upgrade and restore the previous firmware; and
3) the digest of the first upgrade block of the factory installed firmware (called Digest F in Figure 8) to restore the TPM to its factory state.

To enable option 2) above, the TPM may support TPM2_FirmwareRead() so that the software performing the upgrade can save a copy of the current TPM firmware in case the upgrade fails.

NOTE TPM2_FirmwareRead() may not be supported on a TPM even if the TPM can perform a field upgrade.

If _TPM_Init is received while the TPM is in FUM, then TPM Reset is required after the field upgrade completes, regardless of the nature of the firmware changes. This reset is required because the TPM does not accept TPM2_Startup() while in FUM, and the TPM will not reflect the state of the platform.

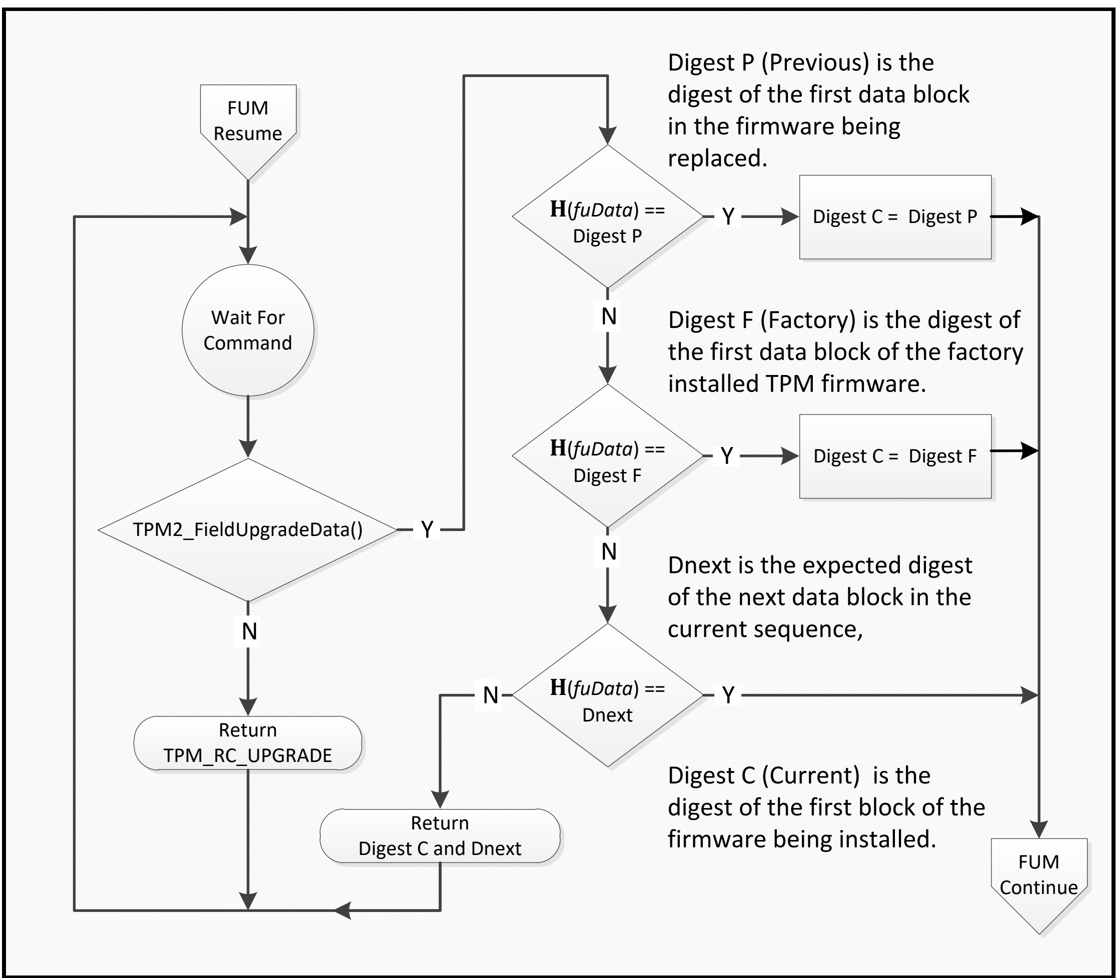


Figure 8 — Resuming Field Upgrade Mode after _TPM_Init

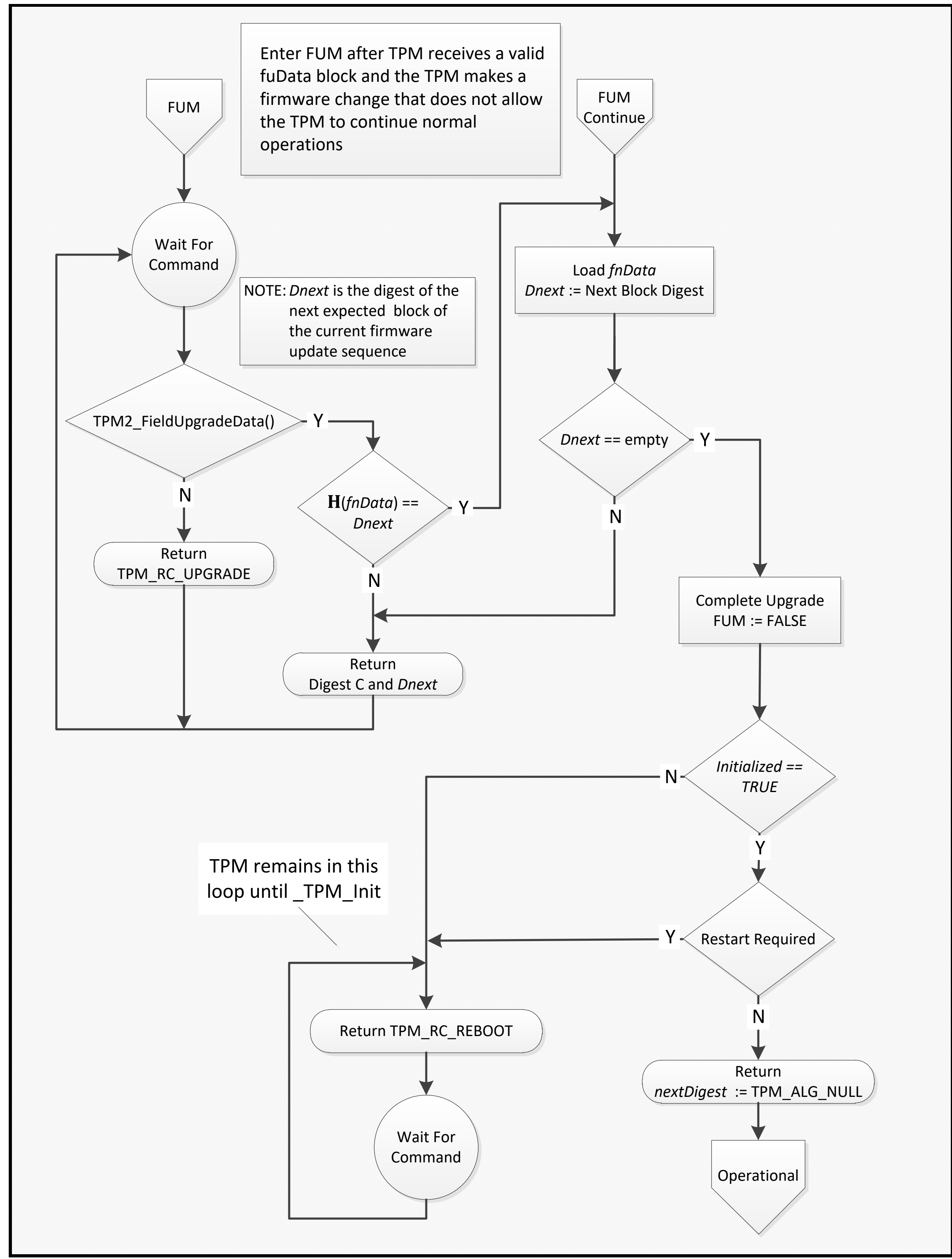


Figure 9 — Field Upgrade Mode

12.5.3 Preserved TPM State

A field upgrade may not cause exposure of any data that is specific to a TPM instance. This includes:

 Copyright © TCG 2006-2020

- Primary Seeds;
- Hierarchy *authValue*, *authPolicy*, and *proof* values;
- *lockoutAuth* and authorization failure count values;
- PCR authValue and authPolicy values;
- NV Index allocations and contents;
- Persistent object allocations and contents; and
- Clock.

In particular, if the TPM supports TPM2_FirmwareRead(), the returned data is not allowed to contain any data that is unique to the TPM instance.

A field upgrade should not cause the loss of any data that is specific to a TPM instance.

NOTE 1 A platform manufacturer may provide a means to change preserved data to accommodate a case where a field upgrade fixes a flaw that might have compromised TPM secrets.

12.5.4 Field Upgrade Implementation Options

The method described above for management of a TPM field upgrade is intended for use in a TPM that is implemented as stand-alone component (that is, when the TPM is manufactured and sold as a component that is added to a platform). When the TPM is not a stand-alone component, other methods of field upgrade are possible and are not precluded by this specification.

If other methods are used, the security of that method is the responsibility of the platform manufacturer.

 Copyright © TCG 2006-2020

13 TPM Control Domains

13.1 Introduction

Three entities control the TPM: the platform firmware, the platform Owner, and the Privacy Administrator. The Owner and Privacy Administrator are often the same entity. This control does not give these entities the ability to access user keys or data, but it does give them the ability to control selected TPM resources.

Each of the three entities has its own domain of control. Within that domain are TPM resources reserved to that entity. Each entity exercises its control over its domain by use of domain-specific authorization values.

The NV space defined by the platform firmware has an additional control, *phEnableNV*. When SET, NV space defined by the platform firmware is accessible. When CLEAR, it is inaccessible. This permits independent control of the platform firmware hierarchy and its NV space. For example, the platform hierarchy can be disabled while still permitting access to platform firmware NV space.

13.2 Controls

The platform firmware, platform Owner, and Privacy Administrator each have an authorization value and an authorization policy to control some portion of the TPM, including a specific Primary Seed (see clause 14). The authorizations, policies, and Primary Seed for each domain are:

- *platformAuth/platformPolicy*/PPS for platform firmware;
- *ownerAuth/ownerPolicy*/SPS for the Owner; and
- *endorsementAuth/endorsementPolicy*/EPS for the Privacy Administrator.

Associated with each hierarchy is a logical switch (that is, an "enable") that determines whether the hierarchy is enabled. These enables are *phEnable*, *shEnable*, and *ehEnable*.

When the enable for a hierarchy is SET (1) and the specification indicates that an action may be authorized with an authorization value, the corresponding policy is also allowed. For instance, when *phEnable* is SET and *platformAuth* is allowed, *platformPolicy* may also be used.

When the enable for a hierarchy is CLEAR, neither the corresponding *authValue* nor *authPolicy* may authorize operations.

The interaction of the two authorization types (value and policy) and the associated hierarchy enable are intended to provide a flexible set of controls. Table 2 shows the control combinations.

Table 2 shows the *authValue* as either being "Known" or "Unknown". These correspond to the enabled and disabled states for an *authValue*. When the *authValue* is known, it can be used for authorization, but it cannot be used when the *authValue* is unknown. Since a zero-length string (Empty Buffer) is a valid, knowable *authValue*; the way to make the *authValue* unknown, and disable its use, is to set it to a large random number and then discard that number.

Table 2 shows the *authPolicy* as either being "Set" or "Empty". These also correspond to the enabled and disabled states for an *authPolicy*. An *authPolicy* will have to match the value of a digest (*policyDigest*) in order for it to be a valid authorization. Since no digest has a zero length, setting the *authPolicy* to an Empty Buffer will disable use of the *authPolicy*. It is also possible to disable use of the *authPolicy* by setting it to any value that does not represent a known policy but the conventional way to disable use of *authPolicy* is to set it to an Empty Buffer (see 19.7 for the description of *policyDigest* generation and use).

 Copyright © TCG 2006-2020

Table 2 — Hierarchy Control Setting Combinations

hierarchy *enable*	*authValue*	*authPolicy*	Description
SET	Known	Set	The hierarchy is enabled, and objects in it may be loaded. Either authValue or authPolicy may manage resources related to the hierarchy.
SET	Unknown	Set	The authValue may be made unknown by setting it to a random value and then discarding the value. This prevents the authValue from being used. This combination is useful for keeping the hierarchy enabled but using a policy-based delegation scheme for managing hierarchy-related resources. An example is delegating control of creating Primary Objects in a hierarchy to one entity while delegating control of related NV resources to a different entity.
SET	Known	Empty	When the authPolicy is empty, it cannot match any policyDigest value so the use of authPolicy is disabled. This combination is most analogous to the control scheme of TPM 1.2, where an authValue (ownerAuth) is used to manage the resources of the single hierarchy supported by a 1.2 TPM.
CLEAR	N/A	N/A	When an enable is FALSE, the corresponding authValue and authPolicy may not be used to authorize any TPM action.

TPM2_HierarchyChangeAuth() may change the *authValue* associated with a hierarchy but only if the hierarchy is enabled. Either the *authPolicy* or the *authValue* of a hierarchy may be used to authorize a change to the *authValue*.

13.3 Platform Controls

The platform firmware has overall control of the TPM and the availability of the TPM to the platform Owner or Privacy Administrator. The platform firmware is assumed to be provided by the platform manufacturer and performs the management of the hardware in preparation for use by an operating system (the operating system may be provided by a different vendor). In some systems, platform firmware runs after the OS is loaded. Often this firmware is required to ensure the safety of the system.

EXAMPLE Some systems have thermal properties that, if not managed properly, could lead to destruction of the system, and could even lead to the system becoming a fire hazard.

If the firmware is crucial to the safety of the system, the platform manufacturer may design in a firmware update process that ensures that only firmware approved by the manufacturer for a specific machine is allowed to be loaded on the system. This firmware may use cryptography to validate the firmware update before it is loaded. The TPM has cryptographic functions that are similar or identical to the functions needed by the platform firmware for its management of the system. Rather than replicate those cryptographic capabilities, the platform firmware is given its own set of TPM resources for its use. Reuse of the TPM cryptographic capabilities by the platform is intended primarily as a cost savings.

The platform manufacturer decides if it is possible to disable use of the TPM by the platform. The method for disabling use of the TPM by the platform is platform-manufacturer specific.

The properties of the TPM required by the platform manufacturer need not match those of the Owner. The platform manufacturer decides what cryptographic algorithms are required to safeguard the platform. These algorithms may differ from the algorithms use by the Owner or the Privacy Administrator.

Platform controls allow the following operations not available to an ordinary TPM user:

- allocation of TPM NV memory;
- PCR configuration;

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

- control of the availability of any key hierarchies; and
- change of the PPS, SPS, and EPS and reset of associated authorization values and policy.

NOTE 1 This is not a comprehensive list. The uses of the platform controls are documented in TPM 2.0 Part 3. In that document, an authorization of a command that allows the use of the platform handle (TPM_RH_PLATFORM) indicates that the command accepts *platformAuth* or *platformPolicy*.

phEnable gates use of both *platformAuth/platformPolicy* and the PPS hierarchy, as described in the previous clause. When *phEnable* is CLEAR, a _TPM_Init is required to SET it.

On any _TPM_Init, *phEnable* is SET to ensure that the platform may use the TPM during its initialization.

On TPM Reset or TPM Restart, *platformAuth* is set to an EmptyAuth, and *platformPolicy* is set to an Empty Policy.

NOTE 2 Platform controls are reset on TPM Restart because the BIOS goes through a full initialization and has no memory of any previous authorization values.

NOTE 3 phEnable must be SET before TPM2_Startup when accommodating the case of an interrupted field upgrade that prevents startup from running. phEnable must be SET to permit field upgrade authorization.

A *platformAuth/platformPolicy* may be used in TPM2_HierarchyControl() to SET or CLEAR *shEnable* or *ehEnable*.

13.4 Owner Controls

The TPM controls available to the Owner are a subset of those available to the platform. These include

- allocation of TPM NV memory, and
- control of the availability of any storage hierarchies.

The s*hEnable* gates use of both *ownerAuth/ownerPolicy* and the SPS hierarchy, as described in 13.2.

The s*hEnable* is SET on each TPM Reset, TPM Restart, or when the SPS is changed (TPM2_Clear()). The s*hEnable* may be CLEAR (TPM2_HierarchyControl()) using either Lockout Authorization or Platform Authorization. When *shEnable* is CLEAR, it may only be SET (TPM2_HierarchyControl()) if Platform Authorization is provided.

The o*wnerAuth* and *ownerPolicy* values are persistent. They are set to standard initialization values when the SPS is changed (TPM2_Clear()): *ownerAuth* is set to an EmptyAuth, and *ownerPolicy* is set to an Empty Policy. They may be explicitly changed by designated commands.

13.5 Privacy Administrator Controls

The Privacy administrator has control over the Endorsement Hierarchy and reporting of privacy-sensitive data.

The Privacy Administrator uses *endorsementAuth* and *endorsementPolicy* to exercise its control. The Privacy Administrator has a more limited domain of control than those of the platform firmware and the Owner. The cases when *endorsementAuth* or *endorsementPolicy* are required are:

- when creating Primary Objects in the Endorsement hierarchy, and
- when controlling the availability of the Endorsement hierarchy.

 Copyright © TCG 2006-2020

Other actions that may be considered to be privacy-sensitive require use of objects in the Endorsement hierarchy. For example, certification of a TPM object by the TPM produces a data structure that has data that could allow cross-correlation of the objects. This data is obfuscated unless the certifying key is in the Endorsement hierarchy. The privacy administrator of the TPM is expected to manage the creation of objects in the Endorsement hierarchy to ensure that the use of those objects is in accordance with their privacy policy.

The e*hEnable* gates use of *endorsementAuth/endorsementPolicy* and the EPS hierarchy, as described in 13.1. It also gates use of the vendor-specific handles TPM_RH_AUTH_00 to TPM_RH_AUTH_FF. Additionally, when the SPS changes, the objects in the EPS hierarchy are flushed from the TPM, and new EPS objects (that is, Primary Objects) must be created.

NOTE Clearing the hierarchy is necessary to ensure that the new Owner may not abuse objects created by a previous one and so that objects belonging to the previous Owner may not compromise the new one.

The e*hEnable* is SET on each TPM2_Startup(TPM_SU_CLEAR) (that is, TPM Reset or TPM Restart) or when the SPS is changed (TPM2_Clear()). The e*hEnable* may be CLEAR using either Endorsement Authorization or Platform Authorization. When *ehEnable* is CLEAR, it may be SET using Platform Authorization

NOTE TPM2_HierarchyControl() will SET or CLEAR *ehEnable* if the proper authorization is provided.

The e*ndorsementAuth* and *endorsementPolicy* values are persistent. They are set to standard initialization values when the SPS (TPM2_Clear()) or EPS (TPM2_ChangeEPS()) are changed: *endorsementAuth* is set to an EmptyAuth, and *endorsementPolicy* is set to an Empty Policy. They may be explicitly changed by designated commands.

13.6 Primary Seed Authorizations

Use of a Primary Seed to create a Primary Object requires use of the authorization associated with that Primary Seed: Platform Authorization for the PPS, Owner Authorization for the SPS, and Endorsement Authorization for the EPS.

13.7 Lockout Control

A TPM is required to implement a lockout mechanism to protect against so-called “dictionary attacks,” where an attacker tries numerous authorization values until one succeeds. Dictionary attack protection is common for security devices, such as smartcards, that use human input for authorization. A human source of authorization likely has too little entropy to protect against an automated attack, so logic that prevents high-speed guessing of the values is required.

When the dictionary attack lockout is engaged, preventing use of some resources, it is helpful to have a secret value that resets lockout. The TPM stores the secret value as *lockoutAuth.* Alternatively, a policy *(lockoutPolicy)* can be used to reset lockout.

NOTE 1 The primary attack model for the dictionary attack begins when a system falls into the hands of a thief. The thief tries to recover data on the system by guessing the password used to protect a disk’s encryption keys. The dictionary attack logic defeats this attack by preventing the thief from making many guesses before the TPM locks out further attempts. When/if the system is returned to its rightful owner, that owner can enter the *lockoutAuth* value or satisfy *lockoutPolicy*, access the disk encryption keys, and return to normal operation.

NOTE 2 Unfortunately, dictionary attack logic is not forgiving of poor typing or a short memory. If someone types his or her password incorrectly due to clumsiness or poor memory, the dictionary attack logic might not differentiate this from an attack, so it locks the TPM. Lockout Authorization allows recovery from this situation.

 Copyright © TCG 2006-2020

The *lockoutAuth* value is reset to EmptyAuth and *lockoutPolicy* to the Empty Buffer when TPM2_Clear() is executed.

NOTE 3 TPM2_Clear() changes the SPS rendering all previously-created user objects inaccessible. There are, therefore, no keys for the dictionary attack logic to protect.

The l*ockoutAuth* value may be changed (TPM2_HierarchyChangeAuth()) only when its current value is provided. *LockoutPolicy* may be changed using TPM2_SetPrimaryPolicy().

Generally, dictionary attack protection is not applied to objects associated with the PPS or to NV Indexes defined using Platform Authorization. The platform firmware is expected to select a high-entropy value when setting the *platformAuth* after a TPM reset. Additionally, since Platform Authorization does not provide access to user data protected by the TPM, disclosure of *platformAuth* does not expose user secrets.

See 19.8 for full details on setting of parameters associated with dictionary attack logic and other aspects of the dictionary attack protection.

13.8 TPM Ownership

13.8.1 Taking Ownership

Taking ownership of a TPM is the process of inserting authorization values for the *ownerAuth*, *endorsementAuth*, and *lockoutAuth*.

A TPM that has been cleared (TPM2_Clear()) has its *ownerAuth*, *endorsementAuth*, and *lockoutAuth* values set to EmptyAuth and its *ownerPolicy, endorsementPolicy,* and *lockoutPolicy* values set to Empty Buffers. The OS is expected to change these values and manage them on behalf of the platform Owner.

The platform may prevent access to the hierarchies associated with Owner Authorization and Endorsement Authorization and prevent use of the TPM's persistent storage by the operating system and user applications. TPM cryptographic capabilities would still be available, and these could be used as if the TPM were a software cryptographic library.

13.8.2 Releasing Ownership

TPM2_Clear() clears the current Owner from the TPM. A persistent TPM control (TPMA_PERMANENT.*disableClear*) controls whether TPM2_Clear() is functional. If *disableClear* is CLEAR, then TPM2_Clear() may be authorized using either Platform Authorization or Lockout Authorization. If the control is SET, then TPM2_Clear() is not functional.

NOTE TPMA_PERMANENT.*disableClear* may be SET or CLEAR using platformAuth/platformPolicy, giving the platform the ability to enable execution of TPM2_Clear() when needed.

TPM2_Clear() instructs the TPM to:

- flush any transient or persistent objects associated with the SPS or EPS hierarchies (PPS objects are not affected);
- release any NV Index locations that do not have their TPMA_NV_PLATFORMCREATE attribute SET;
- set *shEnable* and *ehEnable* to TRUE;
- set ownerAuth, endorsementAuth, and lockoutAuth to an EmptyAuth;
- set ownerPolicy, endorsementPolicy, and lockoutPolicy to an Empty Policy;

- replace the existing SPS with a new value from the RNG; and
- recompute *shProof*, and *ehProof*.

 Copyright © TCG 2006-2020

14 Primary Seeds

14.1 Introduction

A Primary Seed is a large, random value that is persistently stored in a TPM; it is never stored off the TPM in any form. Primary Seeds are used in the generation of symmetric keys, asymmetric keys, other seeds, and proof values.

A Primary Seed generates Primary Objects using the methods described in Clause 27.5. In brief, the caller provides the parameters of an object to be created, and the TPM uses these parameters and the Primary Seed in a key derivation function (KDF) to produce an object of the desired type. After the TPM generates a Primary Object, it uses the parameters of that object and the Primary Seed to generate a symmetric key to encrypt the sensitive portion of the object (that is, the private data and authorizations). It then returns the public portion and name of the object to the caller. The Primary Object may then be context saved and loaded like any other object. It may be stored persistently in the TPM's NV memory (TPM2_EvictControl()).

Primary Seeds generate only Primary Objects. All other objects use the random number generator of the TPM as the source of entropy for generating secrets in the object.

14.2 Rationale

The algorithm flexibility provided by this specification makes it possible for the TPM to support many different asymmetric key types. TPM 1.2 supported only the RSA algorithm with a limited number of commonly used parameters. The addition of ECC support significantly increases the number of parameters because curve parameters may vary based on application.

While this flexibility is a major benefit of TPM 2.0, it creates new challenges for managing TPM Endorsement Keys (EKs) and EK certificates. As mentioned in 9.4.3.2, an EK is an identity for the Root of Trust for Reporting (RTR), and algorithm agility creates the possibility of having many identities for the same RTR, with each identity based on a different set of cryptographic algorithms.

One possible approach for handling many EKs and their associated certificates would be for the TPM manufacturer to have the TPM create EKs for many key parameters and store them on the TPM; in this way, a key with the correct parameters would be available in most situations. The TPM vendor could then create one or more certificates for those keys. However, this approach would require a prohibitive amount of NV memory to store all the key pairs and associated parameters. The approach taken in this specification allows certification of a large number of EKs with different parameters without requiring that any of them be stored in persistent TPM memory.

The mechanism of this specification uses a persistent, randomly generated seed value from which EKs are derived. The derivation process lets the TPM generate a different EK for each set of key parameters. As long as the seed value does not change, the same key parameters generate the same EKs.

The typical use of this EK generation approach is as follows: The TPM manufacturer or the platform manufacturer has the TPM create a new Endorsement Primary Seed (EPS) and then generate key pairs based on sets of input parameters and that EPS. The TPM retains the generated keys. Combinations of key parameters should be chosen to ensure that likely TPM users would find a combination to suit their needs. The manufacturer then generates one or more certificates for the generated public keys and then ships the TPM/system with no EK pair stored on it. The system owner decides which key types are needed, and the parameters for those types are entered into the TPM. If the parameters are the same as those used by the manufacturer, the TPM generates the same key pair. The system owner then has an EK with its certificate. Since an EK is not generally duplicable, the owner has a choice to make. They may either re-create the EK whenever it is needed or tell the TPM to save the EK in persistent memory.

 Copyright © TCG 2006-2020

The seed key concept may be applied to two other TPM key hierarchies: one used by platform firmware, and one used for the owner's Storage hierarchy. The Endorsement Keys (EK) are generated from the Endorsement Primary Seed (EPS), platform keys from the Platform Primary Seed (PPS), and Storage Root Keys (SRK) from the Storage Primary Seed (SPS). Each seed value has a different life cycle, but the way it seeds the associated hierarchies is approximately the same.

It is preferred that a TPM manufacturer generate a certificate for at least one EK before the device ships. This certificate would be based on the EPS that is present in the TPM at that time. While it is possible for the manufacturer to let the TPM populate the EPS and generate an RSA key pair, the unpredictability of the generation time may make injecting an EPS a more attractive option. The time taken to inject an EPS would be deterministic and one or more RSA key pairs could be generated for that EPS outside of the TPM. This could save considerable time during manufacturing.

The external algorithm for generating a key pair from the EPS has to be the same as the algorithm used by the TPM; otherwise, they will generate different keys. The generation times for the external and TPM processes will be proportional so the manufacturer can use the time for external key generation time as an indicator of the time that the TPM will take when the end user attempts to recreate the EK. If the manufacturer does decide to inject an EPS and generate RSA keys outside of the TPM, there is an opportunity to benefit the customer by discarding EPS values that result in long key pair generation times for the certified values.

Another possible option is to inject the EPS and a precomputed pair of RSA primes that are compatible with a specific template (compatible meaning that the primes are the right size and that the prime (p) and p-1 are not evenly divisible by the public exponent). The TPM could access those primes when the associated template is used for an EK. If this method is used, the TPM manufacturer has to make sure it is critical that the precomputed primes are only associated with a single template and that the primes are erased from the TPM when the EPS is changed.

The primary seed approach allows multiple storage hierarchies with differing security properties, as needed by various applications, without requiring that all of the SRKs occupy persistent TPM memory. An SRK may be made persistent in TPM NV memory if required by the application.

This scheme is also used in support of the Platform hierarchy for implementation simplicity.

14.3 Primary Seed Properties

14.3.1 Introduction

A Primary Seed is required to have at least twice the number of bits as the security strength of any symmetric or asymmetric algorithm implemented on the TPM.

EXAMPLE 1 RSA2048 is considered to have a security strength of 112 bits. If it were the strongest algorithm on the TPM, then the required size of an associated Primary Seed would be at least 224 bits.

EXAMPLE 2 If AES256 were implemented, the Primary Seed would be 512 bits even if: (1) the desired security strength is 196 bits, and (2) AES256 is used only for convenience, as is the case with Suite B.

A different authority controls each Primary Seed. In normal use, Primary Seeds are expected to have different lifetimes.

After a field upgrade that changes the Primary Seed strength, or that changes the algorithm that uses the Primary Seed, the TPM shall generate the original EKs corresponding to the EK certificates provisioned by the TPM manufacturer if the same template is provided to the TPM2_CreatePrimary() command until such time as TPM2_ChangeEPS command changes the EPS.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

For a field upgrade that does not change the Primary Seed strength or the algorithm that uses the Primary Seed and does not otherwise affect the security of Primary Objects based on the seeds, TPM2_CreatePrimary() with the same inputs should produce the same Primary Object in the platform, storage, and endorsement hierarchies after the field upgrade as it did before the field upgrade.

This requirement shall not be in effect for other keys derived from the EPS or for keys derived from the SPS or PPS.

EXAMPLE A field upgrade can cause TPM2_CreatePrimary() to generate a different key for the same input template. For example, revisions prior to revision 01.38 used KDFa, while revision 01.38 and after use DRBG. In addition, the security strength requirement could cause a change in the seed length if the field upgrade implements a stronger algorithm.

14.3.2 Endorsement Primary Seed (EPS)

The EPS is used to generate EKs and is the basis of the RTR's identity.

The TPM creates an EPS whenever it is powered on and no EPS is present. TPM2_ChangeEPS() may change the EPS (replace it with a new EPS), but this requires authorization by Platform Authorization.

The TPM manufacturer may inject an EPS and, under controlled conditions, compute the asymmetric EKs that the TPM would generate given specific input parameters. Only the TPM vendor may inject an EPS.

When an EPS is replaced, all objects in the Endorsement Hierarchy are invalidated, and certificates associated with the EKs generated from that EPS are no longer useful. This means that certificates for new EPS-based EKs may be needed. The environment in which this process occurs may not provide assurance that the EKs are generated from a genuine TPM. To support recertification in such an environment, the TPM allows cross certification of keys between the Platform hierarchy and the Endorsement hierarchy under control of the platform firmware. Cross certification allows a chain of trust to be maintained as the seeds are changed.

When a platform enters the distribution channel, it is expected to have a certificate for at least one EK for the TPM on that platform.

Either e*ndorsementAuth* or *endorsementPolicy* is required to use the EPS for creation of a Primary Object in the Endorsement hierarchy.

14.3.3 Platform Primary Seed (PPS)

The PPS is used to generate the hierarchy controlled by platform firmware. The hierarchies derived from this seed are for exclusive use by platform firmware and should not be made available to user-installable software (such as, OS and applications).

NOTE 1 The platform firmware may be changed because of actions by a person with possession of the platform, but that is not included in the definition of user-installable software.

The TPM creates a PPS whenever it is powered on and no PPS is present. TPM2_ChangePPS() may change the PPS (replace it with a new PPS), but this requires authorization by Platform Authorization.

A PPS may be injected but only by the TPM manufacturer.

Platform Authorization is required to use the PPS to create a Primary Object in the Platform hierarchy.

The authorization for use of objects in the PPS hierarchy should use a policy containing a reference to *platformAuth* and not be based on a key-specific authorization value.

NOTE 2 The TPM does not enforce this imperative.

 Copyright © TCG 2006-2020

NOTE 3 A simple way to achieve this control is to create a policy that references *platformAuth* in a TPM2_PolicySecret(). If the only component of the policy is TPM2_PolicySecret() referencing TPM_RH_PLATFORM, the policy may be the same for all objects in the Platform hierarchy and for all platforms that implement the chosen policy hash.

14.3.4 Storage Primary Seed (SPS)

The SPS is used to generate hierarchies controlled by the platform owner. This seed generates the keys that serve as Storage Root Keys for normal OS and application use.

The TPM creates the SPS whenever it is powered on and no SPS is present. TPM2_Clear() may be used to change the SPS if the TPM owner wants to ensure that no previously generated keys in the Storage hierarchy may be used in the future.

Changing the SPS invalidates all objects in the Storage Hierarchy and they cannot be recreated. Changing the SPS also invalidates all objects in the Endorsement Hierarchy and only the Primary Objects in the Endorsement Hierarchy may be recreated.

14.3.5 The Null Seed

The Null Seed is set to a random value on every TPM reset. The Null Seed can be used to generate hierarchies (primary objects and children of primary keys) that are only usable until the next TPM reset.

Objects in the null-hierarchy cannot be made into persistent objects. However, in other respects objects in this hierarchy behave like objects in the other hierarchy.

14.4 Hierarchy Proofs

The TPM uses a proof value to prove that it created or checked an externally provided value. A proof value is associated with a hierarchy and is statistically unique. The proof values are used in tickets. The tickets use the hierarchy-specific proof values. A ticket may not be used when its associated hierarchy is disabled.

EXAMPLE 1 The TPM may validate asymmetrically signed data. After doing so, it produces a ticket that is an HMAC over the signed data, with the HMAC key being a proof value. This proves to the TPM that it has already checked the asymmetric signature, so it does not have to do so again. Subsequently, when the TPM needs to check that the data was properly signed, it may use symmetric cryptography (a hash) rather than asymmetric cryptography to validate the signature.

EXAMPLE 2 When the TPM performs TPM2_ContextSave() on an object in the Storage hierarchy, it may include the Storage hierarchy proof (*shProof*) in the object’s integrity value. When the SPS is changed, *shProof* will change so that the saved contexts may not be reloaded.

A Platform hierarchy proof (*phProof)*, used for objects associated with the Platform hierarchy. *phProof* changes when the PPS changes. An *shProof*, used for the Storage and Endorsement hierarchies, changes when the SPS changes.

NOTE It is possible to create objects in the Endorsement Hierarchy that are not Primary Objects. Those Ordinary Objects are considered to belong to a specific TPM Owner. A change of the SPS indicates a change of Owner for the TPM. Inclusion of *ehProof* in the protection of Ordinary Objects in the Endorsement Hierarchy ensures that those Objects will be deleted when the Owner changes, because *ehProof* also changes when the Owner changes.

A proof is a value that may be kept in permanent storage on the TPM or it may be regenerated from the PPS or SPS on each boot or as needed. A proof value is never stored off the TPM in any form. Hierarchy proof values are only used as an HMAC key if the result of the computation is stored off the TPM. Examples are saved contexts and tickets. A hierarchy proof value may be used in other computations as long as the result of the computation does not leave the TPM.

 Copyright © TCG 2006-2020

The TPM should produce proof values that are the larger of either

- the size of the largest digest produced by any hash algorithm implemented on the TPM, or
- twice the size of the largest symmetric key supported by the TPM.

EXAMPLE 3 If the TPM implements SHA384 and AES256, the proof value will have a size of 512 bits.

NOTE According to SP800-57, the security strength of SHA256 in an HMAC function equals 256 bits. Since security strength is not improved when the key size is larger than the digest size, the recommendation for proof size provides the appropriate strength when the TPM is implementing balanced algorithm sets. A TPM using SHA256, ECC256, and AES128 is balanced, and the proof value is 256 bits.

 Copyright © TCG 2006-2020

15 TPM Handles

15.1 Introduction

TPM resources are referenced by handles that uniquely identify a resource that occupies TPM memory — either RAM or NV. A handle is a 32-bit value. Its most significant octet identifies the type of referenced resource. At any given instant, its low-order 24 bits identify a unique resource of that type. The actual resource identified by the low-order 24 bits may change with time.

A specific handle value may refer to only one TPM-resident resource at a time.

15.2 PCR Handles (MSO=00_{16})

To reduce confusion, PCR are assigned handles that have the same values as in previous versions of the specification. A PCR handle is an Index into an array of PCR. A PCR's Index and handle value are the same.

15.3 NV Index Handles (MSO=01_{16})

An NV Index is associated with a persistent TPM resource created by TPM2_NV_DefineSpace().

15.4 Session Handles (MSO=02_{16} and 03_{16})

The TPM assigns session handles when an authorization session is started (TPM2_StartAuthSession()). An HMAC session is assigned a handle with an MSO of 02_{16} and a policy session is assigned a handle with an MSO of 03_{16}. Each authorization session handle is associated with a unique context that may exist in only one place at a time: either on the TPM in a Shielded Location, or in a saved context as a Protected Object. The handle remains associated with the session as long as the session exists and does not change when the session is context-saved and reloaded.

The low order 3 octets of each session handle are unique. They are assigned interchangeably to HMAC or policy sessions but to only one at a time.

EXAMPLE 1 If a policy session has a value of 03 00 00 01_{16}, then an HMAC session with a value of 02 00 00 01_{16} will not be assigned at the same time.

NOTE 1 The policy and session handles are assigned from a common pool of handle values.

When TPM2_GetCapability() is used to obtain a list of sessions that are currently loaded on the TPM, the caller would use a handle with an MSO of 02_{16}. While this would normally be an HMAC handle reference, the TPM will respond with a list that includes both HMAC and policy sessions. The handles will be returned in ascending order of the low-order three octets.

EXAMPLE 2 A list of loaded handles returned by the TPM in response to a TPM2_GetCapability(*capability* = TPM_CAP_HANDLES, *property* = 02 00 00 00_{16}), the TPM might return the list: 02 00 00 02_{16}, 03 00 00 04_{16}, and 02 00 00 05_{16}

When TPM2_GetCapability() is used to obtain a list of sessions that are active but not on the TPM, the caller would use a handle with an MSO of 03_{16} which normally would reference a policy session. The TPM will respond with a list of session handles that are in use, but not on the TPM. Since the TPM does not keep a record of whether the saved session context was an HMAC or policy session, all of the handles in the list will have an MSO of 02_{16}.

The TPM is required to maintain a list of all, currently assigned session handles as well as the correct "version number" for any saved session contexts.

 Copyright © TCG 2006-2020

NOTE 2 the "version number" is how the TPM prevents replay of an authorization.

When an authorization session is no longer needed, TPM2_FlushContext() may be used to delete all context associated with the session from TPM memory (see 30.6). The session handle for this command may use an upper octet of either 02_{16} or 03_{16}.

NOTE 3 Flushing a session context deletes any data in the TPM relating to the context and frees the handle associated with that context and invalidates the version number of any saved context.

NOTE 4 An alternative method of flushing a session context exists that is not available for other entities. On the last use of the session, the caller may indicate (in one of the session attributes) that the session is no longer needed. If the command completes successfully, the TPM will complete the response computations for the session and delete the session context from TPM memory (see 18.6.4).

All session contexts in TPM memory are flushed on any TPM2_Startup(). The saved session contexts remain valid until a TPM Reset.

15.5 Permanent Resource Handles (MSO=40_{16})

Fixed resource handles refer to Shielded Locations that are always associated with the same handle. These resources have handles with an MSO of 40_{16}. Examples of these resources are Owner, Platform, and Endorsement hierarchy controls and the Lockout authorization value.

Another type of permanent resource handle is the vendor-specific authorization value. These optional resources may be populated with authorization values that are known only by the TPM manufacturer or some other privileged entity. The update of these authorization values is TPM-manufacturer-dependent.

If present, a vendor-specific authorization value can be used as a bind value within an authorization session or to authorize a policy using the TPM2_PolicySecret command. In the former case, an entity that knows the authorization value could create an auditable authorization session that only that entity could execute. In the latter case, the entity could create and/or use TPM resources with an authorization policy that only that entity could execute.

Since vendor-specific authorization values might be usable by an entity who knows them to identify the TPM, the use of these authorization values is under the control of the privacy administrator. These authorization values are only usable when the Endorsement Hierarchy is enabled as described in 13.5.

NOTE A use case for the vendor-specific authorization values is to recover in the field from a flaw in the TPM firmware. For example, TPM vendors may provide a mechanism that updates one or more of these authorization values based on the measurement of the TPM firmware. This update mechanism could be used to give the manufacturer confidence that a valid, uncompromised version of the TPM firmware is running. In this scenario, if the manufacturer wished to provide a certificate for an endorsement key generated in the field after a field upgrade to a trusted firmware version occurred, the manufacturer could use an auditable authorization session using the vendor-specific authorization value to verify the properties of the endorsement key and then create a certificate for that new endorsement key.

15.6 Transient Object Handles (MSO=80_{16})

The TPM assigns Object handles when an Object is loaded or when the Object's persistence is changed (TPM2_EvictControl()). Transient Objects in TPM RAM have handles with an MSO of 80_{16}; they may have a different value for the three LSOs each time the Object is used. This is because the Object's context may have been swapped out and the TPM assigned a new handle when the object was swapped back in. The TRM ensures that the handle references the correct object.

All Transient Objects are flushed from TPM memory on any TPM2_Startup(). A loaded Transient Object context may be flushed from TPM memory using TPM2_FlushContext() and indicating the handle of the loaded context to be flushed.

15.7 Persistent Object Handles (MSO=81_{16})

TPM2_EvictControl() may make a Transient Object into a Persistent Object. A Persistent Object, placed in the TPM's NV memory, is not cleared by a TPM2_Startup().

Making an Object persistent requires either Platform Authorization or Owner Authorization.

When the TPM changes a Transient Object to a Persistent Object, the caller indicates the handle to be assigned to the Persistent Object. The MSO of the handle is required to be 81_{16}. The next most significant bit is required to be CLEAR if the authorization is provided using Owner Authorization and SET if the authorization is provided using Platform Authorization. If the handle is not already in use, and space is available, a persistent copy of the Object is created and assigned the handle provided by the caller. This handle always references the same Persistent Object as long as it remains persistent. The handle assigned to a Persistent Object may be assigned to a new Persistent Object if the first Object is deleted from persistent storage.

 Copyright © TCG 2006-2020

16 Names

The Name of an entity is its unique identifier. The handle associated with an object may change due to context management (TPM2_ContextSave() / TPM2_ContextLoad()), but the Name of an object remains constant. The Name associated with an NV Index will change based on changes to the attributes of the Index.

EXAMPLE When an NV Index is initially defined, it will have a Name for an Index with TPMA_NV_WRITTEN CLEAR. After the Index is written, the Name will change to reflect that TPMA_NV_WRITTEN is SET for the Index.

When an NV Index becomes locked (TPMA_NV_WRITELOCKED or TPMA_NV_READLOCKED is SET), the Name of the NV Index changes. This has two implications:

The caller should use its copy of the NV public area and calculate the Name before using it in an HMAC authorization calculation. Otherwise, an invalid authorization may trigger the dictionary attack protection depending on TPMA_NV_NO_DA.

The TPM must check access control before checking authorization. For example, it should reject a read to a read locked NV Index before doing an authorization check that might trigger the dictionary attack protection.

The method of computing the Name for an entity varies according to the entity type that is the MSO of the handle. Table 3 shows the method and the handle's MSO for different entity types.

When the computation of a Name uses a hash algorithm, the algorithm identifier is included in the Name structure. If the Name is a handle, the Name is only the handle value.

Table 3 — Equations for Computing Entity Names

<table>
<tr><th>MSO of Handle</th><th>Entity Type</th><th>Equation for Computing the Name</th></tr>
<tr><td>00_{16}</td><td>PCR</td><td rowspan="4">$Name \coloneqq handle$
No hash is performed on the handle to produce the name and the name is only the size of the handle.</td></tr>
<tr><td>02_{16}</td><td>HMAC Session</td></tr>
<tr><td>03_{16}</td><td>Policy Session</td></tr>
<tr><td>40_{16}</td><td>Permanent Values</td></tr>
<tr><td>01_{16}</td><td>NV Index</td><td>$Name \coloneqq nameAlg \mid\mid \mathbf{H}_{nameAlg}(handle \rightarrow nvPublicArea)$
where
$nameAlg$ algorithm used to compute $Name$
$\mathrm{H}_{nameAlg}$ hash using the $nameAlg$ parameter in the NV Index location associated with $handle$
$nvPublicArea$ contents of the TPMS_NV_PUBLIC associated with $handle$</td></tr>
<tr><td>80_{16}</td><td>Transient Objects(1)</td><td rowspan="2">$Name \coloneqq nameAlg \mid\mid \mathbf{H}_{nameAlg}(handle \rightarrow publicArea)$
where
$nameAlg$ algorithm used to compute $Name$
$\mathrm{H}_{nameAlg}$ hash using the $nameAlg$ parameter in the object associated with $handle$
$publicArea$ contents of the TPMT_PUBLIC associated with $handle$</td></tr>
<tr><td>81_{16}</td><td>Persistent Objects</td></tr>
<tr><td colspan="3">NOTE 1) The Name of a sequence object is an Empty Buffer (see 32.4.5).</td></tr>
</table>

When an object is created, a "template" for the public area is used to define the properties for the new object. That template has the structure of an object's public area. The Name of a public area template is computed in the same way as the Name of a Transient Object.

 Copyright © TCG 2006-2020

17 PCR Operations

17.1 Initializing PCR

All platform configuration registers (PCR) are reset to their default initial condition on TPM Reset and TPM Restart. Some PCR may be designated as being preserved by TPM Resume. Those that are preserved are restored to the state that they had at the last TPM2_Shutdown(STATE) operation. When TPM2_Startup() completes successfully, PCR that are not designated as being preserved by TPM Resume will be in their default initial condition.

If allowed by its attributes, a PCR may also be reset by TPM2_PCR_Reset() or by a Dynamic Root of Trust (D-RTM) sequence (see 34.2). PCR attributes are defined in a platform-specific specification. They determine the reset value of a PCR as well as the localities required to perform the reset.

The default initial condition for any PCR, other than PCR[0], is either all bits CLEAR or all bits SET. For PCR[0], the default initial condition may all bits CLEAR, all bits SET, the locality at which TPM2_Startup() was received, or an indicator that the first measurement came from an H-CRTM. Other platform types may use other means of identifying the locality of the access.

A platform-specific specification may choose from the options list above.

EXAMPLE 1 A platform-specific specification may designate that the default initial condition for PCR[0-16] is all zeros, and for PCR[17-20], it is all ones.

EXAMPLE 2 A platform-specific specification may designate that the default initial condition for PCR[0] is the locality indicator and that PCR[1-16] have an initial condition of all zeros.

NOTE The locality indicator is an integer value between 0 and the maximum locality implemented on a TPM. Currently, the maximum hardware locality is 4. In a TPMA_LOCALITY, a locality of four would be represented by the octet $0001\ 0000_2$. When encoded for a PCR initial value, locality 4 would be represented by the octet $0000\ 0100_2$.

EXAMPLE 3 A virtual TPM may use a unique identifier for each of the software entities that might access it. If specific software is associated with a specific PCR, then the reset value of that PCR may be the unique identifier of the software that is allowed to change it.

TPM2_PCR_Reset() requires that the proper authorization be provided for the operation (see 17.7).

17.2 Extend of a PCR

Other than reset, described above, the only way to change a PCR value is to Extend it. The Extend operation on a PCR is defined as

$$PCR_{new} \coloneqq \mathbf{H}_{alg}\left(PCR_{old} \mid\mid digest\right) \tag{13}$$

After each Extend, the PCR value is unique for the specific order and combination of digest values that were Extended.

Except for D-RTM, authorization is required to extend a PCR (see 17.7).

17.3 Using Extend with PCR Banks

TPM2_PCR_Extend() has a handle to indicate the PCR to Extend and the data to be Extended. Extended data is a structure that contains one or more digests along with the algorithm identifier for the digest(s). Each digest is Extended to the PCR bank that has the same algorithm. If no digest data is provided for one of the PCR banks, no change is made to the PCR in that bank.

 Copyright © TCG 2006-2020

The TPM should perform the following operation for each algorithm in which *pcrNum* is defined:

$$PCR.digest[pcrNum][alg]_{new} \coloneqq \mathbf{H}_{alg}(PCR.digest[pcrNum][alg]_{old} \,||\, digest) \tag{14}$$

where

$\mathbf{H}_{alg}$	hash function using the algorithm associated with the PCR instance
$PCR.digest$	digest value in a PCR
$pcrNum$	PCR numeric selector
alg	PCR algorithmic selector
$digest$	digest part of the list entry that has the same algorithm identifier as the PCR bank

EXAMPLE If a TPM supports three PCR banks (such as, SHA-1, SHA256, and SHA512), then an Extend to PCR[2] with a SHA-1 digest and SHA256 digest would be Extended to PCR[2] in the SHA-1 bank, and the SHA256 digest would be Extended to PCR[2] in the SHA256 bank. There would be no change to any PCR in the SHA512 bank.

17.4 Recording Events

An alternative way to record log entries is to input the full log entry to the TPM rather than performing the digests outside the TPM. This performs a hash on the log entry for each of the hash algorithms supported by the TPM. Events no larger than 1024 octets may use TPM2_PCR_Event(). Events exceeding 1024 octets may use the sequence commands: TPM2_HashSequenceStart(), TPM2_SequenceUpdate(), and TPM2_EventSequenceComplete()).

TPM2_PCR_Event() and TPM2_EventSequenceComplete() return a list of tagged digests. The digests are the digests of the event data using each implemented hash algorithm.

EXAMPLE For a TPM implementing two algorithms (such as, SHA256 and SM3), the event commands return a list of two tagged digests.

TPM2_EventSequenceComplete() requires that proper authorization be provided (see 17.7).

Recording of an event may also occur as the result of a _TPM_Hash_Start/_TPM_Hash_Data/_TPM_Hash_End sequence (an *H-CRTM Event Sequence*). The indications for the H-CRTM sequence come from the TPM interface and not through the command buffer. On receipt of _TPM_Hash_Start, the TPM will created an Event Sequence context. If no object context space is available when the TPM receives _TPM_Hash_Start, the TPM will flush a context (vendor's choice) in order to create the Event Sequence context. _TPM_Hash_Data is used to update the H-CRTM Event Sequence context and _TPM_Hash_End completes the sequence. The digest or digests computed during the H-CRTM Event Sequence will be extended into the PCR designated by the relevant platform-specific specification. A platform-specific specification may allow an H-CRTM Event Sequence before or after TPM2_Startup(). An H-CRTM Event prior to TPM2_Startup() affects PCR[0]. After TPM2_Startup(), an H-CRTM Event affects PCR[17].

17.5 Selecting Multiple PCR

TPM2_PCR_Event() implicitly selects all PCR with the same Index. Some commands allow the selection of multiple PCR in different banks. Examples are TPM2_PCR_Read(), TPM2_Quote(), and TPM2_PolicyPCR() that allow the caller to make arbitrary selections of PCR in multiple banks.

 Copyright © TCG 2006-2020

When a command allows multiple PCR to be selected, a list of selectors is used. Each entry in the list consists of an algorithm ID followed by a bit array. Each bit in the bit array corresponds to one PCR. If a bit is SET, then the indicated PCR in the bank corresponding to the algorithm ID is selected.

The bit correspondence to PCR is that the bit corresponding to PCR[n] is the ($n \bmod 8$) bit in the $\lfloor n/8 \rfloor$ octet of the array.

EXAMPLE An array to select PCR[0] and PCR[13] in a TPM with 16 PCR would be 01 20_{16}. The bit for PCR[0] is the 0 **mod** 8 = 0th bit in the $\lfloor 0/8 \rfloor = 0^{th}$ octet (the octet with the 01_{16} value) and the bit for PCR[13] is the 13 **mod** 8 = 5th bit in the $\lfloor 13/8 \rfloor = 1^{st}$ octet (the octet with the 20_{16} value).

The list of selectors is processed in order. The selected PCR are concatenated, with the lowest numbered PCR in the first selector being the first in the list and the highest numbered PCR in the last selector being the last.

TPM2_PCR_Read() returns a list of PCR values that correspond to the PCR selected in the selector list. TPM2_Quote() and TPM2_PolicyPCR() digest the concatenation of PCR.

It is not an error for the PCR selection to indicate a PCR that is not implemented in a bank. No value is included in the concatenation of PCR for an unimplemented PCR. It is an error if the algorithm ID selects a hash algorithm that is not implemented.

17.6 Reporting on PCR

17.6.1 Reading PCR

TPM2_PCR_Read() reads the current values of a selection of PCR. For this command, the caller indicates a list of PCR to be read using a PCR selection structure. This structure is an array of lists. Each array entry has a hash identifier and a bit field. The hash identifier indicates the bank of PCR, and the bit field indicates the PCR being selected in the bank.

In the response, the TPM provides a PCR selection structure and a list of PCR values. The PCR selection structure indicates the PCR that are present in the return structure. The size of the requested return data structure may not fit in the available TPM output buffer. In that case, the list of PCR values is truncated, and the response PCR selection structure indicates the PCR that were returned. If the returned structure does not contain all of the PCR, the caller may modify the selection structure and make another read request to get additional PCR values.

Since the PCR may change between the calls to collect the full set of PCR of interest, the TPM returns a counter that increments on most invocations of TPM2_PCR_Extend(), TPM2_PCR_Event(), TPM2_EventSequenceComplete(), or TPM2_PCR_Reset() (see 17.9 for exemptions). If this counter value changes between calls, the sequence may need to be repeated until the desired PCR are all returned with no change to the counter value.

17.6.2 Attesting to PCR

In some cases, it is necessary for selected PCR to be in a specific state. When indicating that state, it is not desirable to have to list the contents of each PCR. Instead, a digest of a concatenation of PCR (a composite PCR digest) will indicate the current contents of all of the PCR of interest.

The PCR to be included in the composite digest are selected by the same type of structure used for TPM2_PCR_Read(). The selection structure is first filtered so that unimplemented PCR are not in the selection structure. Then, a composite digest of all of the selected PCR is created. Finally, the filtered selection structure and the composite digest are hashed to create the final digest value. That digest may be compared to a required digest (TPM2_PolicyPCR()) or returned in an attestation (TPM2_Quote()).

 Copyright © TCG 2006-2020

To validate an attestation quote, a remote caller will typically use the PCR to recalculate the digest value. The TPM 1.2 quote command returns the PCR values along with the digest. In TPM 2.0, because of hash agility, the PCR set could have exceeded the response buffer size. Therefore, TPM2_Quote() returns only the digest, and the PCR values must be retrieved separately.

This can lead to a race condition. The PCR values can change between the time of the quote and the time they are read. There are several solutions. The PCR can be read before and after the quote to ensure that they did not change. Alternatively, the quote digest can be validated locally against the PCR before returning results to a remote caller, and the quote can be rerun until the validation succeeds.

17.7 PCR Authorizations

TPM2_PCR_Reset(), TPM2_PCR_Extend(), TPM2_PCR_Event(), and TPM2_EventSequenceComplete() require authorization for the PCR being modified. The type of the authorization may differ based on the PCR being modified. A PCR may be defined as having a fixed, EmptyAuth; a variable *authValue;* or a variable *authPolicy*.

The authorization (*authValue* or *authPolicy*) for a PCR may apply to a set of PCR. That is, several PCR may be designated as using the same authorization value so that changing the authorization value (*authValue* or *authPolicy*) of any PCR in the set will change the value for all PCR in the set. A set of PCR that are authorized by an *authValue* are in an *authorization set*. A set of PCR that are authorized by an *authPolicy* are in a *policy set*.

The type of authorization associated with each PCR is fixed by a platform-specific specification. For each set, the platform-specific specification defines the PCRs that are in the set. A PCR should not be in more than one policy set or one authorization set.

A PCR may be in both a policy set and an authorization set. If it is in both, the only way to use the *authValue* of the authorization set is with a policy that contains TPM2_PolicyAuthValue() or TPM2_PolicyPassword().

An indication of the PCR in an authorization set may be read using TPM2_GetCapability(*capability* == TPM_CAP_PCR_PROPERTIES, *property* == TPM_PT_PCR_AUTH) and the PCR in a policy set may be read using TPM2_GetCapability(*capability* == TPM_CAP_PCR_PROPERTIES, *property* == TPM_PT_PCR_POLICY).

NOTE 1 The reference implementation only provides support for one set of each type. If additional sets are needed, the property types for TPM_CAP_PCR_PROPERTIES may be extended.

NOTE 2 If a PCR is in multiple policy or authorization sets, the TPM will use the policy or authorization of the lowest numbered set. That is, the set with the lowest TPM_PT_PCR_POLICY or TPM_PT_PCR_AUTH property.

To authorize a PCR, the correct authorization type is required, which will depend on the authorization set of a PCR. In all cases, The EmptyAuth value may be provided in either an HMAC session using a zero-length *authValue* in the HMAC calculation or as a zero length password.

17.7.1 PCR Not in a Set

If the PCR is in no set, then the authorization may only be with an EmptyAuth value.

17.7.2 Authorization Set

If the PCR is in an *authorization set,* then the *authValue* of the PCR is provided either with an HMAC session or in a password. When a PCR has a fixed, EmptyAuth value, an authorization session is still required.

 Copyright © TCG 2006-2020

When a PCR has a variable *authValue*, that *authValue* is reset to an EmptyAuth on each STARTUP(CLEAR). It is preserved across STARTUP(STATE). A variable *authValue* may be changed using TPM2_PCR_SetAuthValue() by an entity with knowledge of the *authValue*.

17.7.3 Policy Set

An *authPolicy* for a policy set has both a hash algorithm and a digest value.

If the hash algorithm for the *authPolicy* is TPM_ALG_NULL, the policy has not been set. This *uninitialized policy set* will use an EmptyAuth.

If the digest algorithm for the policy is not TPM_ALG_NULL, then the policy set is an *initialized policy set*. If the PCR is in an initialized policy set, then the authorization may only be given with a policy session.

The hash algorithm for all policy sets is set to TPM_ALG_NULL by TPM2_ChangePPS(). The algorithm and *authPolicy* associated with a PCR may only be changed using TPM2_SetAuthPolicy() by an entity with knowledge of the Platform Authorization.

If an HMAC session or a password is used for a PCR in an initialized policy set, then the TPM will return an error (TPM_RC_AUTH_TYPE). If a policy session is used for a PCR that is not in an initialized policy set, then the TPM will return an error (TPM_RC_POLICY_FAIL). Neither of these two failures would cause an update of the dictionary attack protection.

17.7.4 Order of Checking

When determining the correct type of authorization for a PCR, the TPM will use the authorization type. If the authorization is a password or HMAC session, The TPM will check to see if the PCR is in an authorization set.

17.8 PCR Allocation

A TPM may support reallocation of the PCR by the platform. To change the allocation of PCR, the platform would use TPM2_PCR_Allocate(). The allocation structure has a PCR selection for each implemented hash algorithm. To allocate a PCR in a bank, the corresponding bit would be SET in the selection for that bank.

The TPM2_PCR_Allocate() changes to PCR allocation take effect upon the next _TPM_Init and persist until the next TPM2_PCR_Allocate().

NOTE 1 Because of RAM limitations, an implementation may not allow arbitrary allocation of PCR within a bank. This does not create a deployment issue as the platform is expected to be able to manage the TPMs that would be attached to that platform.

An allocation may not be made for PCR if the attributes for the PCR are not defined by the platform-specific specification of that TPM.

NOTE 2 The attributes for a PCR include the Startup() initialization value, the locality for reset, and the locality for extend.

There is a requirement that a bank exists for each hash algorithm but there is no requirement that the bank have any PCR (that is, all selection PCR selection bits for the bank may be CLEAR).

It is a valid implementation for the TPM to ship with a specific PCR allocation that is not changeable. If the TPM does not allow the changing of the allocation, it would not implement TPM2_PCR_Allocate().

 Copyright © TCG 2006-2020

17.9 PCR Change Tracking

To support the use of PCR in policy the TPM maintains a *pcrUpdateCounter.* In general, this counter is incremented each time a PCR is modified (either extended or reset). This counter is used when a policy requires that PCR have a specific value (see 19.7.7.6).

A platform-specific specification may designate that updates of selected PCR will not cause a change to *pcrUpdateCounter*.

A bitmap of the PCR that can be updated without changing *pcrUpdateCounter* can be read with TPM2_GetCapability(*capability* == TMP_CAP_PCR_PROPERTY, property == TPM_PT_PCR_NO_INCREMENT).

17.10 Other Uses for PCR

The PCR-related commands defined in this library cover common use cases: for example, logging of components during boot or a runtime-switch in the TCB. Platform-specific specifications define PCR attributes that control this behavior and describe how PCR should be used by external software.

However, PCR are designed for more generalized representation of platform state, and platform-specific specifications may define additional PCR behaviors that capture this. Generally, a platform specification may define a PCR to represent any value that is authoritatively known by the TPM or has been securely communicated to the TPM. For instance, a TPM for a “trusted lock” might define a PCR that has value of zero to indicate that a door is closed, and one to indicate that a door is open or a virtual-TPM specification might define a PCR that has a value that represents some characteristic of the virtual machine that is issuing the TPM command. This specification demands no particular behavior or value-semantics for such PCR.

NOTE A PCR can "represent" a value either by having the PCR set to that value or by having the PCR extended with the value. In the case of the "trusted lock," it is more likely that the PCR would contain either a zero or one to represent the state of the lock than that each change to the lock be extended to a PCR.

This does not mean that the platform-specific working groups are allowed to define new commands to operate on PCR.

 Copyright © TCG 2006-2020

18 TPM Command/Response Structure

18.1 Introduction

A command is a TPM Protected Capability that indicates an operation to be performed by the TPM. It contains from one to five components, in the following order:

1) a command header that indicates the overall size of the command, the command code, and a tag indicating whether the Authorization Area is present;
2) a command-dependent number (zero to three) of handles identifying the Shielded Locations with/on which the command (Protected Capability) operates;
3) a 32-bit value indicating the size of the Authorization Area;
4) an Authorization Area containing one to three session structures; and

NOTE Components 3 and 4 always occur together. The authorization size parameter is not present if there are no sessions in the Authorization Area.

5) a command-dependent parameter area containing qualifying information for the command.

A response contains

1) a response header that indicates the overall size of the response, the response code, and a tag indicating whether the Authorization Area is present;
2) a command-dependent number (zero or one) of handles identifying the Shielded Locations with/on which the command (Protected Capability) operates;
3) a 32-bit value indicating the size of the parameter area;
4) a command-dependent parameter area containing the values produced by the TPM; and
5) an Authorization Area containing one to three session structures.

NOTE Components 3 and 5 always occur together. That is, if the Authorization Area is empty, the 32-bit value for the parameter size will not be present.

As with the command, the formats for the remaining areas of the response are dependent on the value of the associated command code. The session and parameter area order are reversed in a response.

The ordering of authorization structures and command-dependent parameters is intended to minimize TPM complexity. In a command, the authorization structures are first in order that the TPM can generate its authorization digests from the command-dependent parameters as they arrive. In a response, command-dependent parameters are first in order that the TPM can use the output buffer to assemble the command-dependent parameters prior to generating its authorization digests.

NOTE: In traditional implementations, all of the octets of a command are available at the same time so skipping around in the data structure was not an issue. In some anticipated implementations, this will not be the case and the processing of a command or response will need to be more linear.

For tables in this specification, the separators indicating the demarcations between the header, handle, authorization, and parameter components are shown in Table 4.

 Copyright © TCG 2006-2020

Table 4 — Separators

Separator	Meaning
	This type of separator is followed by one or more handles.
	In a command, this type of separator is followed by a 32-bit value indicating the number of octets in the *Authorization Area*. In a response, it is followed by a 32-bit value indicating the number of *parameter* octets (present only if tag for command/response is TPM_ST_SESSIONS).
	This type of separator is followed by one or more session structures (present only if tag for command/response is TPM_ST_SESSIONS).
	This type of separator is followed by one or more parameters

Figure 10 and Figure 11 show the basic layout of a TPM command and response (see 18.9 for a detailed example command and 18.10 for a detailed example response).

header
{ handles }
Number of handles is command-dependent and may be zero
{ authorizationSize }
{ Authorization Area }
One or more sessions
{ parameters }
Contents are commandCode-dependent

Figure 10 — Command Structure

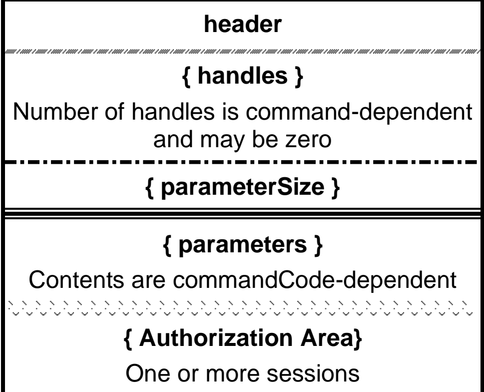


Figure 11 — Response Structure

NOTE Not all sessions in the Authorization Area are required to be used for authorization. Sessions may also be used for audit or parameter encryption.

18.2 Command/Response Header Fields

A command or response header always contains three values, displayed in Figure 12.

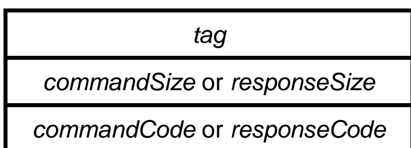


Figure 12 — Command/Response Header Structure

 Copyright © TCG 2006-2020

18.2.1 tag

A *tag* is present in all commands sent to the TPM and in responses received from the TPM. The *tag* indicates whether a command is formatted according to TPM 1.2 or this 2.0 specification. If the latter, the *tag* indicates if any session data is present.

Table 5 lists the *tag* values used for commands and response defined in this specification.

NOTE The tags for commands defined in this specification indicate only whether the command uses one or more sessions, and do not indicate the number of sessions present in the Authorization Area. Each session structure that uses a variable session handle follows the same format, which may be parsed to find the start of the next session.

Table 5 — *Tag* Values

Value	Description
TPM_ST_NO_SESSIONS	This value indicates that the command or response is formatted according to this specification and that the Authorization Area is empty. It is used in a response if the command used this tag or if the command did not complete successfully.
TPM_ST_SESSIONS	This value indicates that the command or response is formatted according to this specification and that the Authorization Area contains one or more authorizations. It indicates that the *authorizationSize* value is present; in a response, it indicates that the *parameterSize* value is present.

18.2.2 commandSize/responseSize

The *commandSize*/*responseSize* value indicates the total number of octets of this command/response, starting with the first octet of *tag.*

18.2.3 commandCode

The *commandCode* appears only in the command to the TPM. It indicates the operation that the TPM should perform and the formats of the handle and parameter areas for the command and response. The *commandCode* parameter is included in the command parameter hash (*cpHash*) and the response parameter hash (*rpHash*).

18.2.4 responseCode

The *responseCode* appears only in the response from the TPM. A *responseCode* of TPM_RC_SUCCESS (zero) indicates that the TPM has successfully completed the command and, depending on the command format, that the handle, parameter, and authorization components are present.

A non-zero *responseCode* indicates an error or fault. In this case, *tag* will be TPM_ST_NO_SESSIONS, and *responseSize* is 10, indicating that no octets follow the *responseCode*. No handle, parameter, or session response components are present.

18.3 Handles

Handles are TPM-assigned values that let the caller indicate the TPM-resident structure that a command is to manipulate. That is, the handle identifies the Shielded Location with/on which a Protected Capability is to operate. Some TPM commands (such as, TPM2_Startup()) require no handles.

 Copyright © TCG 2006-2020

The number of handles in the command and in the response is implied by the *commandCode*. It also indicates the command handles that have an associated authorization session. Handles that require authorization in an associated authorization session are listed ahead of handles that do not have an associated authorization session.

EXAMPLE TPM2_ObjectChangeAuth() has two handles, one (*objectHandle*) that uses an authorization session, and one (*parentHandle*) that does not. The standard command syntax requires that *objectHandle* occur first.

A response may have handles only if the *responseCode* is TPM_RC_SUCCESS.

The architectural limit for the number of handles in the handle area is seven. This limit is determined by the error-reporting scheme.

NOTE No currently defined command uses more than three handles.

18.4 Parameters

The *commandCode* indicates the structure of the optional handle and parameter areas. The contents of these parameter areas differ for commands and responses. Some TPM commands (such as, TPM2_Clear()) require no parameters.

All parameter values and the *commandCode* are included in the *cpHash* or *rpHash*. *authorizationSize* is not included in the *cpHash*, and *parameterSize* is not included in the *rpHash*.

NOTE 1 If a parameter is encrypted, it is included in the *cpHash/rpHash* after encryption. Because audit also uses *cpHash* and *rpHash*, audit of an encrypted session, although valid, is unlikely to be useful at the application level.

A response may have parameters only if the *responseCode* is TPM_RC_SUCCESS.

The architectural limit for the number of parameters in the handle area is 15. This limit is determined by the error-reporting scheme.

NOTE 2 This is the limit of parameters in the parameter list, not the number of values that may be in the parameter area. If a command needs more than 15 parameters, a new structure may be defined that encapsulates two or more of those parameters into a single structure, which may then be unmarshaled as a unit. The only loss is that error reporting may not provide as much detail when a compound parameter has an error.

As described in clause 21, for a command or response parameter to be encrypted, it must be the first parameter and it must be a TPM2B type.

NOTE 3 In order to encrypt more than one parameter, they must be encapsulated in a TPM2B making them a single parameter.

EXAMPLE The TPM2B_SENSITIVE_CREATE is the first parameter to TPM2_CreatePrimary(). The data member, TPMS_SENSITIVE_CREATE, has two members, a TPM2B_AUTH and a TPM2B_SENSITIVE_DATA. The encapsulation of them in the TPM2B_SENSITIVE_CREATE permits both to be encrypted.

18.5 *authorizationSize*/*parameterSize*

These values are only present if the tag of the command/response is TPM_ST_SESSIONS.

In a command, the *authorizationSize* indicates the number of octets in all of the authorization structures in the Authorization Area of the command. *authorizationSize* does not include the four octets of the *authorizationSize* value. The minimum value for *authorizationSize* is 9.

 Copyright © TCG 2006-2020

NOTE 1 The maximum value depends on the size of the largest digest produced by any hash implemented on the TPM.

NOTE 2 The driver and the TPM use the *authorizationSize* field to determine the number of authorizations. After *authorizationSize* bytes have been processed, there are no more authorizations.

In a response, *parameterSize* indicates the number of octets in the parameter area of the response and does not include the four octets of the *parameterSize* value. *parameterSize* may have a value of zero.

authorizationSize is not included in *cpHash*, and *parameterSize* is not included in the *rpHash*.

18.6 Authorization Area

18.6.1 Introduction

The Authorization Area is present in a command only if *tag* for the command is TPM_ST_SESSIONS. If present, the Authorization Area will contain:

- zero, one, or two authorizations (session or password);
- an optional session used for decrypting data sent to the TPM;
- an optional session used for encrypting data sent by the TPM; or
- an optional session used for auditing.

If *tag* is TPM_ST_SESSIONS, then the Authorization Area will have at least one but no more than three authorization/session blocks. If tag is TPM_ST_NO_SESSIONS, then there is no Authorization Area.

The number of authorization sessions that a command will have is indicated in the command schematic in TPM 2.0 Part 3. If a handle in the handle area has the "@" decoration, then an authorization session is required be present (an authorization session being either a password, a policy session, or an HMAC session).

The authorization sessions occur in the order of the associated entity handles. That is, the first handle with an "@" decoration will be associated with the first session in the Authorization Area.

Other sessions may be added to the Authorization Area. Those sessions may be designated as being for encryption, decryption, or audit; in any combination, in any order. However, in a single command, only one session is allowed to have the *encrypt* attribute, one session is allowed to have the *decrypt* attribute, and one session is allowed to have the *audit* attribute.

A single session may be used for authorization, encryption, decryption, and audit at the same time. That is, if a session has one handle with the "@" decoration, the associated authorization session may have the *encrypt*, *decrypt*, and *audit* attributes all set. A password authorization may not be used for anything but authorization and the TPM will return an error (TPM_RC_ATTRIBUTES) if *encrypt*, *decrypt*, or *audit* is SET in a password authorization.

NOTE 1 If an authorization session has *encrypt*, *decrypt*, and *audit* all SET, then the command can only have one authorization session.

The combinations of attributes allowed for each session are summarized in Table 6.

Table 6 — Use of Authorization/Session Blocks

Position	password authorization(1)(6)	authorization session(2)(6)	encryption session(3)	decryption session(4)	audit session(5)
1	✓	✓	✓	✓	✓
2	✓	✓	✓	✓	✓
3			✓	✓	✓

NOTES:

1) a password authorization may not be used for encryption, decryption, or audit.
2) an HMAC authorization session may also be used for encryption, decryption, and audit and a policy authorization session may also be used for encryption and decryption
3) only one session may be designated as being used for encryption
4) only one session may be designated as being used for decryption
5) password authorization sessions and policy sessions may not be used for audit
6) authorization sessions come before sessions used only for encryption, decryption, or audit

In TPM 2.0 Part 3, the schematic for each command will indicate if it has handles and if use of those handles requires authorizations. If there is an *at* symbol ("@") character in front of the handle name, then use of the TPM resource associated with the handle requires authorization and an authorization (session or password) will be present. An authorization will be present for each TPM resource that requires authorization (each handle with an "@"). An additional indication that a handle requires authorization is that, in the "Description" column of the command schematic, each handle has an "Auth Index:" entry. If that entry says "None", then no authorization is required. If that entry is followed by a number, then the number indicates the order of the associated authorization in the list of authorizations.

NOTE 2 Currently, no command requires more than two authorizations.

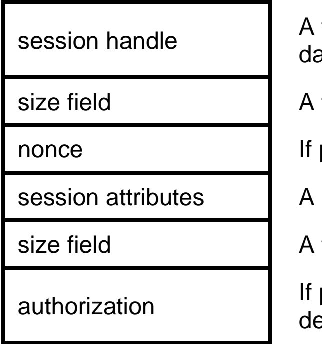
If a command requires authorizations, then those authorizations will be first in the list of authorizations/sessions. They may then be followed by other sessions used for encryption, decryption, or audit.

If the *responseCode* is TPM_RC_SUCCESS, the response has the same number of sessions in the same order as the request. Otherwise, no authorization or audit sessions are present.

18.6.2 Authorization Structure

18.6.2.1 Command

In a command, each authorization structure has the format shown in Figure 13.



A four-octet value indicating the session handle associated with this data block (will be TPM_RS_PW for a password authorization)

A two-octet value indicating the number of octets in *nonce*

If present, an octet array that contains a number chosen by the caller

A single octet with bit fields that indicate session usage

A two-octet value indicating the number of octets in *authorization*

If present, an octet array that contains either an HMAC or a password, depending on the session type

Figure 13 — Authorization Layout for Command

18.6.2.2 Response

In a response, each session structure has the format shown in Figure 14.

size field	A two-octet value indicating the number of octets in *nonce* (will be zero for a password authorization)
nonce	If present, an octet array that contains a number chosen by the TPM
session attributes	A single octet with bit fields that indicate session usage
size field	A two-octet value indicating the number of octets in *acknowledgment*
acknowledgment	If present, an octet array that contains an HMAC

Figure 14 — Authorization Layout for Response

Clause 19.6.7 describes the methods for creating an authorization session.

18.6.3 Session Handles

Session handles are described in 15.4. They identify the session being referenced by a specific session structure.

For a given command, the handle associated with a specific HMAC or policy session may occur only once in the Authorization Area. The handle representing a password authorization (TPM_RS_PW) can occur multiple times.

18.6.4 Session Attributes (*sessionAttributes*)

Each session has a *sessionAttributes* octet to indicate how the session is to be applied. Table 7 explains the meaning of the fields in this octet.

If a session is not being used for authorization, at least one of decrypt, encrypt, or audit must be SET.

Table 7 — Description of *sessionAttributes*

Attribute	Meaning
continueSession	This attribute is used to indicate to the TPM if the session is to remain 'active' when the command completes. If this attribute is CLEAR in the command and the command completes successfully (TPM_RC_SUCCESS), then the session will be flushed from TPM memory and the associated session handle will be available to be assigned to new sessions. When the TPM responds, it will echo this attribute to indicate that the session remains open (see the exception for password authorization below). NOTE In this context, "echo" means that the value of a session attribute will be the same in the response as it was in the command. The primary purpose of this attribute is to eliminate having to do explicit flushes (TPM2_FlushContext()) of a session when it is no longer used. Having this bit CLEAR on the last use of the session will end it and reclaim the TPM resources assigned to this session. For a password authorization, this attribute has no effect, as there are no TPM resources associated with a password authorization. This attribute will always be SET in a response associated with a password authorization. If the audit attribute is SET, then this attribute should also be SET since the audit data will be lost if the session is flushed.

 Copyright © TCG 2006-2020

Attribute	Meaning
decrypt	This attribute is used to indicate to the TPM that the secrets associated with the session are to be used to decrypt the first parameter of the command (the session-based encryption scheme is defined in clause 21). The parameter will be decrypted after the HMAC computations are successfully completed. This attribute may only be SET in a command that has a sized buffer as its first parameter. This attribute is required to be CLEAR in a password session. If SET in a password session, then the TPM will return an error because there is no session key for the decrypt operation This attribute is echoed by the TPM in the corresponding session in the response This attribute may only be SET in one session per command. A session with this attribute does not need to be associated with an entity identified in the handle area. That is, the session may be added just for using the session's secret for parameter decryption. This attribute can be SET in combination with any other session attribute.
encrypt	This attribute is used to indicate to the TPM that the secrets associated with the session are to be used to encrypt the first parameter of the response (the session-based encryption scheme is defined in clause 21). The parameter will be encrypted before the TPM performs the HMAC computations for any of the sessions. This attribute may only be SET in a response that has a sized buffer as its first parameter. This attribute is required to be CLEAR in a password session. If SET in a password session, then the TPM will return an error because there is no session key for the encrypt operation. This attribute is echoed by the TPM in the corresponding session in the response. This attribute may only be SET in one session per command. A session with this attribute does not need to be associated with an entity identified in the handle area. That is, the session may be added just for using the session's secret for parameter decryption. This attribute can be SET in combination with any other session attribute.
audit	This attribute indicates that the session is being used for audit. A digest is maintained in the session context and is updated each time the session is used with a command and *audit* is SET. This attribute does not need to be SET in every use of the session but the TPM will only update the audit data when the session is used with this attribute SET. This attribute has no meaning for a password authorization and is required to be CLEAR. This attribute is not allowed to be SET in a policy or trial policy session. This is because the context of the policy session would have to increase in order to hold the additional audit digest. This is significant overhead and, rather than require the additional memory in policy sessions, use of audit is restricted to HMAC sessions. After an HMAC session is started (TPM2_StartAuthSession(*sessionType* = TPM_SE_HMAC), this attribute may be set in any subsequent use of the session. On the first use of the session with this attribute set, the TPM will initialize the audit digest to 0…0 and then extend the concatenation of cpHash for the command and rpHash for the response. This attribute will be echoed by the TPM in the response. This attribute may be used in combination with any other session attributes but only one session in each command may have this attribute SET.

Attribute	Meaning
auditExclusive	This attribute is used to restrict use of an audit session. When this attribute is SET, the TPM will validate that the session has been used for all auditable commands since the audit sequence was started. NOTE An audit sequence is started when the audit digest is reset to 0…0. The audit digest is set to 0…0 when the session is first used as an audit session and when the audit digest is reset (see the description of the *auditReset* attribute below). If the session was used for all auditable commands, then it is said to be "exclusive"(see 20.2 for explanation of exclusive audit sessions). If this attribute is SET and the session is exclusive, then the command will execute. Otherwise, the TPM will fail this command to indicate to the caller that some TPM actions were not included in the audit sequence. Evaluation of the exclusive status is done at the start of the command. A session does not obtain the exclusive status until the end of the command (this prevents a session from becoming exclusive if the command fails). The implication of this processing is that, if this attribute is SET in the command that starts the audit sequence, the command will fail because the session has not yet become exclusive. In a response, this attribute will be SET if the session has exclusive status. When a session is first used as an audit session this attribute will be SET in the response as no command has executed without this session since the start of the sequence. This attribute may only be SET when the *audit* attribute is SET which excludes this attribute from being SET on a password authorization or a policy session.
auditReset	This attribute allows the caller to restart an audit sequence with a session that has previously been used for audit. If the associated command completes successfully, the TPM will initialize the session audit hash with 0…0 before Extending the cpHash and the rpHash. The response will have the exclusive attribute SET. This attribute may only be SET if audit is SET. The TPM will echo this attribute in the response.

18.7 Command Parameter Hash (*cpHash*)

The command parameter hash (*cpHash*) is used in the computation of a command authorization HMAC and is included in the digests of session and command audits (depending on the policy, the *cpHash* may also be used in the authorization). The *cpHash* is computed from the parameters of the command as follows:

$$cpHash \coloneqq \mathbf{H}_{sessionAlg}(commandCode \; \{|| \; Name1 \; \{|| \; Name2 \; \{|| \; Name3 \;\}\}\} \; \{|| \; parameters \;\}) \qquad (15)$$

where

$\mathbf{H}_{sessionAlg}$ hash function using the algorithm selected for the session when it was initialized

$commandCode$ command code for the command

$Name1$ unique identity of the entity associated with the first handle

$Name2$ unique identity of the entity associated with the second handle

$Name3$ unique identity of the entity associated with the third handle

 Copyright © TCG 2006-2020

$parameters$	remaining command parameters

18.8 Response Parameter Hash (*rpHash*)

The response parameter hash is used in the computation of a response acknowledgment HMAC and is included in the digest of session and command audits. The *rpHash* is computed from the parameters of the response as follows:

$$rpHash \coloneqq \mathbf{H}_{sessionAlg}(responseCode \mid\mid commandCode \{\mid\mid parameters\}) \quad (16)$$

where

$\mathbf{H}_{sessionAlg}$	hash function using the algorithm selected for the session when it was initialized
$responseCode$	command result code
$commandCode$	the commandCode from the command
$parameters$	response parameters

The contents of the *handles* area of the response are not included in the *rpHash*.

NOTE An *rpHash* needs to be computed only when the *responseCode* is TPM_SUCCESS, which means that it is redundant to include the response code. It is retained for legacy reasons.

18.9 Command Example

Table 8 shows an example of a command schematic used in this specification. The command has two object handles (*handleA* and *handleB*). The "@" on the handleA name indicates that use of the entity associated with the handle requires authorization. The command has at least one session to authorize use of *handleA*. It will not have a session for use of *handleB*. The Authorization Area may have an additional audit session and a session used only for parameter encryption. Since one session is required, *tag* is TPM_ST_SESSIONS, and the *authorizationSize* field is present.

Although they are not shown in the command schematic, the *authorizationSize* value and the Authorization Area would be present in the command buffer and be located between *handleB* and *dataSize*.

NOTE: The Authorization Area is not shown with the command schematic because no single representation is possible.

The command and response tables have three columns.

1) **Type** – This column indicates the data type of the parameter passed to the TPM in a command or received from the TPM in a response.
2) **Name** – This column indicates the name of the parameter. This name is referenced in the description of the command that precedes the command table and in the detailed actions of the command that follows the response table.

 Copyright © TCG 2006-2020

3) **Description** – This column provides a limited description of the parameter and indicates the possible options for the command.

EXAMPLE 1

Table 8 — Command Layout for Example Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Example
TPM_HANDLE	@handleA	handle to use for one object of the command Auth Index: 1 Auth Role: USER
TPM_HANDLE	handleB	handle to use for the second object Auth Index: None
UINT32	dataSize	example data size
OCTET	data[dataSize]	example data

Table 9 illustrates all command octets for the command in Table 8. In this example, the nonce size is 20 octets and the authorization HMAC is computed using SHA256. The values in shaded cells are not shown in the TPM 2.0 Part 3 schematic of the command but are included in the command data sent to the TPM.

EXAMPLE 2

Table 9 — Example Command Showing *authorizationSize*

Offset	Size	Parameter	Value
0	2	tag	TPM_ST_SESSIONS
2	4	commandSize	**209** < size in octets of the command >
6	4	commandCode	TPM_CC_Example
10	4	handleA	< a valid TPM resource handle>
14	4	handleB	< a valid TPM resource handle>
18	4	authorizationSize	**61** < size of the authorization session >
22	4	authHandle	< a valid TPMI_SH_AUTH_SESSION >
26	2	nonceCallerSize	**20** < size of *nonce* >
28	20	nonceCaller	< a 20-octet random value>
48	1	sessionAttributes	(continueSession=1)
49	2	hmacSize	**32** <size of *HMAC*>
51	32	HMAC	< a 32-octet HMAC value based on SHA256>
83	2	dataSize	**32** < size of the buffer >
85	124	data[dataSize]	< 124 octet buffer >
209			

 Copyright © TCG 2006-2020

18.10 Response Example

Table 10 shows an example schematic as it would appear in TPM 2.0 Part 3. The example is for a response sent from the TPM after successful completion of the example command in Table 8. The response has the same number of sessions in the same order as did the command.

EXAMPLE 1

Table 10 — Response Layout for Example Command

Type	Name	Description
TPM_ST	tag	TPM_ST_SESSIONS
UINT32	responseSize	
TPM_RC	responseCode	response code of the operation
TPM_HANDLE	handle	not included in the rpHash
UINT32	dataSize	size in octets of the following data
OCTET	data[dataSize]	a returned block of information

Table 11 illustrates the full response for the command in Table 8. As in the command, the nonce size is 20 octets and the acknowledgment HMAC is computed using SHA256. The values in shaded cells are not shown in the TPM 2.0 Part 3 schematic of the response but are present in the response data from the TPM.

EXAMPLE 2

Table 11 — Example Response Showing *parameterSize*

Offset	Size	Parameter	Value
0	2	tag	TPM_ST_SESSIONS
2	4	responseSize	**203** < size in octets of the response >
6	4	responseCode	**0** < success >
10	4	handle	< a valid TPM_HANDLE >
14	4	parameterSize	**128**
18	4	dataSize	**124**
22	124	data[dataSize]	< 124 octet buffer >
146	2	nonceTpmSize	**20**
148	20	nonceTPM	< a 20-octet random value >
168	1	sessionAttributes	(continueSession=1)
169	2	hmacSize	**32**
171	32	HMAC	< a 32-octet HMAC value based on SHA256 >
203			

 Copyright © TCG 2006-2020

19 Authorizations and Acknowledgments

19.1 Introduction

Many commands to the TPM reference TPM-resident structures and use of these structures may require authorization. This authorization is provided in structured data that follows the command data. When an authorization is provided to a TPM, the TPM will provide an acknowledgment.

To provide flexibility in how the authorizations are given to the TPM, this specification defines three authorization types:

1) password;

2) HMAC; and

3) policy.

Depending on the command, zero, one, or two authorizations may be required. In a command, the authorizations follow the handles, and in a response, the authorization replies follow the response parameters. The command definition indicates how many authorizations are required.

19.2 Authorization Roles

For each object and NV Index, there is a set of operations that can be performed on or with that object or NV Index. The operations are divided into groups, based on the impact of the operation on the object. To perform an operation with or on an object in a group, the authorization specific to that group must be provided. When performing an operation in one of the groups, the caller is acting in a specific role with respect to that object.

The TPM supports three different authorization roles. The role and attributes determine whether a password or HMAC can be used for authorization. A policy (if not the Empty Policy) can always be used.

1) **USER** – this authorization role is used for the normal uses of a key (e.g., signing with a signing key, or loading the child of a Storage Key). Methods are defined to allow USER role authorization to be provided either with an authorization value (*authValue*) or a policy. If *userWithAuth* is SET, then USER role authorization may be provided with a password authorization or an HMAC session. If *userWithAuth* is CLEAR, then a password and HMAC authorizations may not be used to provide USER role authorizations. A policy session that satisfies the *authPolicy* of the entity may be used regardless of the setting of *userWithAuth*.

NOTE 1 For USER role, an *authPolicy* is satisfied when the *policyDigest* of a policy session matches the value of the *authPolicy* value of the object.

NOTE 2 If use of an object is to be gated based on PCR values, a policy session is required (see 19.7). If the intent is that different Users have access to the object but only if the PCR are correct, then it is likely that authorization with the *authValue* will be disabled; otherwise, the caller could circumvent PCR protections simply by providing the *authValue*.

2) **ADMIN** – the object Administrator controls the certification of an object (TPM2_Certify() and TPM2_ActivateCredential()) and controls changing of the *authValue* of an object (TPM2_ObjectChangeAuth()). When an action requires ADMIN role authorization, that authorization may be provided using the *authValue* of the object if the *adminWithPolicy* attribute of the object is CLEAR. As with USER role authorization, ADMIN role may always be provided with a policy session as long as the policy session satisfies the *authPolicy* of the object.

NOTE 3 For ADMIN role, an *authPolicy* is satisfied when *policySession→policyDigest* matches the value of the *authPolicy* value of the object and *policySession→commandCode* matches *commandCode* for the authorized command.

EXAMPLE If the *adminWithPolicy* attribute of an object is SET, and if no branch in the object's policy equation contains TPM2_PolicyCommandCode(TPM_CC_Certify), then certification of that key may not occur.

3) **DUP** – this authorization role is only used for TPM2_Duplicate(). If duplication is allowed, authorization must always be provided by a policy session and the *authPolicy* equation of the object must contain a command that sets the policy command code to TPM_CC_Duplicate.

19.3 Physical Presence Authorization

Authorization for some commands requires that it be provided with Platform Authorization. Authorization for some other commands allows use of either Platform Authorization or Owner Authorization (Most of these commands cause persistent state change of the TPM). For these commands, it is possible to require that authorization be augmented with an out-of-band method.

For commands that require Platform Authorization and commands that require a hierarchy authorization, it is possible to require an out-of-band authorization. This may take any number of forms, such as a dedicated pin in the TPM, a special signaling method through the TPM interface, or any desired alternative. Whatever the form, the out-of-band authorization is referred to in this specification as Physical Presence (PP). This does not mean that the signaling requires a human to be physically present in order for the indication to be provided. The term is used in this specification because it was used in previous TPM specifications to refer to a similar concept.

The TPM maintains a table of the commands that require that PP be asserted to authorize command execution. Only certain commands may be included in this table. If, in TPM 2.0 Part 3, the schematic for a command has TPM_RH_PLATFORM in the "Description" column for one of the handles, then that command can be added to the list of commands that require PP. Otherwise, it may not.

NOTE 1 In the "Description" column, TPM_RH_PLATFORM will be followed by +PP if assertion of Physical Presence is required or "+{PP}" to indicate that assertion of Physical Presence may be required if indicated by the table.

NOTE 2 A platform-specific specification may require that the table be initialized in a specific way. It could even require that the table have certain commands defined to require PP confirmation even though a PP interface is not provided on the TPM. This would serve to disable the use of that command by the platform.

When the authorization handle is TPM_RH_PLATFORM, the TPM checks the table to see if the command requires confirmation with PP. If so, PP is checked before the TPM performs any other authorization checks.

TPM2_PP_Commands() is used to change the contents of the table of commands that require confirmation with PP authorization. Authorization of the command TPM2_PP_Commands() requires that PP be asserted and TPM2_PP_Commands() may not be removed from the list of commands that require PP.

NOTE 3 This constraint on TPM2_PP_Commands() prevents setting or modification of the table if no PP interface exists on the TPM.

The contents of the table may be read using TPM2_GetCapability(*capability* == TPM_CAP_PP_COMMANDS).

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

19.4 Password Authorizations

A plaintext password value may be used to authorize an action when use of an *authValue* is allowed. A plaintext password may be appropriate for cases in which the path between the caller and the TPM is trusted or when the authorization value is well known. For these instances, encryption of parameters or the hiding of authorization values in an HMAC is not required.

NOTE 1 While it may seem relatively easy for a caller to perform an HMAC, there are situations where the caller is resource-constrained and unable to do so. This is especially true when the calling software does not support the hash algorithms implemented in the TPM. Additionally, authentication using a cryptographic protocol makes it difficult to provide operating system abstractions.

A reserved authorization handle (TPM_RS_PW) indicates that the authorization is a password.

TPM_RS_PW is always available, and a separate action to create an authorization session is not required. A password authorization does not use nonces. *sessionAttributes→continueSession* is ignored.

A password authorization lets the caller send more or fewer octets than are present in the object's authorization field. The TPM truncates any octets of zero on either of the two values before they are compared.

If present, a password authorization is always associated with a command handle that requires authorization as there is no session context associated with a password that would allow it to be used for encryption or command audit.

Unlike other handles for other session types, the TPM_RS_PW session handle may be used for more than one authorization.

Password authorization data sent to the TPM has the format shown in Table 12.

Table 12 — Password Authorization of Command

Type	Name	Description
TPMI_SH_AUTH_SESSION	authHandle	required to be the reserved authorization session handle TPM_RS_PW
TPM2B_NONCE	nonceCaller	required to be an Empty Buffer
TPMA_SESSION	sessionAttributes	only *continueSession* may be SET
TPM2B_AUTH	password	authorization compared to the *authValue* of the TPM entity

Table 13 illustrates the format of a password authorization in a response. This structure is provided to ensure a one-to-one correspondence between the sessions in the command and in the response.

Table 13 — Password Acknowledgment in Response

Type	Name	Description
TPM2B_NONCE	nonceTPM	zero-length for a password authorization
TPMA_SESSION	sessionAttributes	copy of the flags from the password authorization in the command, *continueSession* will be SET
TPM2B_AUTH	hmac	zero-length buffer for a password authorization

NOTE 2 This structure is used to provide symmetry between password and other response sessions.

 Copyright © TCG 2006-2020

19.5 Sessions

A session is a collection of TPM state that changes after each use of that session. When an object context is loaded into the TPM, multiple copies of the object context may exist both on the TPM and in saved contexts (see clause 30). When a session context is created, only one copy of that context may exist, either on the TPM or as a saved context. The context of a session changes on each use.

A session has a handle that is assigned by the TPM when the session is created. That handle will always refer to the same session until the session is closed. If a handle is re-assigned to a subsequently created session, the session context data will contain a TPM-generated nonce that makes the new instance of the session unique, even though the handle may have been used previously. This nonce will change each time the session is used so that previous instances of the same session can be distinguished from each other (i.e., the nonce prevents reuse of stale session contexts).

There are three uses of a session:

1) **authorization** – A session associated with a handle is used to authorize use of an object associated with a handle. If it is not a password authorization, it may also be used to provide keys for encryption of command or response parameters. A policy session used to authorize may not also be used as an audit session. An HMAC session used to authorize may be used as an audit session.
2) **audit** – An audit session collects a digest of command/response parameters to provide proof that a certain sequence of events occurred. An audit session may also be used to provide secrets for encryption of command or response parameters and may be used for authorization of an HMAC session.
3) **encryption** – A session that is not used for authorization or audit may be present for the purpose of encrypting command or response parameters. If an encryption-only session exists, it will follow the authorization sessions and may come before or after a session used only for audit.

A command may have as many as three authorization blocks. Password blocks may only be used for authorization, so the maximum number of password blocks is equal to the number of authorizations required by the command.

19.6 Session-Based Authorizations

19.6.1 Introduction

Session-based authorizations are used both for protocols that require confidentiality for the authorization value and for audit sessions that require tracking of a sequence of commands sent to the TPM. An authorization session also provides a means of linking the uses of the session.

There are two types of session-based authorization: HMAC and policy. Both types of session are initiated using TPM2_StartAuthSession(). That command establishes the parameters that will be used for the authorizations. The *sessionType* parameter determines if the session will be an HMAC or policy session. When the session is started, the hash algorithm and TPM nonce size used in the session are specified by the caller. The command may include an initial caller nonce and a *salt* value to generate the session key. The parameters of each session are independent from the parameters of any other session and are limited only by the capabilities of the TPM. When TPM2_StartAuthSession() completes successfully, the TPM returns a handle for the session as well as the initial *nonceTPM* value.

Once an authorization session is established, it may be used to authorize actions in multiple commands. The session is not ended until explicitly closed or flushed.

The secret values of a session are determined by the handles used when the session is started. The command for starting a session allows selection of up to two object handles. One handle indicates a TPM

object that is used to encrypt a salt value that is sent when the session is started. A second handle indicates an object containing a shared secret. The salt value and the shared secret are combined with a nonce provided by the caller to create the session secrets.

NOTE Using the endorsement key for which the certificate chain has been validated as the salt key can ensure that the caller is connected to an authentic TPM.

19.6.2 Authorization Session Formats

For a session-based authorization session, the authorization structure for a command is as shown in Table 14.

Table 14 — Session-Based Authorization of Command

Type	Name	Description
TPMI_SH_AUTH_SESSION	authHandle	the handle for the authorization session
TPM2B_NONCE	nonceCaller	the caller-provided session nonce; size may be zero
TPMA_SESSION	sessionAttributes	the flags associated with the session
TPM2B_AUTH	hmac	the session HMAC digest value

In a response, the format for the acknowledgement is as shown in Table 15.

Table 15 — Session-Based Acknowledgment in Response

Type	Name	Description
TPM2B_NONCE	nonceTPM	the TPM-provided session nonce. Size is as specified when the session was started.
TPMA_SESSION	sessionAttributes	the flags associated with the session. This attribute should be the same as the values in the command except *continueSession* may be CLEAR.
TPM2B_AUTH	hmac	the session HMAC digest value

19.6.3 Session Nonces

19.6.3.1 Overview

The primary use of a nonce in a session is to prevent an authorization from being reused. When the session is started by TPM2_StartAuthSession(), the caller indicates, among other things, the size of the nonces to be used in the authorization HMAC and an initial nonce value (*nonceCaller*). After establishing the session, the TPM returns a handle to identify the session and a TPM-generated random nonce (*nonceTPM*). The TPM stores this *nonceTPM* in the context of the session.

Each time the session is used for authorization, the caller performs an HMAC using, along with other parameters, the last *nonceTPM* for the session and a new *nonceCaller* for the session. The TPM then uses the received *nonceCaller* and the saved *nonceTPM* to validate the HMAC. For a response, the TPM uses the last *nonceCaller* and a newly generated *nonceTPM* in the HMAC. The caller then uses the received *nonceTPM* and the saved *nonceCaller* to validate the HMAC in the response.

A nonce has a size field indicating the number of octets in the nonce followed by the nonce data. The nonce size is not included in the HMAC computation.

 Copyright © TCG 2006-2020

19.6.3.2 Session Nonce Size

When an authorization session is created, the caller provides an initial nonce (*nonceCaller*). The size field of *nonceCaller* is retained by the TPM and used to determine the size of all nonces generated by the TPM (*nonceTPM*) in the subsequent uses of the session. The minimum size for *nonceCaller* in TPM2_StartAuthSession() is 16 octets.

After the initial session setup, the caller may use any size for a *nonceCaller* in each use of the session. The *nonceCaller* size may vary from zero (0) up to the size of *nonceTPM* (the initial *nonceCaller* size).

NOTE A TPM implementation may allow larger nonce sizes but the caller should not expect a TPM to accept a nonce size larger than the initial *nonceCaller* size.

The maximum size that may be requested for *nonceTPM* is the size of the digest produced by the authorization session hash.

EXAMPLE For SHA-1 the maximum size for *nonceTPM* is 20 octets and for SHA256 it is 32 octets.

When a session nonce is used in the authorization session HMAC, the *size* field of the nonce is not included in the authorization computation. If the nonce *size* field is zero (0), then the nonce does not affect the authorization HMAC value.

19.6.3.3 Guidance on Nonce Size Selection

The size of the nonce should be chosen to provide a reasonable guarantee that a TPM-generated nonce value will not be used twice with the same *sessionKey*. The choice of nonce size is not related to the number of uses of a specific authorization session but is related to the number of uses of the *sessionKey*.

An HMAC *sessionKey* is derived from the *authValue* kept in an object and that *authValue* may have a long lifetime. To prevent replay attacks on a long-lived *authValue,* use of large nonces is recommended.

NOTE 1 The combined *nonceCaller* plus *nonceTPM* are what determine the anti-replay protection provided by the nonces. Making the combined size larger than the block size of the session hash is not particularly useful. If the caller does not have a good source of entropy for an RNG, then making the *nonceTPM* the size of the digest of the session hash is recommended, so that a *nonceCaller* size of zero would be satisfactory.

NOTE 2 When using a session for encryption, if a parameter is encrypted in a response to one command and a parameter is encrypted in the request of the next command, and they both use the same session for encryption, then the caller should provide a *nonceCaller* in order to prevent the use of the same encryption key on the input and output. A nonce of length 1 with a value of zero would suffice.

19.6.3.4 Nonce Binding

A command may have sessions other than those required for authorization. One use of an extra session is to encrypt a command or response parameter. If an extra encrypting session were removed by an attacker, the TPM would not properly encrypt/decrypt the data and could, as a result, fail to encrypt a response parameter. To prevent removal of extra encrypting sessions, the *nonceTPM* of each of these sessions is included in the HMAC computation of the first authorization session of a command. If an extra session is removed by an attacker, the first authorization will fail, and the command will not be executed.

To simplify the logic in the TPM, the *nonceTPM* of any session used for encryption of command or response data is included in the HMAC computation for the first session even if the encrypt or decrypt session is also an authorization session.

NOTE If the first session is a password authorization, then the path to the TPM is trusted and there is no need to guard against the extra session being removed, also there is probably no need for parameter encryption when a trusted path is present.

 Copyright © TCG 2006-2020

19.6.4 Authorization Values

19.6.4.1 Overview

An object may have a value used to authorize various actions on the object. An authorization session is the mechanism through which a caller proves knowledge of the authorization value (*authValue*) needed to allow an action.

An *authValue* may be sent as a password that does not provide confidentiality (see 19.4), or in an HMAC-based authorization session that can provide confidentiality of the *authValue*.

19.6.4.2 authValue Size

An *authValue* may be as small as zero octets but not larger than the digest size of the algorithm used to compute the Name of the object.

EXAMPLE If the Name algorithm for an object is SHA256, then the largest *authValue* for the object would be 32 octets.

19.6.4.3 Authorization Size Convention

When an *authValue* is based on a password or passphrase, then the *authValue* should be the password/phrase as long as the password/phrase is no larger than the digest produced by the *nameAlg* of the object.

EXAMPLE If the passphrase is “This is a sample passphrase”, and *nameAlg* is TPM_ALG_SHA256, then the *authValue* is 27 octets long containing the value “This is a sample passphrase”.

Trailing octets of zero are to be removed from any string before it is used as an *authValue*.

If the password/phrase, with trailing zeros removed, is longer than the digest produced by the *nameAlg* of the object, then the password/phrase – with trailing octets of zero removed – is hashed using *nameAlg* and the resulting hash given to the TPM as the *authValue* for the object.

19.6.5 HMAC Computation

The HMAC computation for all session types is the same. A *sessionKey* value is concatenated to an *authValue* to create the key that is used in the computation of the HMAC in a command or response. If *sesssionkey* and *authvalue* are both the Empty Buffer, see 19.6.15.

$$\begin{aligned} authHMAC \coloneqq \mathbf{HMAC}_{sessionAlg}(&(sessionKey \mathbin{||} authValue), \\ &(pHash \mathbin{||} nonceNewer \mathbin{||} nonceOlder \\ &\{ \mathbin{||} nonceTPM_{decrypt} \} \{ \mathbin{||} nonceTPM_{encrypt} \} \\ &\mathbin{||} sessionAttributes)) \end{aligned} \quad (17)$$

where

$\mathbf{HMAC}_{sessionAlg}$	the HMAC function using the hash algorithm specified when the session was started
$sessionKey$	a value that is computed in a protocol-dependent way, using **KDFa**(). When used in an HMAC or KDF, the size field for this value is not included.
$authValue$	a value that is found in the sensitive area of an entity. This value is an EmptyAuth if the HMAC is being computed to authorize an action on the

 Copyright © TCG 2006-2020

object to which the session is bound. The size field for this value is not included in any KDF or hash function.

NOTE 1 For policy sessions, the *authValue* is not included in the HMAC calculation unless the policy session included TPM2_PolicyAuthValue() and it was not superseded by TPM2_PolicyPassword().

NOTE 2 Trailing zeros are always removed from an *authValue* before it is used in an authorization computation.

$pHash$ digest of the command (cpHash) or response parameters (rpHash) using the session hash algorithm.

$nonceNewer$ a value that is generated by the entity using the session. A new nonce is generated on each use of the session. For a command, this will be nonceCaller and for a response, nonceTPM. The nonce size field is not included in the HMAC.

$nonceOlder$ a value that was received the previous time the session was used. For a command, this will be nonceTPM and for a response, nonceCaller. The nonce size field is not included in the HMAC.

$nonceTPM_{decrypt}$ in the HMAC computation for the first authorization session of a command, if a different session is being used for parameter decryption, then the nonceTPM for that session is included in the HMAC of the first authorization session; but only in the command (see 19.6.3.4). The nonce size field is not included in the HMAC.

NOTE 3 The *decrypt* session is used by the TPM to decrypt a parameter in the command.

NOTE 4 The nonce of the *decrypt* session is included even if that session is also used for authorization.

$nonceTPM_{encrypt}$ in the HMAC computation for the first authorization session of a command, if a different session is being used for parameter encryption, then the nonceTPM for that session is included in the HMAC of the first authorization session; but only in the command (see 19.6.3.4). The nonce size field is not included in the HMAC.

NOTE 5 The *encrypt* session is used by the TPM to encrypt a parameter in the response.

NOTE 6 The nonce of the *encrypt* session is included even if that session is also used for authorization.

NOTE 7 If the same session (not the first session) is used for decrypt and encrypt, its *nonceTPM* is only used once. If different sessions are used for decrypt and encrypt, both *nonceTPM*s are included.

$sessionAttributes$ an octet indicating the attributes associated with a particular use of the session

With the exception of $sessionAttributes$, all the values are large numbers, typically with sizes of 20 octets or more.

In the HMAC computation equations shown below, the possibility that the HMAC computation may include $nonceTPM_{decrypt}$ or $nonceTPM_{encrypt}$ is indicated by "$nonceOlder*$" (asterisk added).

 Copyright © TCG 2006-2020

19.6.6 Note on Use of Nonces in HMAC Computations

In equation (17), and the HMAC computation equations that follow, all of the nonce values are in TPM2B_NONCE data structures. In the HMAC computations, the nonce entries should all be read as if they had the *.buffer* suffix indicating that only the data portion of a nonce is ever used in an HMAC computation.

19.6.7 Starting an Authorization Session

TPM2_StartAuthSession() is used to start an authorization session. The parameters of this command may be chosen to produce sessions with different properties.

Table 16 — Schematic of TPM2_StartAuthSession Command

Type	Name	Description
TPM_ST	tag	
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_StartAuthSession
TPMI_DH_OBJECT+	tpmKey	handle of a loaded key used to encrypt *salt* may be TPM_RH_NULL Auth Index: None
TPMI_DH_ENTITY+	bind	entity providing the *authValue* may be TPM_RH_NULL Auth Index: None
TPM2B_NONCE	nonceCaller	initial *nonceCaller,* sets nonce size for the session
TPM_SE	sessionType	indicates the type of session (HMAC or policy)
TPM2B_ENCRYPTED_SECRET	encryptedSalt	*tpmKey* algorithm-dependent secret if *tpmKey* is TPM_RH_NULL, this shall be an Empty Buffer
TPMT_SYM_DEF+	symmetric	the algorithm and key size for parameter encryption may select TPM_ALG_NULL
TPMI_ALG_HASH	authHash	hash algorithm to use for the session; and shall be a hash algorithm implemented on the TPM and not TPM_ALG_NULL

The two values that determine the session protection values are *tpmKey* and *bind*. Both of these handles can reference TPM_RH_NULL or a TPM entity. The *tpmKey* parameter references the key that is used to encrypt a salt value that is used in the computation of the *sessionKey*. The *bind* parameter references a TPM entity that may provide an *authValue* to the computation for the *sessionKey*. The four variations for *tpmKey* and *bind* give sessions with different properties.

 Copyright © TCG 2006-2020

Table 17 — Handle Parameters for TPM2_StartAuthSession

tpmKey	bind	session properties
TPM_RH_NULL	TPM_RH_NULL	Unbound session
TPM_RH_NULL	TPM entity	Bound session
TPM key	TPM_RH_NULL	Salted session
TPM key	TPM entity	Salted and bound session

19.6.8 *sessionKey* Creation

A *sessionKey* value is used in the HMAC computation as shown in equation (17). If both *tpmKey* and *bind* are TPM_RH_NULL, then *sessionKey* is set to an Empty Buffer. Otherwise, the *sessionKey* is created as follows:

$$sessionKey \coloneqq \mathbf{KDFa}(sessionAlg, (authValue \mathbin{||} salt), \text{“ATH”}, nonceTPM, nonceCaller, bits) \qquad (18)$$

where

$sessionAlg$	a TPM_ALG_ID for a hash that was chosen by the caller when the session was started
$authValue$	if *bind* is not TPM_RH_NULL, a TPM2B_AUTH.buffer that is found in the sensitive area of a TPM entity; otherwise, an Empty Buffer
$salt$	if *tpmKey* is not TPM_RH_NULL, then the salt value recovered from *encryptedSalt*; otherwise, an Empty Buffer
“ATH”	a four-octet label value (see 4.1)
$nonceTPM$	a TPM2B_NONCE that is generated by the TPM when the session was started
$nonceCaller$	a TPM2B_NONCE that is provided by the caller when the session was started.
$bits$	the number of bits returned is the size of the digest produced by *sessionAlg*

NOTE When an authorization failure occurs, the TPM will check to see if the use of the object is exempt from dictionary attack protection. If it is exempt, the response code is changed from TPM_RC_AUTH_FAIL to TPM_RC_BAD_AUTH and no increment of the failed authorization counter occurs (see 19.8).

19.6.9 Unbound and Unsalted Session Key Generation

In this session key generation method used by TPM2_StartAuthSession(), *tpmKey* and *bind* are both TPM_RH_NULL. This results in the session having no *sessionKey* (it is an Empty Buffer). The session is not bound to any object.

NOTE This session type is similar to the OIAP session of TPM 1.2.

A session started using this format can be used for parameter encryption while executing TPM commands. However, during these commands, the key used to encrypt the parameter will only use the *authValue* of the object being accessed by the commands in the key generation, so the strength of the encryption will be no better than the entropy in the *authValue* of the object.

When computing the HMAC, the *authValue* of the referenced entity is used:

 Copyright © TCG 2006-2020

$$authHMAC \coloneqq \mathbf{HMAC}_{sessionAlg}\,(authValue_{entity}.buffer,\ (pHash \,||\, nonceNewer.buffer \,||\, nonceOlder^{*}.buffer \,||\, sessionAttributes)) \quad (19)$$

If the size of *authValue* is zero, then the caller may omit the HMAC from the authorization (see No HMAC Authorization19.6.15).

Table 18 — Format to Start Unbounded, Unsalted Session

Type	Name	Description
TPM_ST	tag	
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_StartAuthSession
TPMI_DH_OBJECT+	tpmKey	TPM_RH_NULL
TPMI_DH_ENTITY+	bind	TPM_RH_NULL
TPM2B_NONCE	nonceCaller	initial *nonceCaller,* sets *nonceTPM* size for the session
TPM2B_ENCRYPTED_SECRET	encryptedSalt	00 00_{16}
TPM_SE	sessionType	indicates the type of the session (HMAC, policy, or trial)
TPMT_SYM_DEF+	symmetric	will normally be TPM_ALG_NULL for an unbound and unsalted session
TPMI_ALG_HASH	authHash	hash algorithm to use for the session; required to be a hash algorithm implemented on the TPM and not TPM_ALG_NULL

NOTE When *sessionType* is TPM_SE_TRIAL, there is no benefit in using any other version of TPM2_StartAuthSession() as a trial session is not allowed to be used for authorization. This means that the *sessionKey* of the session will never be used so there is no point in having the TPM generate it.

19.6.10 Bound Session Key Generation

In this session key generation method used by TPM2_StartAuthSession(), *tpmKey* is TPM_RH_NULL indicating that no *salt* value is present but *bind* references some TPM entity with an *authValue*.

NOTE 1 This session type has properties that are similar to an OSAP session in TPM 1.2.

The *sessionKey* is computed using the *authValue* from *bind* and an Empty Buffer in place of the *salt* value.

$$sessionKey \coloneqq \mathbf{KDFa}\,(sessionAlg,\ authValue_{bind},\ \text{“ATH”},\ nonceTPM,\ nonceCaller,\ bits) \quad (20)$$

NOTE 2 If handle references a TPM resource that has an EmptyAuth, the *sessionKey* is still computed.

When performing an HMAC for authorization, the HMAC key is calculated as follows:

a) When the session is an HMAC session

1) If the authorization is not for the entity to which the session is bound, the HMAC key is the concatenation of the entity's *authValue* to the session's *sessionKey* (created at TPM2_StartAuthSession() (see equation (21).

 Copyright © TCG 2006-2020

2) If the authorization is for the entity to which the session is bound, the HMAC key is the session's *sessionKey* (created at TPM2_StartAuthSession() (see equation (22)).

b) When the session is a policy session

1) If the session has *isAuthValueNeeded* SET (by TPM_PolicyAuthValue()), the HMAC key is the concatenation of the entity's *authValue* to the session's *sessionKey* (see equation (21)).

2) If the session has *isAuthValueNeeded* CLEAR, the HMAC key is the session's *sessionKey* (created at TPM2_StartAuthSession() (see equation (22)).

$$authHMAC \coloneqq \mathbf{HMAC}_{sessionAlg}\,(sessionKey \,||\, authValue_{entity}),\ (pHash \,||\, nonceNewer \,||\, nonceOlder^{*} \,||\, sessionAttributes)) \tag{21}$$

$$authHMAC \coloneqq \mathbf{HMAC}_{sessionAlg}\,(sessionKey,\ (pHash \,||\, nonceNewer \,||\, nonceOlder^{*} \,||\, sessionAttributes)) \tag{22}$$

NOTE 3 Binding to an entity different from the one being authorized is a way of adding entropy to the session key. It is useful in cases where the entity being authorized has a low entropy authorization value.

The TPM is required to keep track of the entity to which the session is bound. This is nominally accomplished when the session is started by recording, in the session context, the Name of the *bind* entity. For an NV Index or persistent handle, the TPM is required to also record the authorization value associated with the entity.

NOTE 4 In the Part 4 reference implementation , the authorization value is combined with the Name and stored in the SESSION→boundEntity member.

NOTE 5 Recording of the NV Index authorization is required to prevent an attacker from "squatting" on an Index. This would be accomplished by creating an NV Index that has properties that are identical to an NV Index that is expected to be created, but with an authorization value known to the attacker. The attacker would then start an authorization session bound to the NV Index and delete the NV Index. When the NV Index to be attacked is created, the attacker would have an authorization session bound to an Index with the same Name and could access to the NV Index even though the actual authorization value is unknown.

On a command, the TPM will check to see if the authorization is being used for the entity to which it was bound. If so, then the *authValue* of the bound entity is not used in the HMAC computation. The TPM will record the fact that the *authValue* was not used in the HMAC computation of the authorization and not include it in the HMAC computation on the response.

NOTE 6 This allows the session to remain bound to an NV Index for the duration of the first command that writes to the Index even though the Name of the Index changes during the command processing. The session will not be bound to the Index when the command completes. The session can continue to be used, but it, in effect, is no longer bound because there is no longer a TPM entity with the correct Name.

For a persistent object, the authorization value is included so that authorization can be revoked. If the administrator for a persistent object changes the authorization, sessions bound to the old authorization should no longer be valid.

NOTE 7 To change the authorization of a persistent object, TPM2_ObjectChangeAuth() would be called. It would return a new sensitive area. The current persistent object would be deleted (TPM2_EvictControl()) and the object with the new authorization loaded (TPM2_Load()). Finally, the loaded object would be made persistent (TPM2_EvictControl()). It is only required that the old object be deleted if the new object is to have the same handle or if it is desired to revoke the old authorization.

 Copyright © TCG 2006-2020

The *noDA* attribute of the bind object is recorded in the session context. For a description of the rationale, see clause 19.8.7.

Table 19 — Format to Start Bound Session

Type	Name	Description
TPM_ST	tag	
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_StartAuthSession
TPMI_DH_OBJECT+	tpmKey	TPM_RH_NULL
TPMI_DH_ENTITY	bind	entity providing the *authValue* to which the session is bound and not TPM_RH_NULL
TPM2B_NONCE	nonceCaller	initial *nonceCaller,* sets *nonceTPM* size for the session
TPM2B_ENCRYPTED_SECRET	encryptedSalt	00 00_{16}
TPM_SE	sessionType	indicates the type of the session (HMAC, policy, or trial)
TPMT_SYM_DEF+	symmetric	if the session is to be used for parameter encryption, set this to an algorithm and key size
TPMI_ALG_HASH	authHash	hash algorithm to use for the session; required to be a hash algorithm implemented on the TPM and not TPM_ALG_NULL

 Copyright © TCG 2006-2020

19.6.11 Salted Session Key Generation

In this session key generation method used by TPM2_StartAuthSession(), *bind* is TPM_RH_NULL, indicating that no entity is referenced to provide an *authValue,* but *tpmKey* is present and indicates a key used to encrypt the *salt* value. The *sessionKey* is computed with an Empty Buffer in place of the *authValue*.

$$sessionKey \coloneqq \mathbf{KDFa}\ (sessionAlg, salt, \text{“ATH”}, nonceTPM, nonceCaller, bits) \tag{23}$$

Because *bind* is TPM_RH_NULL, the session is not bound to any entity. When the session is used to access any entity, the HMAC will use the *sessionKey* and the *authValue* of that entity.

$$authHMAC \coloneqq \mathbf{HMAC}_{sessionAlg}\ ((sessionKey \mid\mid authValue_{entity}), (pHash \mid\mid nonceNewer \mid\mid nonceOlder^{*} \mid\mid sessionAttributes)) \tag{24}$$

Table 20 — Format to Start Salted Session

Type	Name	Description
TPM_ST	tag	
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_StartAuthSession
TPMI_DH_OBJECT	tpmKey	handle of a loaded key used to encrypt *salt*
TPMI_DH_ENTITY+	bind	TPM_RH_NULL
TPM2B_NONCE	nonceCaller	initial *nonceCaller,* sets *nonceTPM* size for the session
TPM2B_ENCRYPTED_SECRET	encryptedSalt	conveys a secret value used to generate the *sessionKey* – method of conveying this value is dependent on the type of *tpmKey*
TPM_SE	sessionType	indicates the type of the session (HMAC, policy, or trial)
TPMT_SYM_DEF+	symmetric	if the session is to be used for parameter encryption, set this to an algorithm and key size
TPMI_ALG_HASH	authHash	hash algorithm to use for the session; required to be a hash algorithm implemented on the TPM and not TPM_ALG_NULL

 Copyright © TCG 2006-2020

19.6.12 Salted and Bound Session Key Generation

This version of TPM2_StartAuthSession() creates a session that has properties that are similar to the OSAP session type of TPM 1.2 but also allows salting. For this version of the command, *bind* is used to provide an *authValue*, *tpmKey* encrypts the *salt* value and the *sessionKey* is computed using both.

$$sessionKey \coloneqq \mathbf{KDFa}\ (sessionAlg, (authValuebind \mid\mid \text{salt}), \text{“ATH”}, nonceTPM, nonceCaller, bits) \quad (25)$$

If the session is an HMAC session:

- Because *bind* is present, the session is bound to that entity. That is, when the session is used to authorize use of the bound entity, the HMAC will use *sessionKey* but not the *authValue*.

$$authHMAC \coloneqq \mathbf{HMAC}_{sessionAlg}\ (sessionKey, (pHash \mid\mid nonceNewer \mid\mid nonceOlder^* \mid\mid sessionAttributes)) \quad (26)$$

If the session is a policy session:

- The session is not bound to that entity. That is, when the session is used to authorize use of any entity, the HMAC (if required) will use the *sessionKey* and the *authValue*.

$$authHMAC \coloneqq \mathbf{HMAC}_{sessionAlg}\ ((sessionKey \mid\mid authValue_{entity}), (pHash \mid\mid nonceNewer \mid\mid nonceOlder^* \mid\mid sessionAttributes)) \quad (27)$$

- The *noDA* attribute of the bind object is recorded in the session context. For a description of the rationale, see clause 19.8.7.

Table 21 — Format to Start Salted and Bound Session

Type	Name	Description
TPM_ST	tag	
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_StartAuthSession
TPMI_DH_OBJECT+	tpmKey	handle of a loaded key used to encrypt *salt*
TPMI_DH_ENTITY	bind	entity providing the *authValue* and to which the session is bound
TPM2B_NONCE	nonceCaller	initial *nonceCaller,* sets *nonceTPM* size for the session
TPM2B_ENCRYPTED_SECRET	encryptedSalt	contains a secret value used to generate the *sessionKey* – method of encrypting this value is dependent on the type of *tpmKey*
TPM_SE	sessionType	indicates the type of the session (HMAC, policy, or trial)
TPMT_SYM_DEF+	symmetric	if the session is to be used for parameter encryption, set this to an algorithm and key size
TPMI_ALG_HASH	authHash	hash algorithm to use for the session; required to be a hash algorithm implemented on the TPM and not TPM_ALG_NULL

 Copyright © TCG 2006-2020

19.6.13 Encryption of *salt*

19.6.13.1 Overview

The *salt* parameter for TPM2_StartAuthSession() is asymmetrically encrypted using the methods described in this clause.

The value produced by the secret exchange process using *salt* should be the size of the digest produced by the *authHash* of the session. For ECC, the size of the *seed* is limited because it is an ECC point; but for RSA, XOR, and AES, the size of *salt* may vary.

When the value of salt is determined, it is used in the computation of *sessionKey* as shown in equation (18).

19.6.13.2 Asymmetric Encryption of Salt

The methods of encrypting the salt and producing the session secret differ for each asymmetric algorithm. The methods are described in the algorithm-specific annexes to this specification.

19.6.14 Caution on use of Unsalted Authorization Sessions

If an *authValue* has low entropy, confidentiality of the value may not be preserved if the *authValue* is used in an unsalted authorization session. For an unbound, unsalted session, the HMAC computation for the response from the TPM is:

$$authHMAC \coloneqq \mathbf{HMAC}_{sessionAlg}\,(authValue,\ (rpHash \,||\, nonceTPM \,||\, nonceCaller \,||\, sessionAttributes)) \quad (28)$$

If an attacker can read the response from the TPM, then the only values unknown to the attacker are *authValue* and *nonceCaller*. An attacker may be able to determine *nonceCaller* by reading the command as it is sent to the TPM. If the attacker has all the variables but *authValue*, they could perform an "off-line" attack on the *authValue* using trial versions of *authValue* until one is found that produces a matching *authHMAC*.

NOTE 1 In this context, an "off-line" attack means that the attacker can perform computations that do not involve the TPM meaning that the protections that the TPM provides against authValue attacks has no effect.

It is important to note that this vulnerability only occurs if an attacker has access to both the command and response of a successful command using the *authValue*. If a user has a password protecting a key and the system is lost or stolen, the key is protected because the attacker will not be able to observe the legitimate owner of the key perform a successful operation with the key.

For a bound session without salt, the attack is a bit more complicated. The HMAC computation for the response is:

$$authHMAC \coloneqq \mathbf{HMAC}_{sessionAlg}\,((sessionKey \,||\, authValue_{entity}),\ (pHash \,||\, nonceNewer \,||\, nonceOlder \,||\, sessionAttributes)) \quad (29)$$

If the attacker observes a TPM2_StartAuthSession() command and response and the *authValue* for the bind entity has low entropy, then they would have all of the components of *sessionKey* except for the *authValue* of the *bind* entity. Then, by observing another successful transaction, an attacker could know everything but the two *authValues* and they could again perform an offline attack.

NOTE 2 If the successful operation is on the *bind* entity, then only one *authValue* is unknown.

 Copyright © TCG 2006-2020

As with the unbound and unsalted session, the vulnerability for a bound session only occurs if the attacker is able to observe successful command response sequences.

Salting provides a mechanism to allow use of low entropy *authValues* and still maintain confidentiality for the authValue. It is also possible to use a high entropy authValue to protect the confidentiality of a low-entropy value. For instance, if the *bind* entity *authValue* has high-entropy, then there would be greater computational complexity in guessing *sessionKey* || $authValue_{entity}$. Depending on the *authValue* and *salt* sizes, a bound session could have a *sessionKey* that is as difficult to guess as does a salted session.

19.6.15 No HMAC Authorization

For a session-based authorization, both HMAC and policy, an *authHMAC* value is computed as shown in equation (17) and that value is used as *hmac* in an authorization or acknowledgement as shown in Table 14 and Table 15 respectively. If an authorization session is started with *bind* and *tpmKey* both set to TPM_RH_NULL, then *sessionKey* in equation (17) will be an Empty Buffer. If the *authValue* in equation (17) is also an Empty Buffer, then the HMAC key will be an Empty Buffer. When this situation exists, the caller has the option of either providing the results of the *authHMAC* computation, or not.

If *authHMAC* is provided, it will be computed as shown in equation (17) with an Empty Buffer as the HMAC key and the TPM will validate that the value in *hmac* matches the internally calculated value.

If *authHMAC* is not provided, the size of *hmac* (see Table 14) will be zero and the TPM will accept this value of *hmac* as providing valid authorization for the object.

For an HMAC session, *authValue* in equation (17) will only be an Empty Buffer if the *authValue* of the authorized object is an EmptyAuth, the session is a bind session and the authorization is for the entity to which the session is bound, or if the session is not an authorization session.

For a policy session, two situations will result in *authValue* being an Empty Buffer:

1) the *authValue* of the authorized object is an EmptyAuth, or
2) the policy does not use the *authValue* of the object (that is, the evaluated policy does not contain TPM2_PolicyAuthValue())(see 19.7.7.6).

For these two cases, if *sessionKey* is an Empty Buffer, *hmac* is allowed to be either a valid *authHMAC* or an Empty Buffer. For a bound or salted policy session, *sessionKey* is not an Empty Buffer, and *hmac* must be valid.

NOTE A policy session that does not use TPM2_PolicyAuthValue() would use a bound or salted session if that session is also used for encryption.

For a policy session that contains TPM2_PolicyPassword(), the password takes precedence and must be present in *hmac*.

The TPM will use the same formulation in the response as was in the command. This is, if *hmac* was non-zero in the command, the TPM will compute *authHMAC* as shown in equation (17) and use the result as *hmac*. If *hmac* was an Empty Buffer in the command, it will be an Empty Buffer in the response.

19.6.16 Authorization Selection Logic for Objects

Each object has two attributes in its public structure to indicate how use of the object is authorized.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

1) ***userWithAuth*** – If this attribute is SET, then USER role authorization for an object may be provided with an HMAC session or a password. If this attribute is CLEAR, then the *authValue* may not be used for USER role authorization, meaning that authorization may not be done using an HMAC session or a password. USER role authorizations with a policy are always allowed regardless of the setting of this attribute.

2) ***adminWithPolicy*** – If this attribute is SET, then ADMIN role authorization for an object may only be provided with a policy session. If this attribute is CLEAR, then authorization may be provided with a policy session, with an HMAC session, or with a password.

When authorization is with a policy session and ADMIN role authorization is being provided, the command code value of the policy session must match the command code for the command being authorized.

For TPM_RH_OWNER, TPM_RH_ENDORSEMENT, and TPM_RH_PLATFORM); *userWithAuth* and *adminWithPolicy* are always SET.

For an NV Index, NV Index attributes (TPMA_NV) determine authorization selection.

NOTE For TPM_RH_OWNER, TPM_RH_ENDORSEMENT, and TPM_RH_PLATFORM); *userWithAuth* and *adminWithPolicy* do not have to be implemented as separate attributes. The code may simply assume that the attributes are SET and act accordingly.

19.6.17 Authorization Session Termination

The TPM will terminate a session (authorization or audit) and clear all associated context under the following circumstances:

- when TPM2_FlushContext() selects the session;
- if *sessionAttributes.continueSession* is CLEAR in the command, the TPM will CLEAR the continueSession flag in the response and perform TPM2_FlushContext() actions;

NOTE When *sessionAttributes.continueSession* is CLEAR in the command but the command does not return success, then the session is not terminated.

- on TPM Reset, all authorization sessions are terminated; and
- on TPM Resume or TPM Restart, authorization sessions in TPM memory will be terminated but sessions context saved off the TPM will remain active.

19.7 Enhanced Authorization

19.7.1 Introduction

Enhanced authorization is a TPM capability that allows entity-creators or administrators to require specific tests or actions to be performed before an action can be completed. The specific policy is encapsulated in a value called an *authPolicy* that is associated with an entity

When an HMAC session is used for authorization, the *authValue* of the entity is used to determine if the authorization is valid. When a policy session is used for authorization, the *authPolicy* of the entity is used.

Many TPM entities have or may have an associated *authPolicy*. A policy defines the conditions for use of an entity. For example,

- a policy may limit the use of a key unless selected PCR have specific values;
- a policy may not allow use of a key after a specific time;
- a policy may require that authorization to change an NV Index be provided by two different entities; or
- a policy may limit a particular signing key to attest to PCR values but not to certify another TPM key.

A policy may be arbitrarily complex. However, the policy is expressed as one (statistically unique) digest called the *authPolicy*.

The digest representing a particular policy may be included in an Object or NV Index when the Object or NV Index is created (the digest representing a policy is created using the methods described in subsequent parts of this clause). In order to use the Object or Index, a policy session is created and then the TPM is given a sequence of policy commands that modify the digest in the policy session. After executing all of the commands of the policy, the TPM will have computed a digest value that is characteristic of the policy. The policy session is then used as an authorization session. If the digest accumulated in the policy session matches the *policyDigest* of the entity (and certain other optional conditions are true) then the command is authorized.

After a policy session is used for authorization, *policySession→nonceTPM* is changed to a new, random value; *policySession→startTime* is set to the current time; and the other values of the policy session context are initialized to the state they had when the session was first created by TPM2_StartAuthSession() (see19.7.8).

The mechanisms of policy creation and evaluation are explained in the remainder of this clause

19.7.2 Policy Assertion

An assertion is a statement that something is true. In an authorization policy, an assertion is a statement of something that must be true before the policy is satisfied. For example, an assertion may be that a set of PCR must have specific values to allow an object to be authorized for use in a specific command. The list of all policy assertions defined by this specification is in 19.7.7.6.

A combination of one or more assertions is used to construct an authorization policy.

19.7.3 Policy AND

A policy may be expressed in an equation as a set of assertions that must all be satisfied before the policy is valid. For example, a policy that requires that 4 assertions be true could be written as:

$a \,\&\, b \,\&\, c \,\&\, d$

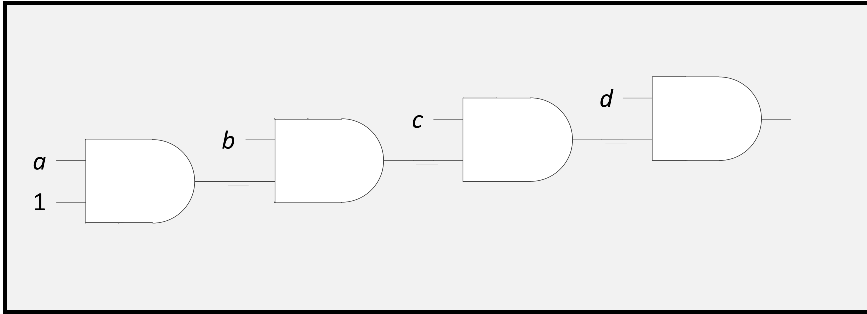
A possible implementation of the policy logic would be to have all the assertions evaluated at the same time to determine if the policy is satisfied. This approach would require that the TPM resources scale with the number of assertions that would need to be evaluated for the policy.

The alternative use in the TPM is to evaluate the expression one assertion at a time with each assertion ANDed with the results of the previous evaluation.

$(((1 \,\&\, a) \,\&\, b) \,\&\, c) \,\&\, d$

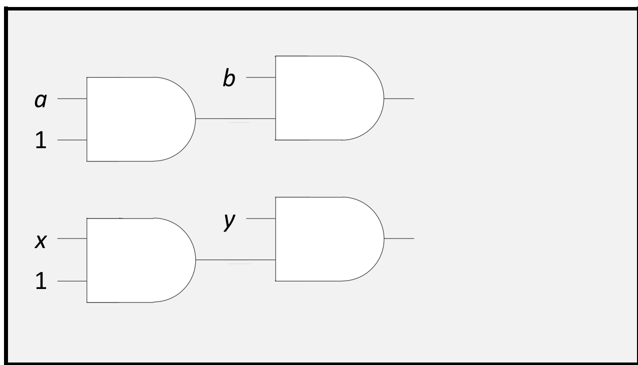
The (1 & *a*) term means that assertion *a* is ANDed with an initial TRUE. This allows each assertion to be just the AND of a new assertion with the results of the previous assertion evaluation. A pictorial representation of the policy evaluation is:

 Copyright © TCG 2006-2020



Any number of assertions can be combined in this way using a fixed set of TPM resources.

The logic of a TPM policy cannot actually be expressed as a simple 1 or 0. For the policy to be valid, not only does it need to evaluate to "TRUE", but it also has to be the correct policy. For example, these two policies may both evaluate to the same logic value (TRUE), but they do not represent the same policies.



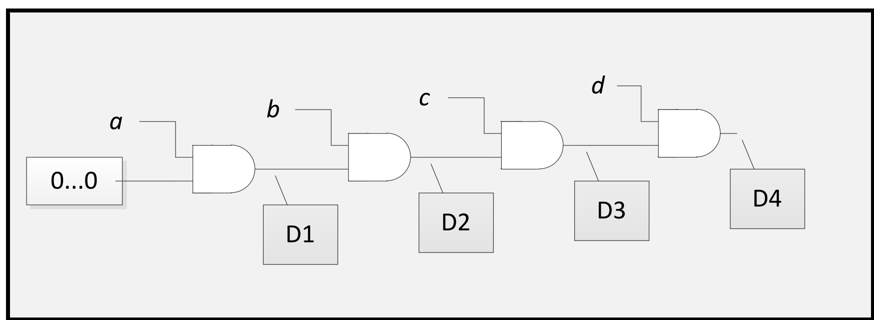
So that it can differentiate (*a* & *b*) from (*x* & *y*), the TPM will update a running digest value for each assertion that is added to the policy. The final digest value indicates the policy that was evaluated.

The running digest value is called the *policyDigest*. The *policyDigest* is initialized to a Zero Digest (0…0) when the policy session is started (TPM2_StartAuthSession()). Then, as each policy assertion is evaluated, the *policyDigest* is updated.

$$policyDigest_{new} := \mathbf{H}(policyDigest_{old} \mid\mid PolicyAssertion)$$

NOTE 1 This should be recognizable as the Extend operation.

The *policyDigest* will only be updated if a policy assertion is valid (TRUE) (see 19.7.10 for exception relating to trial policies). This gives an alternative possibility for interpreting the output of one of the policy AND gates. Instead of simply being a 1 (TRUE) or 0 (FALSE), the output of the gate is current value of the *policyDigest*. Using this perspective, the four-term policy becomes:



 Copyright © TCG 2006-2020

where

0…0	the initial value of the policy digest
D1	$\mathbf{H}$(0…0 || *a*)
D2	$\mathbf{H}$(D1 || *b*)
D3	$\mathbf{H}$(D2 || *c*)
D4	$\mathbf{H}$(D3 || *d*)

NOTE 2 In these illustrations, the parameters for the Extend operations are simple parameters ("*a*", "b", etc.). The actual parameters for the Extend are more complex but including the details in the illustrations would add complexity without adding clarity.

19.7.4 Policy OR

If the only type of policy assertion was an AND, then the policies that could be evaluated by the TPM would be of limited value. To make the policies more flexible, an OR policy assertion is defined. As with a logic OR gate, the OR policy assertion will be valid if any of the inputs is valid.

A simple policy using an OR might be written as:

(*a* & *b*) | (*x* & y)

or as:

(((0…0) & *a*) & *b*) | (((0…0) & *x*) & *y*)

Evaluating the AND branches individually, the left side evaluates to:

$$D_{\text{left}} \coloneqq \mathbf{H}(\mathbf{H}(0...0 \mid\mid a) \mid\mid b)$$

and the right side to:

$$D_{\text{right}} \coloneqq \mathbf{H}(\mathbf{H}(0...0 \mid\mid x) \mid\mid y)$$

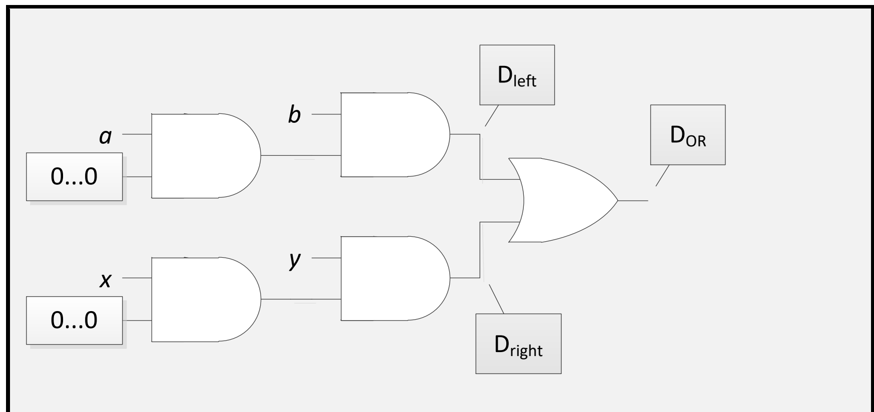
Then, the output from a 2-input policy OR operation will be defined to be

$$policyDigest_{new} \coloneqq \mathrm{H}(D_{\text{left}} \mid\mid D_{\text{right}})$$

Notice that the OR operation replaces the *policyDigest* with a new value instead of Extending it as is done in an AND operation.

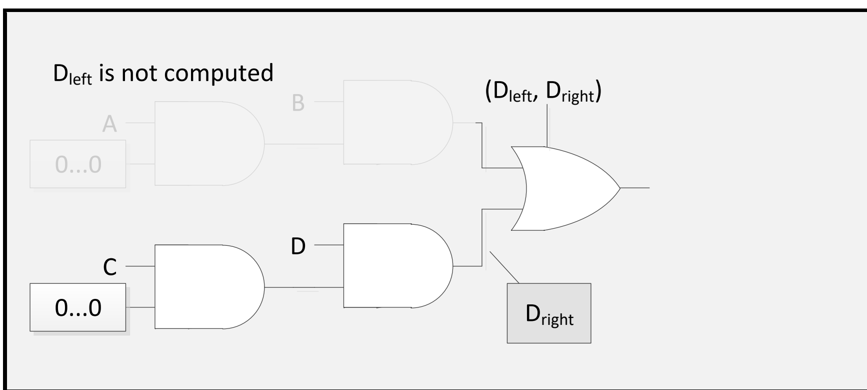
Pictorially, a policy with an OR is:

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019



The TPM processes the OR by comparing the current value of *policyDigest* with a list of digest values provided by the caller. If *policyDigest* is on the list, then the TPM will digest the concatenation of all of the digests in the list. For example, to perform the OR operation above, assume that the TPM has processed (*a* & *b*) producing D_{left}. Then the TPM would be given a list of digests (D_{left}, D_{right}). Because the *policyDigest* is on the list, the TPM computes $D_{OR} \coloneqq \mathbf{H}(D_{left} \mid D_{right})$ and replaces *policyDigest*. Note that if the TPM had processed (*c* & *d*) to compute D_{right} and was then given the same list of digests (D_{left}, D_{right}), the resulting *policyDigest* would be the same.

When processing a policy that has an OR, only one branch of the policy needs to be evaluated. For example, if C and D assertions were valid, then only the *right* branch would need to be evaluated.



The list given to the TPM for a TPM2_PolicyOR() is limited to 8 digests. However, the effective size of the list can be expanded indefinitely by using cascading OR. Figure 15 illustrates one of the many ways to construct a 12 input OR.

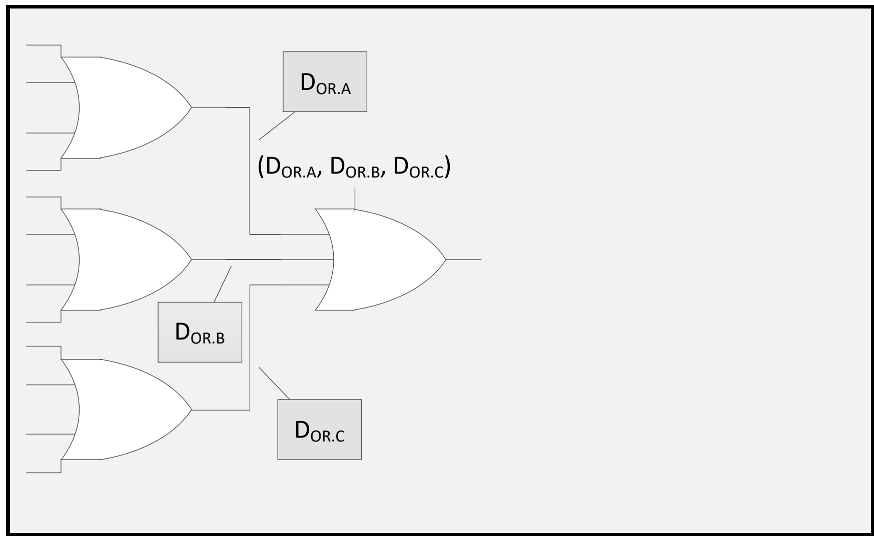


Figure 15 — A 12-input OR Policy

When the OR list can contain 8 digests, 64 different branches can be ORed in just two levels.

The result of an OR operation may be an input to an AND assertion allowing construction of arbitrarily complex policies.

19.7.5 Order of Evaluation

Because the TPM uses digests, the order of evaluations is important. For policy evaluation, (A & B) is not the same as (B & A). In addition, when performing an OR operation, the same list of digests (same number in the same order) must be given to the TPM each time. The list (D_{left}, D_{right}) will not give the same result as (D_{right}, D_{left}) or (D_{left}, D_{right}, D_{other}).

19.7.6 Policy Session Creation

TPM2_StartAuthSession(*sessionType* = TPM_SE_POLICY) is used to start an authorization session. The authorization session may use any of the four options for *tpmKey* and *bind.*

NOTE 1 A policy session does not maintain a binding with a specific object. The bind parameter is used only for session key creation. This allows the context space of the session that is used for the binding value to be dedicated to other policy parameters.

The most typical use of a policy session will be with *tpmKey* and *bind* both set to TPM_RH_NULL. When this option is selected, an HMAC computation might not be performed when the policy session is used and the session *nonce* and *auth* values may be Empty Buffers (see TPM 2.0 Part 3, TPM2_PolicyAuthValue).

NOTE 2 When the session is created, *nonceCaller* still needs to be provided and its size is required to meet the minimum requirements of the command.

When the authorization session is to be used to authorize a command that has an encrypted command or response parameter, then either *tpmKey* or *bind* should be used in the TPM2_StartAuthSession() that starts the session so that a secure *sessionKey* is created.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

19.7.7 Policy Assertions (Policy Commands)

19.7.7.1 Introduction

In TPM 2.0 Part 3 of the specification the set of policy assertions are the commands with names of the form TPM2_Policyxxx() where "xxx" is an indicator of the type of policy assertion. For example, TPM2_PolicySigned() is a policy assertion that an authorization was signed by a specific entity; and TPM2_PolicyPCR() is an assertion that a selected set of PCR have a specific value.

Normally, each policy command will cause the *policyDigest* to be changed in a different way which is why they are different commands. In some cases, the policy command will also cause other changes to the policy session context. For example, TPM2_PolicyLocality() modifies the policy state that indicates the locality that is allowed when the policy session is used for authorization. TPM2_PolicyCommandCode() changes the policy state so that the policy may only be used to authorize a specific command.

The details of the *policyDigest* computation performed by each policy command are provided in the General Description section of each command found in TPM 2.0 Part 3. The description also indicates the policy state that is modified.

The assertions fall into three different groups: immediate, deferred, and combined.

19.7.7.2 Immediate Assertions

For an immediate assertion, the input values are validated and the TPM will return a failure and not update the *policyDigest* if the assertion is not valid. An example of an immediate assertion is TPM2_PolicyNV(). For this assertion, the TPM validates the logical or arithmetic relationship between an input value and an NV Index. If the specified relationship is not valid, the TPM returns an error and the *policyDigest* is not modified. If the relationship is valid, then the *policyDigest* is updated with the Index Name and the relationship that was validated.

19.7.7.3 Deferred Assertions

For a deferred assertion, the TPM will update the *policyDigest* based on the input values and record some parameters in the policy session's context. These parameters are checked when the policy is used for authorization. An example of a deferred assertion is TPM2_PolicyCommandCode(). For this assertion, the input is a TPM command code. The *policyDigest* will be updated to record the fact that the TPM2_PolicyCommandCode() was executed and the *commandCode* value that was specified. The TPM also directly records the *commandCode* parameter in the policy session context. When the policy is used for authorization, the TPM will verify that the command being authorized is the same as the command in the policy and the authorization (and command) will fail if they are not the same.

19.7.7.4 Combined Assertions

For a combined assertion, the TPM will validate some condition of the input and record or modify some parameters in the policy session's context. An example of a combined assertion is TPM2_PolicySigned(). For this assertion type, the TPM validates that the parameters of the command have been signed by the indicated key. If so, it will update the policy session context based on the input parameters. One of the context values that may be updated is the *cpHash* of the session. If the *cpHash* of the authorized command is not the same as the authorized *cpHash* then the command will not be authorized.

19.7.7.5 Repetition of Assertions

In general, any policy assertion may occur multiple times within a policy as long as the assertion is compatible with previous assertions. An example of an incompatible set of assertions is two occurrences of TPM2_PolicyCommandCode() that specify different command codes.

The TPM will return an error if an assertion is incompatible with a previous assertion. It is possible that the failed assertion is incompatible with an assertion of a different type. For example, a TPM2_PolicyCpHash() may be incompatible with a TPM2_PolicySigned(). If they specify different values of *policySession→cpHash,* then the TPM will return an error.

NOTE When referring to an *element* of the policy context, the notation *policySession→element* is used to denote a particular member of the policy context.

19.7.7.6 List of Assertions

The assertions listed in this clause will all update the *policyDigest* of the policy session being operated on if the assertion condition is met. They may also cause a change to other policy session, context values (the list of policy session context values is in 19.7.8) as indicated in the brief description for each assertion.

 Copyright © TCG 2006-2020

- **TPM2_PolicyAuthorize()** – valid if *policySession→policyDigest* has the value authorized by the selected key. This is an immediate assertion and is described in more detail in 19.7.11.
- **TPM2_PolicyAuthorizeNV()** – valid if the specified NV Index contains a hash algorithm identifier and a digest value that match the hash algorithm and *policySession→policyDigest*. This immediate assertion changes *policySession->policyDigest* and is described in more detail in 19.7.11.
- **TPM2_PolicyAuthValue()** – valid if *authValue* of the authorized entity is provided when the policy session is used for authorization. This deferred assertion will SET *policySession→isAuthValueNeeded*. When the policy is used for authorization, the TPM will check *policySession→isAuthValueNeeded*. If it is SET, then the TPM performs an HMAC check on the session as if it were an HMAC session. This HMAC validation will only succeed if the caller is able to prove knowledge of the entity's *authValue* by computing the correct HMAC.
- **TPM2_PolicyCommandCode()** – valid when the authorized command has the specified command code. This deferred assertion sets *policySession→commandCode*.
- **TPM2_PolicyCounterTimer()** – valid when a portion of the TPM's TPMS_TIME_INFO structure has the desired numerical relationship with another value. This is an immediate assertion. If the selected subset of the TPM's TPMS_TIME_INFO structure does not have the specified relationship with the input data, then the TPM will return an error and not change the *policyDigest* (see 36.1 for use cases).
- **TPM2_PolicyCpHash()** – valid if the cpHash of the authorized command has a specific value. This deferred assertion *modifies policySession→cpHash*.
- **TPM2_PolicyDuplicationSelect()** – valid if the handles of the authorized command reference specific objects and the command code is TPM2_Duplicate(). This deferred assertion modifies *policySession→cpHash* and *policySession→commandCode*.
- **TPM2_PolicyLocality()** – valid if the command being authorized is being executed at one of the allowed localities. This is a deferred assertion that modifies *policySession→locality*. For localities 0-4, the input locality parameter is a bit field that indicates the allowed localities. If an execution of this assertion would result in no locality being allowed, then the TPM will return an error. For extended localities, *policySession→locality* is set to the *locality* parameter of the command if the *policySession→locality* was not previously set. Otherwise, the *locality* parameter is required to be the same as the current value of *policySession→locality.*
- **TPM2_PolicyNameHash()** – valid if the handles of the authorized command reference specific objects. This deferred assertion modifies *policySession→cpHash.*
- **TPM2_PolicyNV()** – valid if the contents of NV have the desired relationship with another value. This is an immediate assertion. If the selected portion of the NV Index does not have the specified relationship with the input data, then the TPM will return an error and not change the *policyDigest.*
- **TPM2_PolicyNvWritten()** – valid when the TPMA_NV_WRITTEN attribute of the specified NV Index has the desired value. This deferred assertion sets *policySession→checkNvWritten* and the state of *policySession→nvWrittenState*.
- **TPM2_PolicyOR()** – valid if *policySession→policyDigest* is on a list of digests. This is an immediate assertion. If *policySession→policyDigest* is not on the list of digests, then TPM returns an error. Otherwise, *policySession→policyDigest* is replaced with the digest of the list.
- **TPM2_PolicyPassword()** – valid if the *authValue* of the authorized entity is provided when the session is used for authorization. This deferred assertion will SET *policySession→isPasswordNeeded*. When the policy is used for authorization, the TPM will check *policySession→isPasswordNeeded*. If it is SET, then the TPM performs a password check on the session as if it were a password session. . This password validation will only succeed if the caller is able to prove knowledge of the entity's *authValue* by providing the correct value as the password.

NOTE 1 A session may use TPM2_PolicyAuthValue() and TPM2_PolicyPassword() interchangeably. If TPM2_PolicyAuthValue() and TPM2_Policy Password() are both used, then TPM will perform the check according to the last one used in the policy.

- **TPM2_PolicyPCR()** – valid if the selected PCR have the desired value. This assertion may be either a combined or a deferred assertion. If the caller provides a digest, the TPM validates that the current values of the PCR match the input value and return an error (TPM_RC_VALUE) if not. If this command completes successfully, the *policyDigest* will have been updated with the digest of the selected PCR. The TPM will also record that the PCR have been checked. If the PCR are changed after they are checked but before the policy is used for authorization, then the policy will fail.

NOTE 2 The reference implementation provides this assurance by maintaining a PCR update counter that increments each time the PCR are modified. The update counter is saved in the policy session context. If the update counter does not change between the check of the PCR and the use of the policy session for authorization, then the PCR are the same.

- **TPM2_PolicyPhysicalPresence()** – valid if the physical presence is asserted when the authorized command is executed. This deferred assertion sets *policySession→isPPRequired.*
- **TPM2_PolicySecret()** – valid if the knowledge of a secret value is provided. This assertion is an immediate and possibly also a deferred assertion. Based on the input parameters, this command may modify *policySession→cpHash* and *policySession→timeout.*

NOTE 3 The secret value will be the *authValue* of some TPM entity.

- **TPM2_PolicySigned()** – valid if the parameters are properly signed. This assertion is an immediate and possibly also a deferred assertion. Based on the input parameters, this command may modify *policySession→cpHash* and *policySession→timeout.*
- **TPM2_PolicyTemplate()** – valid if the hash of the *inPublic* parameter of TPM2_Create(), TPM2_CreatePrimary(), or TPM2_CreateLoaded() matches the *templateHash* in this command. This deferred assertion sets *policySession→cpHash.*
- **TPM2_PolicyTicket()** – valid if the ticket is valid. This assertion is an immediate and possibly also a deferred assertion. Based on the input parameters, this command may modify *policySession→cpHash* and *policySession→timeout*.

19.7.8 Policy Session Context Values

A policy session context contains the state and tracking information for evaluation of a policy. The context values are set to their default values when the session is created and again each time the session is successfully used to authorize a command.

The values may be changed by a policy assertion. The policy assertions are listed in 19.7.7.6 with an indication of the policy session context values that they modify. The policy session context values are described further here.

- ***policyDigest*** – digest that is updated by each assertion. The default value for *policyDigest* is a Zero Digest (a buffer with a length equal to the digest size of the hash algorithm with all octets having a value of zero).
  - ***nonceTPM*** – set from the RNG and is sized according to the size of *nonceCaller* in TPM2_StartAuthSession(). This value does not change during the policy evaluation. However, it does change when the policy session is used for authorization.
  - ***cpHash*** – set by an assertion that limits the authorization to a specific set of command parameters. If an assertion would set *policySession→cpHash* and a previous assertion has set *policySession→cpHash* to a different value, then the assertion will fail. The default for *policySession→cpHash* is an Empty Buffer.
  - ***nameHash*** – set by TPM2_PolicyNameHash() and indicates the combination of Name values for a command. This context parameter occupies the same location as *policySession→cpHash*. If an assertion would set *policySession→cpHash* and a previous assertion has set *cpHash* to a different value, then the assertion will fail. The default for *policySession→nameHash* is an Empty Buffer.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

- ***startTime*** – set to TPMS_TIME_INFO.*clockInfo.clock* when *policySession→nonceTPM* changes. No assertion changes this value. It is updated to the current value of *clock* by TPM2_StartAuthSession() and when the session is used for authorization.
- ***timeout*** – the time when the policy session expires. Its default setting is an implementation-specific value corresponding to "never expires." This value is updated if an assertion has a non-zero expiration time that is sooner than the current setting of *policySession→timeOut*. An assertion may only decrease the value of *policySession→timeout.*
- ***commandCode*** – set by an assertion that limits the policy to a specific command but does not limit the command parameters (TPM2_PolicyCpHash() limits the command and its parameters). If an assertion sets *policySession→commandCode* and a previous assertion has set *policySession→commandCode* to a different value, then the TPM will return an error. The default for *policySession→commandCode* is an implementation-specific value that indicates that it has not been set.
- ***pcrUpdateCounter*** – set by TPM2_PolicyPCR(). The TPM maintains a *pcrUpdateCounter* that is incremented each time a PCR changes (with a few exceptions as described in 17.9). When it executes TPM2_PolicyPCR(), the TPM will copy *pcrUpdateCounter* to *policySession→pcrUpdateCounter.* When the policy session is used for authorization, the TPM will verify that *policySession→pcrUpdateCounter* matches *pcrUpdateCounter.* A match provides assurance that the PCR values still match the values evaluated by TPM2_PolicyPCR().
- ***commandLocality*** – indicates the locality required for the command being authorized by the policy. The default for *policySession→commandLocality* is any locality. Each locality that is not enabled in TPM2_PolicyLocality(*locality)* is disabled in *policySession→commandLocality*. If the result of this operation would result in there being no locality at which the policy would be valid, the TPM will return an error and not change *policySession→commandLocality*. If *commandLocality* is set to an extended locality (greater than 31), then the locality cannot be change by subsequent TPM2_PolicyLocality().
- ***isPPRequired*** – SET by TPM2_PolicyPhysicalPresence() to indicate that presence is required to be asserted when authorized command is executed. The default value is CLEAR.
- ***isAuthValueNeeded*** – SET by TPM2_PolicyAuthValue()to indicate that the *authValue* of the authorized entity will need to be provided when the policy session is used for authorization. The *authValue* is required to be included in an HMAC. The default value is CLEAR. It will also be CLEAR by TPM2_PolicyPassword()
- ***isPasswordNeeded*** – SET by TPM2_PolicyPassword() to indicate that the *authValue* of the authorized entity will need to be provided when the policy session is used for authorization. The *authValue* is required to be provided as a password. The default value is CLEAR. It will also be CLEAR by TPM2_PolicyAuthValue().
- ***isTrialPolicy*** – SET to indicate that policySession→policyDigest is to be updated even if the assertion is not valid. The session may not be used for authorization.
- ***checkNvWritten* –** SET to indicate that the TPMA_NV_WRITTEN attribute of the authorized NV Index must be compared with *nvWrittenState.*
- ***nvWrittenState*** – SET when TPMA_NV_WRITTEN is required to be SET in the NV Index being authorized. This attribute has no meaning when ***checkNvWritten*** is not SET.

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

19.7.9 Policy Example

In TPM 1.2, the basic policy for use of a key was limited to a combination of an authorization value and PCR state. This policy was built in to each key. In TPM 2.0 there is no built-in policy. A TPM 2.0 policy that is the same as the TPM 1.2 policy is:

TPM2_PolicyPCR() & TPM2_PolicyAuthValue()

Note This policy could also be written as

TPM2_PolicyAuthValue() & TPM2_PolicyPCR()

This policy would have a different *policyDigest* because the order of evaluation affects the digest.

To associate this policy with a key, evaluate the policy to determine the *policyDigest* that it would generate. Then create the key with this digest as the *authPolicy* and CLEAR the *userWithAuth* attribute. When *userWithAuth* is CLEAR, USER mode actions for the key will require use of the key's *authPolicy*.

19.7.10 Trial Policy

The policy evaluation to determine the value for the *authPolicy* may be done in software that does the same *policyDigest* computation as the TPM. Alternatively, a trial policy session may be used. A trial policy session is created and used in a sequence of policy commands just like a normal policy session. The difference is, in a trial policy, a policy assertion is always assumed to be TRUE and the *policyDigest* updated accordingly. The *policyDigest* value computed in the trial policy can be read from the TPM and used as an object's *authPolicy*. Since the assertions in the trial policy do not need to be valid, the trial session may not be used for authorization.

19.7.11 Modification of Policies

Some policies, such as those associated with the hierarchies, may be altered directly by changing the *authPolicy* value. Policies associated with Objects and NV Indices may not be directly altered. The reason that these policies may not be altered is that the policy can affect the trust that someone places in the use of that entity. For example, a key may only be trusted if it may only be used when the PCR have a specific set of values. If the policy could be changed, then the PCR check could be removed, and the key would no longer be trusted. There would be no way for the trusting entity to know if a version of the key exists where the PCR are not checked.

Even though there is no way to directly change a policy, it can be indirectly changed. The command that allows this is TPM2_PolicyAuthorize(). When this command is included in a policy, it allows a designated entity (an "authority") to authorize a *policyDigest* to be included in the policy. This is best described with an example.

It is common to seal a data value to PCR values so that the data value can only be recovered if the platform has booted in a known way. A problem with this is that if there is a BIOS update, the PCR will change, and the sealed data value can no longer be retrieved, and some kind of recovery process is necessary. The inability of a policy to accommodate changes to PCR values is called "brittleness". That term suggests that the policy is easy to break (make unusable). This brittleness could be a problem with TPM 2.0 if the policy was completely fixed.

Figure 16 illustrates the use of TPM2_PolicyAuthorize() to implement a flexible policy. This assertion evaluation checks to see if the current *policyDigest* is authorized by a signing key – that is, did an authorizing entity sign a digest indicating that a specific value of *policyDigest* represents a known set of PCR values. If the *policyDigest* value was signed, then *policyDigest* is replaced by a digest of the Name of the key that was used for authorization and *policyRef* (see 19.7.12).

 Copyright © TCG 2006-2020

NOTE 1 Other information is included with the Name of the key when the new *policyDigest* is computed in order to indicate that the Name was included as the result of a TPM2_PolicyAuthorized() operation.

NOTE 2 This example purposefully avoids using terms that would indicate that the signing entity does anything other than indicate that the PCR values are the expected values. In particular, the signing entity does not have to certify that the PCR values are safe. The signing entity may provide other assurances but, in the case of PCR, it is not necessary to warrant anything other than that the PCR values are expected.

An example of how of this assertion type may be used to avoid PCR brittleness is shown in Figure 16. This shows the example policy in 19.7.9 but with the ability to satisfy the policy with different PCR values.

NOTE 3 The actual *authPolicy* in the authorized entity would contain (PolicyAuthorize & PolicyAuthValue).

As shown, a PolicyPCR assertion is followed by PolicyAuthorize(). If there is an authorization signed by KEY for the current *policyDIgest* (in this case, $D_{PCR.A}$), then the result of the PolicyAuthorize() will be D_{KEY}. This is the same output that would be produced if the input to the PolicyAuthorize() were $D_{PCR.B}$ and there was an authorization sighed by KEY for $D_{PCR.B}$. That is, in TPM2_PolicyAuthorize(), if the key authorized the current *policyDigest*, *policyDigest* reset to a Zero Digest and then extended with the Name of the key. The *policyDigest* value D_{final} no longer reflects the previous value ($D_{PCR.A}$ or $D_{PCR.B}$).

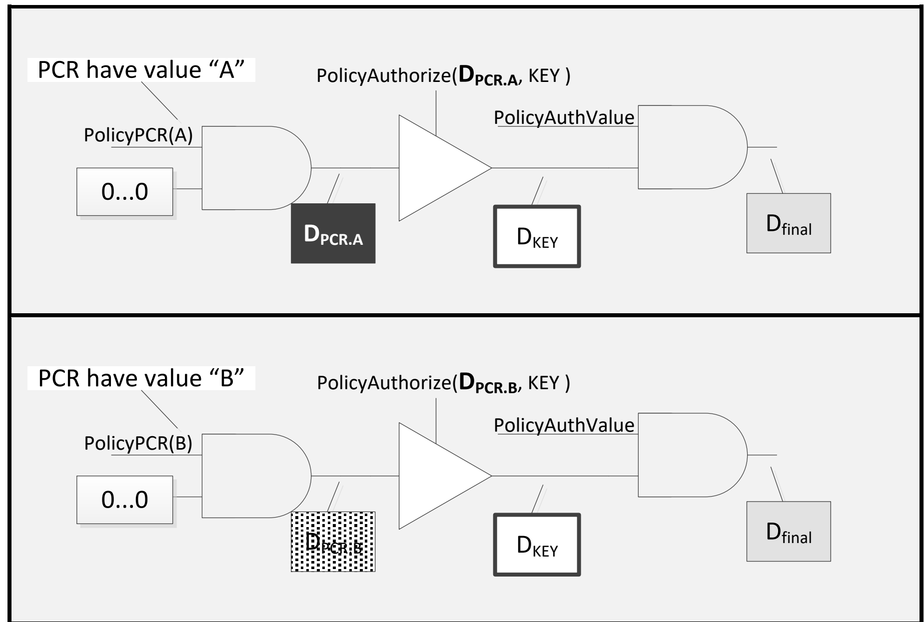


Figure 16 — Use of TPM2_PolicyAuthorize() to Avoid PCR Brittleness

In the case of a BIOS update that changes PCR, the platform OEM could provide a signature for the PCR values created by the new BIOS. Now, if the policy of the sealed data includes a TPM2_PolicyAuthorize() from the OEM, then the BIOS can be updated, and no recovery process would be needed to deal with the new PCR values. That is, with either authorized set of PCR, D_{KEY} and D_{final} will be the same, even though $D_{PCR.A}$ and $D_{PCR.B}$ are different.

An additional way to indirectly modify a policy uses TPM2_PolicyAuthorizeNV(). This command specifies an NV Index location of a policy that will be effective for an entity, the effective policy being the policy digest stored in the data at that NV index. When a policy is formed using TPM2_PolicyAuthorizeNV(), the NV Index Name is specified. When the entity is to be authorized, the policy stored in the data of the named index is satisfied, and then TPM2_PolicyAuthorizeNV() is executed. If the policy stored in that index matches the *policySession->policyDigest*, then the *policySession->policyDigest* is replaced with results of first setting the *policySession->policyDigest* to the Zero Digest, and then extending it with the command code concatenated with the name of the NV index, which will contain the modified policy. The

 Copyright © TCG 2006-2020

main difference between this command and TPM2_PolicyAuthorize() is that multiple policies can satisfy TPM2_PolicyAuthorize(), as long as they are all signed with the appropriate key. Only the policy that is currently stored in the NV index can satisfy TPM2_PolicyAuthorizeNV().

19.7.12 TPM2_PolicySigned(), TPM2_PolicySecret(), and TPM2_PolicyTicket()

The set of assertions discussed in this clause have properties that enable a number of authorization scenarios. Among these are:

- the ability to grant an authorization that can persist for a specific amount of time (because in many protocols, access to a resource (such as, a network) is granted for some time interval); and
- the ability to associate an authorization with a policy of the authorizing entity (because in many instances, the authorizing entity may use the same key or secret for different purposes).

TPM2_PolicySigned() and TPM2_PolicySecret() convey an authorization by signing a set of parameters that indicate the nature of the authorization. With TPM2_PolicySigned() the signature is with a key value (symmetric or asymmetric) and with TPM2_PolicySecret() the signature is with an HMAC using an *authValue* in the HMAC key.

The policy assertions of TPM2_PolicySigned() and TPM2_PolicySecret() can be time limited. When a policy's authorization is time limited, it expires (is no longer valid) when TPM *Time* is greater than the indicated value for the authorization. An expiration time can be expressed in two ways. A *timeout* is an absolute value of *Time*. An *expiration* value is a relative value in seconds from the start *Time* of the policy session to which the authorization applies.

Both TPM2_PolicySigned() and TPM2_PolicySecret() can produce tickets that enable authorizations to be used and reused over a period of time and in different policy sessions. These tickets are used in TPM2_PolicyTicket().

These three commands have several input parameters in common:

- **nonceTPM** – the value returned by TPM2_StartAuthSession() or when a session is used for an authorization. It is used to limit the use of a policy assertion to a specific policy session. If a policy command includes a *nonceTPM,* then the TPM will return an error if it does not match *policySession→nonceTPM*.
- **cpHashA** – if the caller chooses to limit the authorization to a specific command and command parameters, they would include this value in the signed data structure. Use of this parameter allows the caller to provide an authorization that is similar to the HMAC authorization. That type of authorization is only valid for a specific command and set of command parameters. If this parameter is not part of the signed authorization, then this parameter should be set to the Empty Buffer.
- **policyRef** – in some circumstances, it is desirable to have an authorization convey some information relating to the authorizing entity. For example, a fingerprint reader may have a signing key that it uses to verify when it has recognized a fingerprint regardless of whose fingerprint it might be. This type of authorization would be difficult to use if it were not possible to indicate whose fingerprint was scanned. The *policyRef* parameter would allow the fingerprint reader to provide this indication. The TPM includes this value in the *policyDigest*. In the example of the fingerprint reader, this would mean that the *policyDigest* would only have the correct value if the fingerprint reader scanned a finger from the correct person. If this parameter is not part of the signed authorization, then this parameter should be an Empty Buffer.

  NOTE 1 Because TPM2_PolicySigned() does not include the *cpHashA* and *policyRef* size in the *aHash* calculation, it is recommended that the size of *policyRef* not be the same size as that of the entity Name algorithm.

 Copyright © TCG 2006-2020

- **expiration** this parameter is used to place a time limit on an authorization. It is either the number of seconds from the last time that the *nonceTPM* of a policy session was changed, or the value of Time after which a policy assertion is no longer valid.
- **timeout –** this indicates the value of *Time* after which a policy assertion is no longer valid.

If a TPM2_PolicySigned() or TPM2_PolicySecret() has a non-zero *expiration* parameter:

- If *nonceTPM* is not included, *expiration* is a *timeout.*
- If expiration is a negative number, a ticket will be produced

NOTE 2 For TPM2_PolicySecret(), if *authHandle* references a PIN Pass Index, then no ticket will be produced even if *expiration* is negative. This prevents use of a ticket to bypass the limit count on an PIN Pass Index.

When a policy session is started, a *nonceTPM* is generated and the current value of *Time* is copied to *policySession→startTime*. When a policy assertion includes a non-zero *expiration* and a *nonceTPM*, then *policySession→startTime* is added to the absolute value of *expiration* to determine the *timeout* for the policy assertion. If a policy assertion includes a non-zero value in its *expiration* parameter but no *nonceTPM*, then the *expiration* parameter is used directly as a *timeout*

When an assertion produces a *timeout*, the *timeout* value is placed in *policySession→timeout.* If *policySession→timeout* has previously been set, then it will be updated with the lesser of *timeout* and *policySession→timeout*.

When *Time* has a greater value than *policySession→timeout,* the policy session expires and cannot be used for authorization. If the authorization used a *timeout* (no *nonceTPM*), then the authorization will also expire on the next TPM Reset.

NOTE 3 A policy assertion includes an *expiration* when the *expiration* parameter is non-zero. A policy assertion includes a *nonceTPM* when its *nonceTPM* parameter is not the Empty Buffer.

NOTE 4 *expiration* may need to be converted to milliseconds before being added to *policySession→startTime.*

NOTE 5 When an *expiration* parameter is used directly as a *timeout, expiration* is the value of Time in seconds when the assertion expires. When an assertion contains a *timeout* parameter (only in TPM2_PolicyTicket()), *timeout* is an implementation-dependent value.

A ticket contains a digest of the command parameters of the assertion along with a ticket *timeout*. As long as a ticket has not expired, its effect on a *policySession→policyDigest* and *policySession→timeout* will be the same as the TPM2_PolicySigned() or TPM2_PolicySecret() command that generated the ticket. For example, one may use a TPM2_PolicySigned() command with an expiration of -3600 (the negative of the number of seconds in an hour) to return a ticket. For the next hour, that ticket can be used with TPM2_PolicyTicket() to grant whatever other permissions were approved by the TPM2_PolicySigned() command.

When the TPM is not able to report the passage of time (*Time* does not advance), accurate timing of assertions is not possible. To prevent having a timed assertion persist past the intended timeout, a TPM is required to invalidate any time based assertion that was created before a discontinuity in the TPM's measurement of time. Such a discontinuity can occur when *Time* does not advance or when *Time* is reset. This requirement is met by having a number (a counter or a nonce) that changes each time that there is a time discontinuity (an epoch) and by including *timeEpoch* in the computation of time-based assertions. This implies that each policy session will need to:

 Copyright © TCG 2006-2020

- record *timeEpoch* when the session is created (in *policySession→timeEpoch*);
- validate that the *timeEpoch* associated with a time-limited assertion is the same as *policySession→timeEpoch* before the assertion is accepted; and
- when a time-limited policy is used for authorization, verify that the current TPM *timeEpoch* matches *policySession→timeEpoch.*

If a counter is used for *timeEpoch*, it needs to be saved in NV memory whenever it changes. If the number used for *timeEpoch* is a nonce, it can be kept in RAM and changed on each time discontinuity.

NOTE 6 A *timeEpoch* nonce needs to be large enough that a replay is infeasible. That is, a ticket issued with a given nonce should not be useable after a future power cycle because the nonce values happen to match. In the context of a specific ticket, a nonce collision is not a "birthday problem" as the nonce has to match exactly rather than being one of a group of values that are equivalent.

19.7.13 Use of TPM for authPolicy Computation

To use a policy for authorization for an object or NV Index, the creator of an object or NV Index is required to know, at the time of creation of the Object or NV Index, the digest of the policy. The computation of this policy requires duplication of the steps that would be performed by the TPM when it evaluates the policy and updates the accumulated *policyDigest* of the session.

This computation can be done by software but would require that the policy update process for each command be replicated by software. As an alternative, the TPM can be used to perform the computation.

To use the TPM, a policy session is created, and various policy commands are sent to the TPM as if the policy were being evaluated in order to authorize an action. TPM2_PolicyGetDigest() may then be used to read the final *policyDigest* from the TPM. That *policyDigest* value may then be used as the *authPolicy* parameter in a new Object/NV Index.

NOTE There is no requirement that the *authPolicy* for each Object or NV Index be unique.

If the policy is complex and uses TPM2_PolicyOR(), it will be necessary to compute multiple *policyDigest* values. The same policy session can be used for all of the computations by using TPM2_PolicyRestart() after the *policyDigest* for a branch is computed. When the last branch is computed, it may be used in a TPM2_PolicyOR that is constructed from the previously computed values.

TPM2_PolicyGetDigest() could also be used to help validate the software that is implementing the digest computation. The value computed by the TPM can be compared to the value computed by the software library to ensure that they are the same. If desired, TPM2_PolicyGetDigest() can be called after each policy command.

19.7.14 Trial Policy Session

If a policy requires a signed (symmetric or asymmetric) authorization for an action, that authorization may not be available at the time that the Object/NV Index is created, and, in fact, the authorizing entity might not be willing or able to provide the necessary authorization at the time of creation.

EXAMPLE 1 If the Object is to have a duplication authorization, the duplication authority may not provide the authorization for the duplication when the Object is created. If they did, then the migration policy could be computed; the *policyDigest* of the session read and placed in a new Object, and immediately used for duplication of the Object. The duplication authority may not want to allow the duplication at that time.

The TPM provides a special type of policy session to be used for the purpose of computing the policy without enabling the use of the policy. When a session is created by TPM2_StartAuthSession(*policyType*

 Copyright © TCG 2006-2020

= TPM_SE_TRIAL), a policy session is created that cannot be used for authorization. Since it cannot be used for authorization, authorizations are not needed in the computation of the policy.

EXAMPLE 2 If TPM2_PolicySigned() is called to update the digest of a trial policy session, the signature is not validated but the *policyDigest* is updated as if a correct signature was provided.

19.7.15 Use of TPM2_PolicySigned() and TPM2_PolicySecret() without *nonceTPM*

The primary purpose of including *nonceTPM* in TPM2_PolicySigned() and TPM2_PolicySecret() is to restrict the use of the policy assertion to a specific policy session so that the assertion may only be used once.

nonceTPM serves a different purpose when the assertion is structured so that a ticket is produced. In that case, the intent is that the assertion can apply to more than one policy session, so *nonceTPM* serves a different purpose – to associate the assertion with a specific TPM. That is, a non-zero expiration causes the TPM to produce a ticket. If the signer did not include *nonceTPM*, its signature could be used with any session and therefore on any TPM. Including *nonceTPM* binds the signature to a specific session, and thus a specific TPM. The signer forces the ticket to be created on a specific TPM, which ties the ticket to a timer on that TPM.

Each *nonceTPM* is expected to be statistically unique and not appear on any other TPM (this is just expected to be true and not required to be true). Therefore, when an assertion includes *nonceTPM,* the assertion will only be usable on one policy session on a specific TPM.

NOTE When a ticket is produced, that ticket is always restricted to use on a specific TPM because of the use of TPM- and hierarchy-specific proof values in the ticket computation.

When the policy assertion does not include *nonceTPM,* then is it possible to use the assertion with any policy session. For TPM2_PolicySecret() the assertion may still be associated with a specific TPM if the authorization for the *authObject* uses an HMAC or policy session. This is because that authorization session will be TPM specific. For TPM2_PolicySigned(), when there is no *nonceTPM* in the assertion, then the assertion may be used on any policy session on any TPM where the public part of *authObject* may be loaded (that is, any TPM that is compatible with this specification). This may be suitable when an assertion is limited to performing specific actions (through *cpHash*) or specific policies (though *policyRef*), but this capability should be used with caution.

19.8 Dictionary Attack Protection

19.8.1 Introduction

The TPM incorporates mechanisms that provide protection against guessing or exhaustive searches of authorization values stored within the TPM.

The dictionary attack (DA) protection logic is triggered when the rate of authorization failures is too high. If this occurs, the TPM enters Lockout mode in which the TPM will return TPM_RC_LOCKOUT for an operation that requires use of a DA protected *authValue*. Depending on the settings of the configurable parameters described below, the TPM can “self-heal” after a specified amount of time or be programmatically reset using proof of knowledge of an authorization value or satisfaction of a policy

The *authValue* for an object receives DA protection unless the object's *noDA* attribute is SET. The *authValue* for an NV Index receives DA protection unless the TPMA_NV_NO_DA attribute of the Index is SET. The *authValue* associated with a permanent entity, other than TPM_RH_LOCKOUT, does not receive DA protection. Sequence objects created by TPM2_HMAC_Start() and TPM2_HashSequenceStart() do not receive DA protection.

 Copyright © TCG 2006-2020

NOTE 1 Authorization values associated with permanent entities, other than TPM_RH_LOCKOUT, are expected to be high-entropy values that are managed by a computer or will be well-known values, In either case, they will not need DA protection. While it is safer when *lockoutAuth* is a high-entropy value, it is possible that *lockoutAuth* will be a value chosen to be remembered by a human which will likely have less entropy than other permanent entities. As a consequence, *lockoutAuth* is DA protected even though it is a permanent entity.

The reason for being able to exclude entities from DA protection is that lockout of all TPM use could make the system unstable. The OS may have uses for the TPM that should not be blocked due to authorization problems with keys associated with user-mode applications. The OS is expected to use a well-known or high-entropy *authValue* for any entities that it manages and an *authValue* of neither type needs DA-protection.

An *authValue* may be used for authorization in three ways:

1) a password;
2) the *authValue* parameter in the HMAC computation of equation (17); or
3) the *authValue* parameter in the computation of *sessionKey* for a bound session as shown in equation (18).

All uses of a DA protected *authValue* receive DA protection.

NOTE 2 A TPM PIN Index provides a type of DA protection for an individual TPM entity. This is described in37.2.8.

19.8.2 Lockout Mode Configuration Parameters

The TPM uses four, 32-bit, non-volatile state variables to control the initiation and recovery from the DA-lockout mode.

NOTE The "NV" notation indicates that these values are required to be held in persistent memory and be updated in NV when they change

a) ***failedTries*** (NV) – This counter is incremented when the TPM returns TPM_RC_AUTH_FAIL. TPM2_Clear() will reset this counter to zero. This counter is also set to zero on a successful invocation of TPM2_DictionaryAttackLockReset(). This counter is decremented by one after *recoveryTime* seconds if:
   1) the TPM does not record an authorization failure of a DA-protected entity,
   2) there is no power interruption, and
   3) *failedTries* is not zero.

NOTE 1 If the TPM has a trusted source of time that runs when TPM power is lost, then *failedTries* may be reduced when power is restored. The amount that *failedTries* is decremented would be dependent on the duration of the power loss and the value of *recoveryTime*.

NOTE 2 The TPM may keep track of the time elapsed toward *recoveryTime* at shutdown and use that against the *recoveryTIme* upon power up.

b) ***maxTries*** (NV) – The TPM is in Lockout mode as long as *failedTries* equals this value. When a new owner is installed, *maxTries* is set to its default value as specified in the relevant platform-specific specification.

c) ***recoveryTime*** (NV) – This value indicates, in seconds, the rate at which *failedTries* is decremented. This can be set to a large value (2^{32} - 1) which essentially inhibits automatic exit from Lockout mode. When a new owner is installed, this value is set to its default value as specified in the relevant platform-specific specification.

d) ***lockoutRecovery*** (NV) – This value indicates the delay in seconds between attempts to use *lockoutAuth*. The time delay only applies after an authorization failure using *lockoutAuth*. A value of zero indicates that a system reboot (TPM2_Startup(TPM_SU_CLEAR)) is required between lockout attempts.

The parameters *maxTries*, *recoveryTime*, and *lockoutRecovery* are set with TPM2_DictionaryAttackParameters(). This command requires Lockout Authorization.

19.8.3 Lockout Mode

The TPM is in Lockout mode while failedTries is equal to *maxTries*. While in Lockout mode, any use of a DA-protected *authValue* will return TPM_RC_LOCKOUT.

NOTE 1 An exception is that TPM2_DictionaryAttackLockReset() is allowed to execute even though *lockoutAuth* is DA protected.

NOTE 2 If there is an authorization failure that does not increment *failedTries*, the TPM returns TPM_RC_BAD_AUTH

An authorization failure may occur with a password or an HMAC. For a policy authorization, the policy is validated before the HMAC is computed. If the policy fails, the TPM returns TPM_RC_POLICY to indicate that dictionary attack protection was not involved.

NOTE 3 A policy authorization does not always have an associated HMAC.

19.8.4 Recovering from Lockout Mode

The TPM can recover from Lockout mode in three ways.

1) TPM2_DictionaryAttackLockReset() sets *failedTries* to zero. This command requires Lockout Authorization. The TPM does not have to be in Lockout mode in order to use this command.
2) The TPM decrements *failedTries* by one if no TPM resets are recorded during *recoveryTime*.

NOTE 1 If the TPM is in Lockout mode, then the TPM will always leave Lockout mode when *failedTries* decrements because *failedTries* will no longer be equal to *maxTries*.

NOTE 2 The failure count is not decremented below zero.

3) *failedTries* is set to zero if the owner changes.

Configuration and programmatic recovery of the dictionary attack logic requires proof of knowledge of Lockout Authorization. When the TPM owner is changed by changing the SPS, *lockoutAuth* is set to the EmptyAuth and *lockoutPolicy* is set to the Empty Buffer

TPM2_DictionaryAttackLockReset() allows external software to reset the dictionary attack protection logic by providing Lockout Authorization. This command can be used when the TPM is in Lockout mode.

19.8.5 Authorization Failures Involving *lockoutAuth*

When *lockoutAuth* is used in an authorization and that authorization fails, the TPM enters a lockout state intended to provide special protection for the *lockoutAuth* value. An authorization failure associated with *lockoutAuth* causes the TPM to enter this special lockout state regardless of the setting of *failedTries* and *maxTries*.

When in this special lockout state, the TPM will not allow use of *lockoutAuth*. The TPM will exit this state when TPM2_DictionaryAttackLockReset() is used with a successful lockoutPolicy or after the TPM is

 Copyright © TCG 2006-2020

powered for a configurable time period (*lockoutRecovery*). If *lockoutRecovery* is set to zero, then the TPM will not exit this state until the next TPM2_Startup() or until lockoutPolicy is used.

NOTE The design depends upon the trusted computing base to filter commands to the TPM such as TPM2_DictionaryAttackLockReset(). This prevents a rogue application from completing a denial of service attack on the TPM by intentionally sending the command with a bad *lockoutAuth*.

19.8.6 Non-orderly Shutdown

A TPM may be implemented such that the command execution unit does not always have access to NV memory (see 37.7.2). For such an implementation, it may not be possible to increment the NV copy of *failedTries* when the authorization failure occurs. When the failure occurs, the TPM will return TPM_RC_AUTH_FAIL and, until the NV version of *failedTries* is updated, the TPM will be in lockout.

It is possible that the TPM will be reset when a write to the NV version of *failedTries* is pending. If the TPM did not handle this special case, then an attacker could try an authorization for a DA protected object when NV writes are not possible. When the TPM restarted, the failed attempt would not be recorded, and the attacker could try again.

To prevent this type of attack, at TPM2_Startup(), the TPM checks if it is starting after an orderly shutdown. If not, and *failedTries* is not already equal to *maxTries*, then the TPM will increment *failedTries* by one.

NOTE This check and increment of *failedTries* may not be necessary if it is impractical for an attacker to prevent update of the NV version of *failedTries*.

An alternative implementation sets an NV flag indicating that access to a DA protected object occurred during this boot cycle. After a non-orderly restart, if the flag is set, the TPM increments *failedTries* and clears the flag. If the flag is clear, there is no need to increment *failedTries*.

EXAMPLE This handles the case where a platform repeatedly does a non-orderly shutdown, possibly due to a low battery. Without the flag, *failedTries* would increment on each reboot and the TPM would go into lockout.

19.8.7 Justification for Lockout Due to Session Binding

When a bound session is created, the caller does not have to prove knowledge of the *authValue* of the bind object. The *authValue* is used in the creation of the *sessionKey* and if the caller does not know the *authValue*, they will not be able to compute the correct *sessionKey* and use the authorization session.

A bound authorization session may be used to authorize actions on another object. If that object does not have DA protection, then an attacker could use binding to circumvent DA protection on the bind object.

The attack is as follows:

a) An attacker creates an object (D) that has no DA protection and *authValue* known to the attacker.
b) An attacker guesses a possible *authValue* for a DA protected object (object A).
c) The attacker uses object A as the *bind* object in TPM2_StartAuthSession() to create a session (S).
d) The attacker uses session S to authorize an action on object D.
e) If the authorization fails, the attacker goes to step b) and tries a new value.

By retaining the DA state of object A in the session state, the attack is prevented. When the session is used for authorization, the authorization failure counter (*failedTries*) is incremented if either the entity being authorized is subject to DA protection or if the session is bound to an entity that has DA protection.

 Copyright © TCG 2006-2020

NOTE If a session is bound to a permanent entity other than TPM_RH_LOCKOUT, then the session is not bound to an entity that has DA protection.

19.8.8 Sample Configurations for Lockout Parameters

19.8.8.1 Introduction

Two common configurations are anticipated: one for enterprise-managed TPMs, and one for home users.

NOTE It is anticipated that the operating system will layer additional anti-hammering protection atop that provided by the TPM so that it is unlikely that one OS user will be able to interfere with the actions of another user or the trusted computing base (TCB).

19.8.8.2 Enterprise Use

In this use, it is expected that the TPM owner will set the *lockoutAuth* to a high-entropy value that is held in a database and set the *lockoutRecovery* to a small, non-zero value, such as one. The enterprise will use this value to recover the TPM when suitable non-automated validation procedures have been performed.

The enterprise would likely set *maxTries* to a relatively low value (such as, 10).

For a server or data center, the *recoveryTime* would be set to a large value (such as, 2^{32} -1) implying manual recovery. For laptops, a setting of a few hours would provide adequate protection for PINs.

19.8.8.3 Home or Unmanaged Use

In this application, the *lockoutAuth* may be set to a random, high-entropy value that is then erased so that programmatic lockout recovery is not possible. *maxTries* and *recoveryTime* can be set to balance security and convenience.

NOTE If this configuration is used, the only way to execute TPM2_Clear() to change the owner is to use Platform Authorization.

 Copyright © TCG 2006-2020

20 Audit Session

20.1 Introduction

An audit session allows for the auditing of a selected sequence of commands so that evidence may be provided that the commands were executed.

Any HMAC authorization session may be designated for auditing but only one session may be used for audit in each command. A session is designated for auditing by setting the *audit* attribute in the session.

When a session is first used as an audit session, the TPM will initialize the audit hash for the audit session. The initialization value is a Zero Digest with the number of octets determined by the hash algorithm of the session.

If the session was bound when created (see 19.6.10 and 19.6.12), the bind value is lost and any further use of the session for authorization will require that the *authValue* be used in the HMAC.

Since the first use of an audit session may cause the size of the session context to change, the command may fail due to insufficient memory. TPM-management software may save other session contexts and retry the command.

NOTE 1 The TPM is required to have sufficient memory to allow three sessions to be simultaneously loaded, one of which may be an audit session.

For all commands using a session tagged as audit (including the initial use), if the command completes successfully, the *cpHash* and the *rpHash* are Extended to the audit session digest. When a command fails, the audit session digest is not changed and, as is normal in the case of a command failure, the sessions are not included in the response and session nonces are not updated.

The equation for updating the audit session digest is:

$$auditDigest_{new} \coloneqq \mathbf{H}_{auditAlg}\left(auditDigest_{old} \,||\, cpHash \,||\, rpHash\right) \tag{30}$$

The hash algorithm is the algorithm designated in TPM2_StartAuthSession().

NOTE 2: Audit within an encrypted session will record the encrypted cpHash and/or rpHash, which is unlikely to be useful at the application level.

Unless a command description indicates that no sessions are allowed, an audit session may be used with any command. A command may have only one audit session.

An audit session uses the same session format as other HMAC-based sessions. The method of computing the HMAC differs in that, if the audit session is not associated with any object handle, no *authValue* is available for use in the authorization HMAC. All HMAC computations for an audit session will set *authValue* to an Empty Buffer.

NOTE 3 If the *sessionKey* is also an Empty Buffer, then no HMAC computation is performed and the *hmac* parameter of the session structure will be an Empty Buffer.

For commands that do not require authorization, a bound or salted audit session causes an HMAC based on a shared secret to be generated. This provides assurance that the command was executed on the TPM. A bound session allows association with a known *authValue* in a TPM, which can provide assurance that the commands being audited are actually associated with a specific TPM. However, if others know the *authValue*, then the unsalted audit session may have the same association issue as the unbound session. A salted session can be associated with a key that is known to be TPM-resident, so the audit based on a salted session can be reliably associated with a specific TPM.

 Copyright © TCG 2006-2020

NOTE 4 This assurance does not require a signed audit digest to be used.

EXAMPLE TPM2_PCR_Read does not require authorization. A bound or salted audit session will cause an HMAC to be used, and thus provide integrity for the command and response.

20.2 Exclusive Audit Sessions

An exclusive audit session permits the TPM to prove that a series of commands were executed with no intervening commands that were not audited by the exclusive audit session.

In a response, the *auditExclusive* attribute of an audit session will indicate if the TPM has executed any commands that were not audited by the session. If there was another user of the TPM, the *auditExclusive* attribute will be CLEAR. If not, the audit session is exclusive and the *auditExclusive* attribute will be SET.

The TPM keeps track of the current exclusive session. At most, one active session may have the *auditExclusive* status. A session becomes the current exclusive audit session when it is first used as an audit session, regardless of the setting of *auditReset*. It may also become the current exclusive audit session if the *auditReset* attribute of the session is SET in the command. In the response, the *auditExclusive* attribute of the session will be SET and the session is exclusive. The session is no longer the current exclusive audit session if it is flushed (TPM2_FlushContext()) or if an auditable command is executed that does not use the current exclusive audit session.

NOTE 1 *auditReset* may only be SET if *audit* is also SET.

A command that is not allowed to have any sessions will not change the current exclusive audit session. Those commands include the context management commands (TPM2_ContextSave(), TPM2_ContextLoad(), and TPM2_Flush()), TPM2_Startup(), and TPM2_ReadClock().

NOTE 2 It is the responsibility of the TCG Software Stack (TSS) or other controlling software to preserve the integrity of the exclusive audit session. As the purpose of the exclusive audit session is to show that no other commands were executed during the session, the expectation is that the controlling software would limit access to the TPM to prevent any other uses of the TPM.

20.3 Command Gating Based on Exclusivity

If the *auditExclusive* attribute of an audit session is SET in the command, then the TPM will return TPM_RC_EXCLUSIVE if the audit session is not the current exclusive audit session.

NOTE 1 As with other error returns, no change is made to the state of the session and it remains active.

NOTE 2 *auditExclusive* in a command only determines whether the command is executed. It does not affect the exclusive status of the session.

NOTE 3 *auditExclusive* may only be SET if audit is also SET.

20.4 Audit Session Reporting

The audit status of an audit session can be determined with TPM2_GetSessionAuditDigest(). This command returns a data structure that includes the audit session digest. If the handle for the signing key is not TPM_RH_NULL, the TPM returns a signature over the data structure.

In the atypical case where the TPM2_GetSessionAuditDigest() sessionHandle is the same as the handle of the audit session, because the audit digest is signed before the audit digest is updated, the *cpHash* and *rpHash* for a TPM2_GetSessionAuditDigest() is not included in the audit digest of the signed data structure. Possession of the audit digest is proof that the command executed. However, the *cpHash* and *rpHash* of TPM2_GetSessionAuditDigest() will be included in subsequent audits if the audit session remains active.

 Copyright © TCG 2006-2020

TPM2_GetSessionAuditDigest() requires that the indicated session be an audit session and will return TPM_RC_TYPE if it is not. The TPM does not change internal state unless the command actions complete successfully. This means that a session cannot become an audit session unless the command in which it is designated as an audit session completes successfully. From this we can conclude that a session cannot be designated as being an audit session in a TPM2_GetSessionAuditDigest() in which the same session is the audited session.

20.5 Audit Establishment Failures

If a command is the first use of a session as an audit session, and the command fails, then the state of a session as an audit session will not change. This means that, if a session was not an audit session before the command was executed, it will not be an audit session after the command fails. If a session was an audit session before the command was executed, it will be an audit session after the command fails.

If a command fails, then the exclusive status of sessions does not change. A session that was exclusive before the command failure is exclusive after the command failure.

20.6 Audit Alternative

Both TPM2_GetSessionAuditDigest() and TPM2_GetCommandAuditDigest() require Endorsement Authorization. If an application does not have Endorsement Authorization, it can still obtain proof that a command was run on a particular TPM. The application must have a *fixedTPM* asymmetric encryption key that is trusted to be on the TPM. This key would have similar trust properties to the signing key that would be used with the TPM2_GetSessionAuditDigest() and TPM2_GetCommandAuditDigest() commands. The application uses an audit session that is a salted session with the trusted key specified as *tpmKey*. The salt forces an HMAC session. The HMAC verification is proof that the command was run on that TPM.

 Copyright © TCG 2006-2020

21 Session-based encryption

21.1 Introduction

Several commands have parameters that may need to be encrypted going to or from the TPM. An example is the authorization data that is passed to the TPM when an object is created or when the authorization value is changed. Session-based encryption may be used to ensure confidentiality of these parameters.

Not all commands support parameter encryption. If session-based encryption is allowed, only the first parameter in the parameter area of a request or response may be encrypted. That parameter must have an explicit size field. Only the data portion of the parameter is encrypted. The TPM should support session-based encryption using XOR obfuscation. Support for a block cipher using CFB mode is platform specific. These two encryption methods (XOR and CFB) do not require that the data be padded for encryption, so the encrypted data size and the plain-text data size is the same.

If the symmetric algorithm is TPM_ALG_NULL and encryption or decryption is specified, the TPM returns TPM_RC_SYMMETRIC.

Any first parameter may be encrypted as long as the parameter has a size field.

Session-based encryption uses the algorithm parameters established when the session is started and values that are derived from the session-specific *sessionKey*. The encryption values are created in a way that is dependent on both the session type and the session encryption parameters.

If a session is also being used for authorization, *sessionValue* (see 21.2 and 21.3) is *sessionKey* || *authValue*. The binding of the session is ignored. If the session is not being used for authorization, *sessionValue* is *sessionKey*.

NOTE 1 A policy session that is used for parameter encryption uses *authValue* to calculate *sessionValue* even if the policy does not include TPM2_PolicyAuthValue().

If *sessionAttributes.decrypt* is SET in a session in a command, and the first parameter of the command is a sized buffer, then that parameter is encrypted using the encryption parameters of the session. If *sessionAttributes.encrypt* is SET in a session of a command, and the first parameter of the response is a sized buffer, then the TPM will encrypt that parameter using the encryption parameters of the session. The *encrypt* attribute may only be SET in one session that is used in a command and the *decrypt* attribute may only be SET in one session per command. The attributes may be SET in different sessions or in the same session.

Parameters in commands are encrypted before any *cpHash* is computed. Parameters in responses are encrypted before any *rpHash* is computed.

The parameter data buffer is protected with either XOR obfuscation or CFB mode encryption. The size field of the parameter is not protected.

When a command/response with an encrypted parameter is received, the *cpHash*/*rpHash* is computed as required for the sessions before the parameter is decrypted.

NOTE 2 The caller may obfuscate the true size of an authorization value by adding octets of zero to the end. The extra octets of zero will have no impact on the authorization computations and may be discarded by the TPM.

The two methods of session-based encryption used in this specification are, by themselves, malleable. That is, an attacker can make a controlled change (bit reversal) in the encrypted data that will result in an

 Copyright © TCG 2006-2020

identical change in the decrypted data. This kind of attack is mitigated by the HMAC authorization session verification.

21.2 XOR Parameter Obfuscation

For session-based obfuscation using **XOR**(), the operation is:

$$\mathbf{XOR}(parameter, hashAlg, sessionValue, nonceNewer, nonceOlder) \quad (31)$$

where

parameter	a variable sized buffer containing the parameter to be obfuscated
hashAlg	the hash algorithm associated with the session
sessionValue	see 21.1
nonceNewer	for commands, this will be nonceCaller and for responses it will be nonceTPM
nonceOlder	for commands, this will be *nonceTPM* and for responses it will be *nonceCaller*

NOTE 1 The **XOR**() function is defined in 11.4.7.3.

NOTE 2 The obfuscated size of parameter is the same as the size of the underlying parameter. That is, if a TPM2B_CREATE is obfuscated, the size of the obfuscated data is the same as the size of the data.

21.3 CFB Mode Parameter Encryption

When session-based encryption uses a symmetric block cipher, an encryption key and IV will be generated from:

$$\mathbf{KDFa}\ (hashAlg, sessionValue, \text{“CFB”}, nonceNewer, nonceOlder, bits) \quad (32)$$

where

hashAlg	the hash algorithm associated with the session
sessionValue	see 21.1
“CFB”	label to differentiate use of **KDFa**() (see 4.2)
nonceNewer	*nonceCaller* for a command and *nonceTPM* for a response
nonceOlder	*nonceTPM* for a command and *nonceCaller* for a response
bits	the number of bits required for the symmetric key plus an IV

NOTE 1 The IV size is equal to the block size of the cipher.

The most significant octets of the returned value are used as the encryption key and the remaining octets are used as the IV. The number of octets used for the encryption key and for the IV is dependent on the algorithm parameters of the session.

EXAMPLE For AES, the block size is 16 octets regardless of the key size. If the key size were 256 bits (32 octets), then, in the call to **KDFa** (), *bits* would be set to 48*8. The most significant 32 octets of the returned value would be used as the key for the encryption and the next 16 octets would be used for the IV.

 Copyright © TCG 2006-2020

NOTE 2 If the key size is not an even multiple of 8 bits, the first N octets of the returned value will contain the key and the remaining octets the IV. N is the smallest integer such that (N * 8) ≥ the key size.

The data portion of the parameter is then encrypted using the symmetric key and the symmetric block cipher algorithm associated with the session.

 Copyright © TCG 2006-2020

22 Protected Storage

22.1 Introduction

When a Protected Object is in the TPM, it is in a Shielded Location because the only access to the context of the object is with a Protected Capability (a TPM command). The size of TPM memory may be limited and if the only storage for Protected Objects were the TPM Shielded Locations, the TPM's usefulness would be reduced. The effective memory of the TPM is expanded by using cryptographic methods for Protected Objects when they are not in Shielded Locations.

22.2 Object Protections

The cryptographic protections for a Protected Object include encryption to prevent disclosure of the confidential contents, and an integrity check to allow detection of modifications to the externally stored Protected Object. The integrity check detects modifications to either the confidential or the non-confidential portions of the Protected Object.

The integrity value is computed over the encrypted data. If the integrity check fails, then symmetric decryption will not occur. Since the integrity value contains the digest of the associated public area (its Name), the confidential contents of the Protected Object will not be decrypted if they are not properly paired with a public area.

22.3 Protection Values

The protection of a sensitive area uses two keys. These values are created from a secret value associated with an object's Storage Parent. One of the keys is used as an HMAC key and the second is a symmetric encryption key.

A seed value is used in the generation of the symmetric encryption key and the HMAC key. The source of the seed is dependent on the situation. If the protections are for an object in a hierarchy, the seed is the *seedValue* in the Storage Parent's sensitive area. If the protections are for a duplication blob, the seed is derived from a shared secret that is protected using asymmetric methods of the new parent. The algorithm-specific annexes contain the formulations for deriving the seed when asymmetric protections are used.

To produce the symmetric key, the seed value and object Name are used in **KDFa**() as shown in equation (33). This method is used when a symmetric key is generated for the protection of sensitive areas attached to a hierarchy or sensitive data in a duplication blob (see 23.3).

NOTE 1 This method is also used to generate the symmetric key used for the protection of credential values (see 24.4).

To produce the HMAC key, the seed is used in **KDFa**() as shown in equation (35). This method is used when an HMAC is used to protect the integrity of a sensitive area attached to a hierarchy or for sensitive data in a duplication blob.

NOTE 2 This method is also used to generate the HMAC key for credential values (see 24.4).

When performing symmetric encryption, an IV of zero is used unless the same symmetric key is used multiple times. The same symmetric key is used each time that the sensitive area of a child changes due to TPM2_ObjectChangeAuth(). For encryption of a child, a random IV is generated by the TPM each time it performs the encryption.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

22.4 Symmetric Encryption

A symmetric key is used to encrypt the sensitive area of an object that was created by TPM2_Create() or imported by TPM2_Import(). It is also used when re-encrypting a sensitive area when the authorization value is changed (TPM2_ObjectChangeAuth()). The symmetric key is derived from a seed value contained in the Storage Parent's sensitive area and the Name of the protected object.

The block cipher used for encrypting the object's sensitive area is the symmetric cipher of the Storage Parent.

The symmetric key for the encryption is computed by:

$$symKey \coloneqq \mathbf{KDFa}\ (pNameAlg, seedValue, \text{“STORAGE”}, name, \text{NULL}\ , bits) \tag{33}$$

where

$pNameAlg$	*nameAlg* of the object's Storage Parent
$seedValue$	symmetric seed value in the sensitive area of the object's Storage Parent (see 27.7.4)
“STORAGE”	a value used to differentiate the uses of the KDF
$name$	Name of the object being encrypted / decrypted
$bits$	number of bits required for the symmetric key

When a $symKey$ is being used to protect the sensitive area of a child object, the TPM will create a random IV value ($symIv$) that is the size of an encryption block of the symmetric algorithm. This $symIV$ is included in the private area and in the HMAC computation of the sensitive area. A *symIV* of zero is used when encrypting the sensitive area for duplication or a credential to be used in TPM2_ActivateCredential().

The $symKey$ and $symIv$ are used to encrypt the sensitive data.

$$encSensitive \coloneqq \mathbf{CFB}_{pSymAlg}\,(symKey, symIv, sensitive) \tag{34}$$

where

$\mathbf{CFB}_{pSymAlg}$	symmetric encryption in CFB mode using the symmetric algorithm of the Storage Parent
$symKey$	symmetric key from (33)
$symIv$	IV from RNG or 0
$sensitive$	a TPM2B_SENSITIVE containing the sensitive area structure being protected

NOTE The *size* and *buffer* fields of *sensitive* are encrypted.

After the data is encrypted, the TPM2B_IV containing the random $symIv$ is placed in front of the encrypted data in preparation for the integrity computation. If the *symIV* was zero, then no value is added to the encrypted data.

22.5 Integrity

The HMAC key ($HMACkey$) for the integrity is computed by:

 Copyright © TCG 2006-2020

$$HMACkey \coloneqq \mathbf{KDFa}\ (pNameAlg, seedValue, \text{“INTEGRITY”}, \text{NULL}, \text{NULL}, bits) \quad (35)$$

where

$pNameAlg$	the *nameAlg* of the object's Storage Parent
$seedValue$	the symmetric seed value in the sensitive area of the object's Storage Parent (see 27.7.4)
“INTEGRITY”	a value used to differentiate the uses of the KDF
$bits$	the number of bits in the digest produced by *pNameAlg*

$HMACkey$ is then used in the integrity computation.

An HMAC is performed over the $symIV$ and the $encSensitive$ produced in (34).

NOTE 1 This is called an *outerHMAC* because it is the same HMAC process that is used when an object is duplicated. The duplication can produce an inner and an outer HMAC.

$$outerHMAC \coloneqq \mathbf{HMAC}_{pNameAlg}\ (HMACkey, symIv \mid\mid encSensitive \mid\mid name.buffer) \quad (36)$$

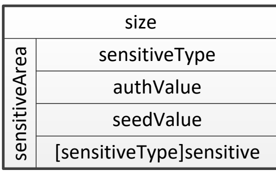
where

$\mathbf{HMAC}_{pNameAlg}$	the HMAC function using *nameAlg* of the object's Storage Parent
$HMACkey$	a value derived from the Storage Parent’s symmetric protection value (*seedValue*) according to equation (35)
$symIv$	a marshaled TPM2B_IV containing the symmetric IV value used in (34). Both the size and buffer fields are included in the HMAC
$encSensitive$	encrypted TPM2B_SENSITIVE produced in (34); after encryption, the size and buffer fields are not separable
$name.buffer$	the Name of the object being protected (does not include a size field)

The integrity value is placed before the symmetric IV.

NOTE 2 Placement of the integrity value at the beginning of the sensitive area in preparation simplifies the process of finding the integrity value when the protected data contains variable-sized elements.

NOTE 3 Inclusion of the Name ensures that the sensitive area is associated with the correct public area.

<table>
<tr><td>Marshal the sensitive area into a TPM2B_SENSITIVE</td><td>
</td></tr>
<tr><td colspan="2">Create a symmetric key and IV for encryption:

$symKey \coloneqq \mathbf{KDFa}\ (pNameAlg, seed, \text{“STORAGE”}, name, NULL, bits)$

$symIV \coloneqq$ bits from the RNG</td></tr>
</table>

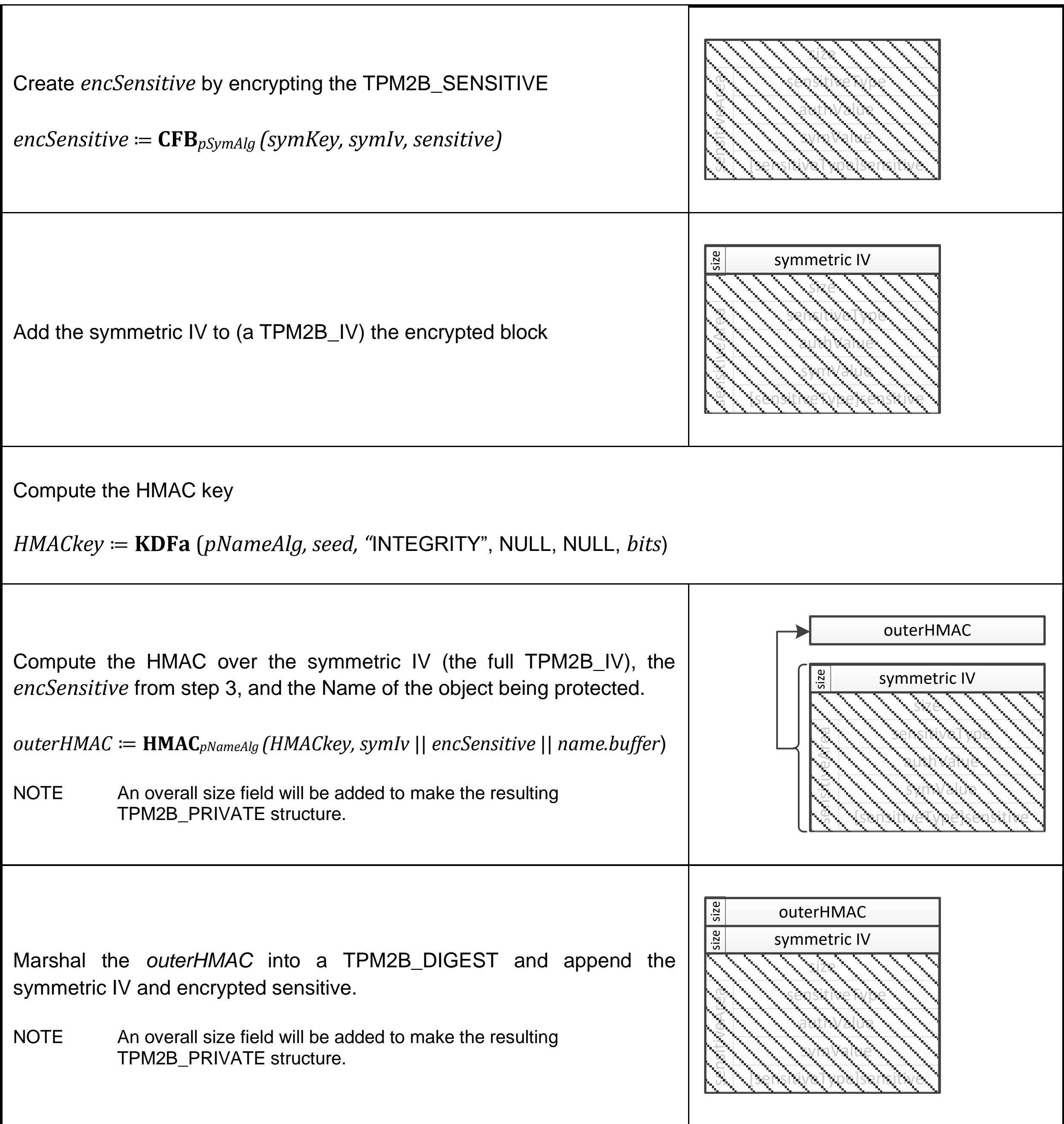


Figure 17 — Creating a Private Structure

23 Protected Storage Hierarchy

23.1 Introduction

The TPM supports the creation of hierarchies of Protected Locations. A hierarchy is constructed with Storage Keys as the connectors to which other types of objects (keys, data, and other connectors) may be attached.

The hierarchical relationship of objects allows segregation of objects based on the system-operating environment (established by PCR or authorizations) as well as simplifying the management of groups of related objects.

23.2 Hierarchical Relationship between Objects

A hierarchy is rooted in a secret seed key, kept in the TPM. To create a hierarchy of keys, the seed key (Primary Seed) is used to generate a key that uses a specific set of algorithms. If this key is a restricted decryption key, then it is a Parent Key. If it is not a Derivation Parent, then it is a Storage Parent under which other objects may be created or attached.

A Storage Parent provides protection for the sensitive area in another object (a child) when that object is stored outside of the TPM. Protection is provided by symmetric encryption and HMAC-based integrity protection of the sensitive area. There are two different cases for sensitive area encryption: storage and duplication.

When an Ordinary Object is created (TPM2_Create() or TPM2_CreateLoaded()) the keys used for protection of the Object's sensitive area are derived from a seed value (*seedValue*) in the sensitive area of the Storage Key. When an Object is prepared for duplication, its sensitive area is protected by a random key and a form of Diffie-Hellman is used to convey the key to the duplication target.

The objects in a hierarchy have a parent-child relationship. A Storage Key that is protecting other objects is a Storage Parent and the objects that it is protecting are its children. The ancestors of an object are the parent keys that connect the object to a TPM Primary Seed. Descendants of a key are all the objects that have that Parent as an ancestor. Unless it is intended to be used as a parent, a child object may be of any type.

A Derived Object is a child of its Derivation Parent. Both Primary Objects and Derived Objects are derived from seed values. For a Primary Object, the seed value is a Primary Seed and for a Derivation Object the seed value is the secret seed value in the Derivation Parent.

The sensitive part of an object created from a seed is not stored off of the TPM except in a context blobs (see clause 30). This means that the seed used to create the Object is not also used to generate protection keys for the Object. When an Object is duplicated, its sensitive area is protected by a random key, so the creation seed is not put at risk by the duplication process (see 23.3).

 Copyright © TCG 2006-2020

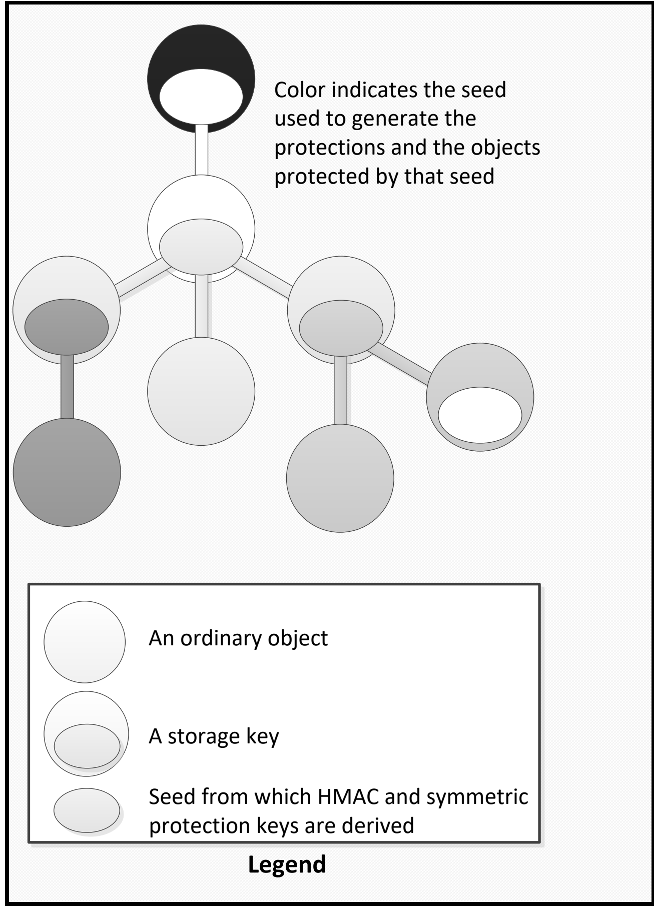


Figure 18 — Symmetric Protection of Hierarchy

There are two classes of Storage Keys: asymmetric and symmetric. All Storage Keys contain a symmetric protection key. An asymmetric Storage Key has a public identity that can be used as the target of an identity-based or secret-based duplication operation. An object that is a symmetric block cipher Object may also be a Storage Key, but may only be the target of secret-based duplication

23.3 Duplication

23.3.1 Definition

Duplication is the process of allowing an object to be a child of additional Storage Parent keys. The new parent (NP) may be in a hierarchy of the same TPM or of a different TPM.

The creator of an object controls the duplication process by selecting the duplication policy for the object.

Authorization for duplication requires a policy session. The policy sequence is required to have a command that causes the *commandCode* value of the policy context to be set to TPM_CC_Duplicate. This enables the DUP role of the policy, which is a requirement for duplication.

Duplication occurs on a loaded object and produces a new, sensitive structure that is encrypted using the methods of the NP. This new sensitive structure may not be used until TPM2_Import() has been executed to convert the object from "external" to "internal" protections.

NOTE 1 External protections use both asymmetric and symmetric cryptography, whereas the internal protections only use symmetric cryptography.

 Copyright © TCG 2006-2020

When an Object is duplicated, its sensitive area may be protected with an outer wrapper, an inner wrapper, or both. The outer wrapper uses Diffie-Hellman based on asymmetric keys and provides identity-based duplication. The inner wrapper uses a symmetric key that is under control of the duplication authority for the Object. Duplication using an inner wrapper is secret-based duplication.

NOTE 2 The duplication authority is the entity that controls the conditions under which an Object may be duplicated.

23.3.2 Protections

23.3.2.1 Introduction

In TPM2_Duplicate(), the caller may specify that the object should be protected with an inner, symmetric encryption. That is, the sensitive area is symmetrically encrypted before it is asymmetrically encrypted using the methods of the NP. If a symmetric inner wrapper is desired, the caller may provide a key or allow the TPM to generate the key.

If the *encryptedDuplication* attribute is SET in the object being duplicated, then it is required that the object have an inner wrapper and that the new parent not be TPM_RH_NULL. For such an object, the TPM will return an error (TPM_RC_SYMMETRIC) if the *symmetricAlg* parameter in TPM2_Duplicate() is TPM_ALG_NULL and TPM_RC_HIERARCHY if the *newParentHandle* parameter is TPM_RH_NULL.

Creation of a duplicate object uses two encryption phases. The first is used to apply an inner wrapper and the second is to encrypt using the algorithms of the NP.

The *encryptedDuplication* attribute of all objects in a duplication group are required to have the same setting. When an object is created with the *fixedParent* attribute CLEAR, then the *encryptedDuplication* attribute may be SET or CLEAR if the *fixedTPM* attribute is SET in the Storage Parent. If the *fixedTPM* attribute of a Storage Parent is not SET, then the *encryptedDuplication* attribute is required to be the same in all descendant objects of that Storage Parent.

23.3.2.2 Inner Duplication Wrapper

For the first phase, the TPM computes an integrity hash over the sensitive data. This hash includes the Name of the public area associated with this object.

$$innerIntegrity \coloneqq \mathbf{H}_{nameAlg}(sensitive \mid\mid name) \tag{37}$$

where

$\mathbf{H}_{nameAlg}$	hash function using the *nameAlg* of the object
$sensitive$	a TPM2B_SENSITIVE
$name$	the Name of the object being protected

A TPM2B_DIGEST containing the integrity digest value is prepended to the sensitive area and the buffer (integrity plus sensitive) is encrypted using CFB.

$$encSensitive \coloneqq \mathbf{CFB}_{pSymAlg}(symKey, 0, innerIntegrity \mid\mid sensitive) \tag{38}$$

where

$\mathbf{CFB}_{symAlg}$	symmetric encryption in CFB mode using the algorithm specified in the command

 Copyright © TCG 2006-2020

$symKey$	*encryptionKeyIn* parameter in TPM2_Duplicate() or a value from the RNG if no key is provided
$innerIntegrity$	value from (37)
$sensitive$	the sensitive value used in (37)

If no inner wrapper is specified, no integrity value is computed, and no encryption occurs in this first phase and

$$encSensitive \coloneqq sensitive \tag{39}$$

23.3.2.3 Outer Duplication Wrapper

In the second phase, the $encSensitive$ produced by phase 1 is encrypted and integrity checked using processes similar to those defined in clause 22. However, the seed from which the protection keys are derived is protected by the asymmetric algorithm of the NP. The method of generating *seed* is determined by the asymmetric algorithm of the NP. The different methods are described in annexes to this TPM 2.0 Part 1, for example, B.10.3 or C.6.3. The seed is selected prior to integrity generation for *encSensitive* or encryption of *encSensitive*.

NOTE For an RSA new parent, *seed* is not allowed to be larger than the size of the digest produced by the *nameAlg* of the object. When the TPM creates *seed*, it will be exactly the size of the *nameAlg* of the new parent.

Given a value for $seed$, a symmetric encryption key ($symKey$) is created by:

$$symKey \coloneqq \mathbf{KDFa}\,(npNameAlg, seed, \text{“STORAGE”}, Name, \text{NULL}, bits) \tag{40}$$

where

$npNameAlg$	the *nameAlg* of the new parent
$seed$	the symmetric seed value
“STORAGE”	a value used to differentiate the uses of the KDF
$Name$	the Name of the object being encrypted or decrypted
$bits$	the number of bits required for the symmetric key

The $symKey$ is used to encrypt the $encSensitive$.

$$dupSensitive \coloneqq \mathbf{CFB}_{npSymAlg}\,(symKey, 0, encSensitive) \tag{41}$$

where

$\mathbf{CFB}_{npSymAlg}$	symmetric encryption in CFB mode using the algorithm of the new parent
$symKey$	symmetric key from (40)
$encSensitive$	value from either (38) or (39)

 Copyright © TCG 2006-2020

Next, an HMAC key is generated from seed:

$$HMACkey \coloneqq \mathbf{KDFa}\,(npNameAlg, seed, \text{“INTEGRITY”}, \text{NULL}, \text{NULL}, bits) \tag{42}$$

where

$npNameAlg$	the *nameAlg* of the object's new parent
$seed$	the symmetric seed value used in (40)
“INTEGRITY”	a value used to differentiate the uses of the KDF.
$bits$	the number of bits in the digest produced by *npNameAlg*

An HMAC is then generated over the *dupSensitive* data. The Name of the associated public area is included in the HMAC computation to ensure that the sensitive area will only be decrypted when the proper public and private areas are used in TPM2_Import().

$$outerHMAC \coloneqq \mathbf{HMAC}_{npNameAlg}\,(HMACkey, dupSensitive\ ||\ Name) \tag{43}$$

where

$\mathbf{HMAC}_{npNameAlg}$	the HMAC function using nameAlg of the new parent
$HMACkey$	a value derived from the new parent symmetric protection value according to equation (42)
$dupSensitive$	symmetrically encrypted sensitive area produced in (41)
$Name$	the Name of the object being duplicated

To complete the duplication process, the TPM2B_PUBLIC and TPM2B_ENCRYPTED_SECRET produced by TPM2_Duplicate() are used in TPM2_Import() at the TPM containing the public and private portions of the NP. If the private area is doubly encrypted, then the symmetric key used for the inner wrapper is also given to the TPM.

TPM2_Import() will recover the symmetric key according to the algorithm of the NP. The TPM2B_PRIVATE is decrypted. If an inner wrapper is present, the TPM2B_PRIVATE is decrypted using the supplied symmetric key. After symmetric decryption, the integrity value is checked.

Marshal the $sensitive$ area into a TPM2B_SENSITIVE NOTE If no inner or outer wrapper is applied to the object, this structure is returned as the *duplicate* parameter in the response for TPM2_Duplicate().	size sensitiveArea: sensitiveType / authValue / seedValue / [sensitiveType]sensitive
Compute an $innerIntegrity$ value $innerIntegrity \coloneqq \mathbf{H}_{nameAlg}\,(sensitive\ ||\ name)$	size / innerIntegrity digest size sensitiveArea: sensitiveType / authValue / seedValue / [sensitiveType]sensitive

 Copyright © TCG 2006-2020

Set the encryption key (*symKey*) to *encryptionKeyIn* or a random value produced by the TPM.	
Create *encSensitive* by encrypting the *innerIntegrity* value and the TPM2B_SENSITIVE $encSensitive \coloneqq \mathbf{CFB}_{symAlg}\,(symKey, 0, innerIntegrity \,\|\|\, sensitive)$	
Using methods of the asymmetric new parent, create a *seed* value	
Create a symmetric key (*symKey*): $symKey \coloneqq \mathbf{KDFa}\,(npNameAlg, seed,$ "STORAGE", $Name,$ NULL, $bits)$	
Create *dupSensitive* by encrypting *encSensitive* $dupSensitive \coloneqq \mathbf{CFB}_{npSymAlg}\,(symKey, 0, encSensitive)$	
Compute the HMAC key from the *seed* created in step 5) $HMACkey \coloneqq \mathbf{KDFa}\,(npNameAlg, seed,$ "INTEGRITY", NULL, NULL, $bits)$	
Compute the HMAC over *dupSensitive* and include the object *Name* $outerHMAC \coloneqq \mathbf{HMAC}_{npNameAlg}\,(HMACkey, dupSensitive \,\|\|\, Name)$ NOTE An overall size field will be added to make the resulting TPM2B_PRIVATE structure.	size outerHMAC

Figure 19 — Duplication Process with Inner and Outer Wrapper

 Copyright © TCG 2006-2020

Figure 20 illustrates the processing of a duplication blob when no inner wrapper is used in the sensitive area.

Step	Structure
Marshal the *sensitive* area into a TPM2B_SENSITIVE	size; sensitiveArea: sensitiveType, authValue, seedValue, [sensitiveType]sensitive
Since there is no inner wrapper set $encSensitive \coloneqq sensitive$	
Using methods of the asymmetric new parent, create a *seed* value	
Create a symmetric key for encryption: $symKey \coloneqq \mathbf{KDFa}\ (npNameAlg, seed,$ "STORAGE", $name,$ NULL , $bits)$	
Create *dupSensitive* by encrypting *encSensitive* $dupSensitive \coloneqq \mathbf{CFB}_{npSymAlg}\ (symKey, 0, sensitive)$	
Compute the HMAC key from the *seed* created in step 3 $HMACkey \coloneqq \mathbf{KDFa}\ (npNameAlg, seed,$ "INTEGRITY", NULL, NULL, $bits)$	
Compute the HMAC over the *dupSensitive* $outerHMAC \coloneqq \mathbf{HMAC}_{npNameAlg}\ (HMACkey, dupSensitive \mid\mid name)$ NOTE An overall size field will be added to make the resulting TPM2B_PRIVATE structure.	size; outerHMAC

Figure 20 — Duplication Process with Outer Wrapper and No Inner Wrapper

 Copyright © TCG 2006-2020

Marshal the *sensitive* area into a TPM2B_SENSITIVE	size sensitiveArea: sensitiveType authValue seedValue [sensitiveType]sensitive
Compute an *innerIntegrity* value $innerIntegrity \coloneqq \mathbf{H}_{nameAlg}(sensitive \mid\mid Name)$	size: innerIntegrity digest size sensitiveArea: sensitiveType authValue seedValue [sensitiveType]sensitive
Set the encryption key (*symKey*) to *encryptionKeyIn* or a random value produced by the TPM.	
Create *encSensitive* by encrypting the *innerIntegrity* value and the TPM2B_SENSITIVE $encSensitive \coloneqq \mathbf{CFB}_{symAlg}(symKey, 0, innerIntegrity \mid\mid sensitive)$ NOTE An overall size field will be added to make the resulting TPM2B_PRIVATE structure.	

Figure 21 — Duplication Process with Inner Wrapper and TPM_RH_NULL as NP

Marshal the *sensitive* area into a TPM2B_SENSITIVE NOTE An overall size field will be added to make the resulting TPM2B_PRIVATE structure. This will result in a TPM2B_SENSITIVE being the only contents of the TPM2B_PRIVATE buffer.	size sensitiveArea: sensitiveType authValue seedValue [sensitiveType]sensitive

Figure 22 — Duplication Process with no Inner Wrapper and TPM_RH_NULL as NP

23.3.3 Rewrap

23.3.3.1 Introduction

TPM2_Rewrap() is a primitive of an exemplar key recovery service that performs all its security-sensitive processes on TPMs.

The effect of the recovery service is indistinguishable from duplication of a source key directly from a source platform to a destination platform.

The advantage of the recovery service is that

- registration of a source key with the recovery service relies upon an operational source platform, but not upon an operational destination platform, and
- delivery of the source key by the recovery service relies upon an operational destination platform, but not upon an operational source platform.

The recovery service keeps a source key from a source platform, irrespective of whether the destination platform is known. The source key is protected from the recovery service by virtue of a backup password that must be kept hidden from the recovery service but revealed to a destination platform. When the destination platform is revealed to the recovery service, the recovery service facilitates the installation of the source key in the destination platform.

a) While the source platform is operational, the source platform uses TPM2_Duplicate() to create a doubly wrapped duplication BLOB using a source key *TpmPrivateKey*, a backup password, and the recovery service's public key. (Duplication BLOBs are described earlier in this subclause. Note that the "Outer Duplication Wrapper" subclause explains that the outer wrapping is symmetric encryption that depends on a seed generated from a public key.)
b) While the source platform is operational, the source platform sends the duplication BLOB (Source BLOB in Figure 23) to the recovery service, which stores the BLOB.
c) When a destination platform is revealed to the recovery service, the recovery service uses TPM2_Rewrap() to derive another doubly wrapped duplication BLOB using the original doubly wrapped duplication BLOB, the recovery service's key, and the destination platform's public key.
d) When the destination platform is operational, the recovery service sends the derived duplication BLOB (Recovery BLOB in Figure 23) to the destination platform.
e) While the destination platform is operational, the destination platform uses TPM2_Import() to create a normal key BLOB from the derived duplication BLOB, the destination platform's key, and the backup password.

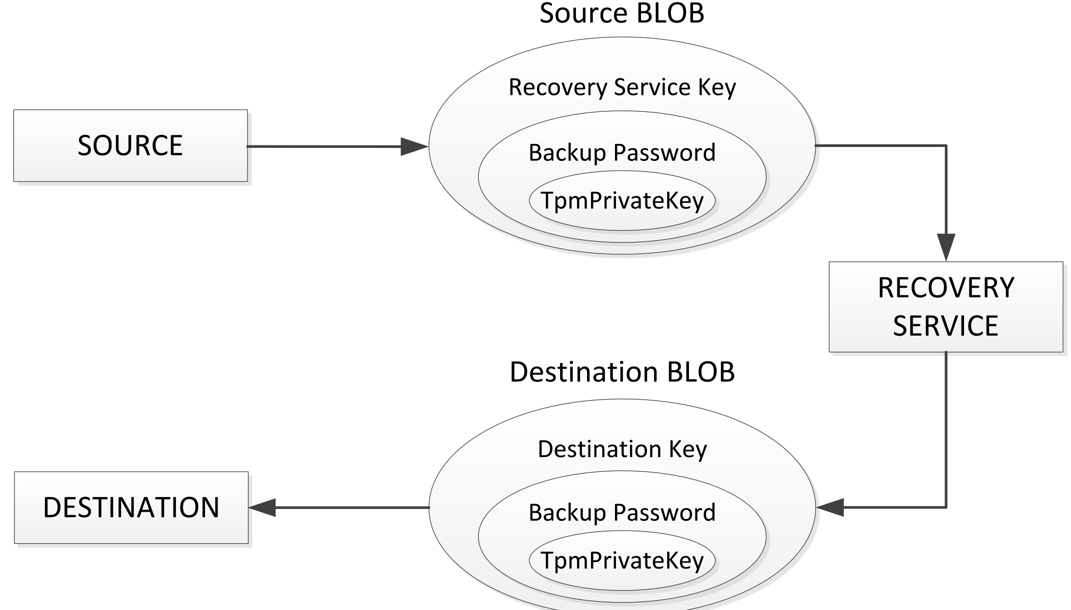


Figure 23 — Key Recovery Process

23.3.3.2 Creating a backed-up key

a) At the source platform, a key to be backed up, *sourceKey* = [*sourcePubKey* , *sourceSensitiveKey*], and the recovery service's public key *recoveryServicePubKey* are loaded in the source TPM.

b) At the source TPM, TPM2_Duplicate() is used to create the doubly wrapped duplication BLOB, which is *sourceSensitiveKey,* wrapped by *encryptionKeyIn*, wrapped by *recoveryServicePubKey*. The parameters to TPM2_Duplicate() are:

   1) *objectHandle* - references the key *sourceKey* to be sent to the recovery service

   2) *newParentHandle* - references the recovery service's public key *recoveryServicePubKey*

   3) *encryptionKeyIn* - is the backup password (this is an optional parameter and if the caller does not provide a value, the TPM will generate one)

   4) *symmetricAlg* – the encryption algorithm for the inner wrapper

c) The TPM returns:

   1) *encryptionKeyOut* – returned only if the TPM generated the key used for the inner wrapper

   2) *duplicate* – the wrapped sensitive area of *objectHandle;* the Source BLOB.

   3) *outSymSeed* - a protected version of the seed used to make the symmetric key used for outer wrapping encryption

d) The duplication BLOB duplicate is sent to the recovery service

23.3.3.3 Recovering a backed-up key

a) At the recovery service's platform, the recovery service's key *recoveryServiceKey =[recoveryServicePubKey , recoveryServiceSensitiveKey]*, and the destination platform's public key *destinationPubKey* are loaded into the recovery service's TPM.

b) At the recovery service's TPM, TPM2_Rewrap() is used to is used to replace the outer wrapper of the Source BLOB with an outer wrapper tied to the Destination Key *destinationPubKey*. The parameters for TPM2_Rewrap() are:

   1) *oldParent* - references the recovery service's key *recoveryServiceKey*

   2) *newParent* – references the destination platform's public key *destinationPubKey*

   3) *induplicate* – the Source BLOB

   4) *inSymSeed* - this is *outSymSeed* from the source platform. It is needed to derive the symmetric key used by the source platform for outer wrapping encryption

c) At the recovery service, the TPM will return

   1) *outDuplicate* – the rewrapped Destination BLOB

   2) *outSymSeed* - a protected version of the seed used to make the symmetric key used by the recovery service for outer wrapping encryption

d) At the destination platform, the destination platform's key *destinationKey =[destinationPubKey , destinationSensitiveKey]* is loaded into the TPM.

e) At the destination platform, TPM2_Import() is used to create *outPrivate* , which is a normal key BLOB that may be loaded into the TPM on the platform. The parameters to TPM2_Import() are:

   1) *parentHandle* – a reference to the destination platform's key (this will become the Storage Parent for the imported object)

   2) *encryptionKey* – the backup password (*encryptionKeyIn* or *encryptionKeyOut*)

   3) *objectPublic* – the public area of the key being imported

   4) *duplicate* – the Destination BLOB *outDuplicate* from the recovery service

   5) *inSymSeed* is *outSymSeed* from the recovery service. It is needed to derive the symmetric key used by the recovery service for outer wrapping encryption

 Copyright © TCG 2006-2020

f) At the destination platform, the TPM returns

1) *outPrivate* – the sensitive area of the imported object

23.4 Duplication Group

The duplication process allows an object or segment of a hierarchy to be duplicated for use in another hierarchy. This ability facilitates key distribution and backup. A duplication group is a group of objects in a hierarchy under a duplication root. The entire duplication group may be moved to another hierarchy by duplicating the duplication root.

When an object is created, its duplication attribute (*fixedParent*) is selected. If *fixedParent* is CLEAR, then the object may be operated on by TPM2_Duplicate(). This command allows the sensitive area of an object to be encrypted under a new parent so that it may be used in a different TPM hierarchy. The act of duplicating a Storage Key has the side effect of duplicating all of its descendants regardless of the setting of their *fixedParent* attribute. That is, if a Storage Parent is usable in a different hierarchy, then all the descendants of the Storage Parent are also usable in the different hierarchy as well.

NOTE 1 No modification of the encryption of a child object is required to make it usable on another hierarchy. This is because the Storage Key that is duplicated contains the information used to protect its children. Duplication of the protection information has the effect of duplicating the objects protected by that information.

NOTE 2 If a particular Storage Parent is usable in multiple hierarchies, then descendants of that Storage Parent are usable in the same hierarchies regardless of when they are created. That is, if they are created after the duplication of the Storage Parent, they are still usable in multiple hierarchies.

If an object has *fixedParent* CLEAR, it is the root of a duplication group. If the object is not a Storage Key, then the group will have a single member. For a Storage Key, the duplication group consists of all objects that are duplicated as a direct consequence of duplicating the group root.

Objects that have *fixedParent* SET cannot be directly duplicated (that is, they may not be the referenced *objectHandle* in TPM2_Duplicate()). However, they can be implicitly duplicated if an ancestor has *fixedParent* CLEAR and that ancestor is duplicated.

Objects that have *fixedParent* SET and have no ancestors with *fixedParent CLEAR* are the only objects that are not part of a duplication group. These objects are identified by having their *fixedTPM* attribute SET. All objects that are in a duplication group have their *fixedTPM* attribute CLEAR.

An object may be a member of more than one duplication group. This would occur if more than one of its ancestor Storage Keys has *fixedParent CLEAR* or if an object and one of its ancestors has *fixedParent* CLEAR.

 Copyright © TCG 2006-2020

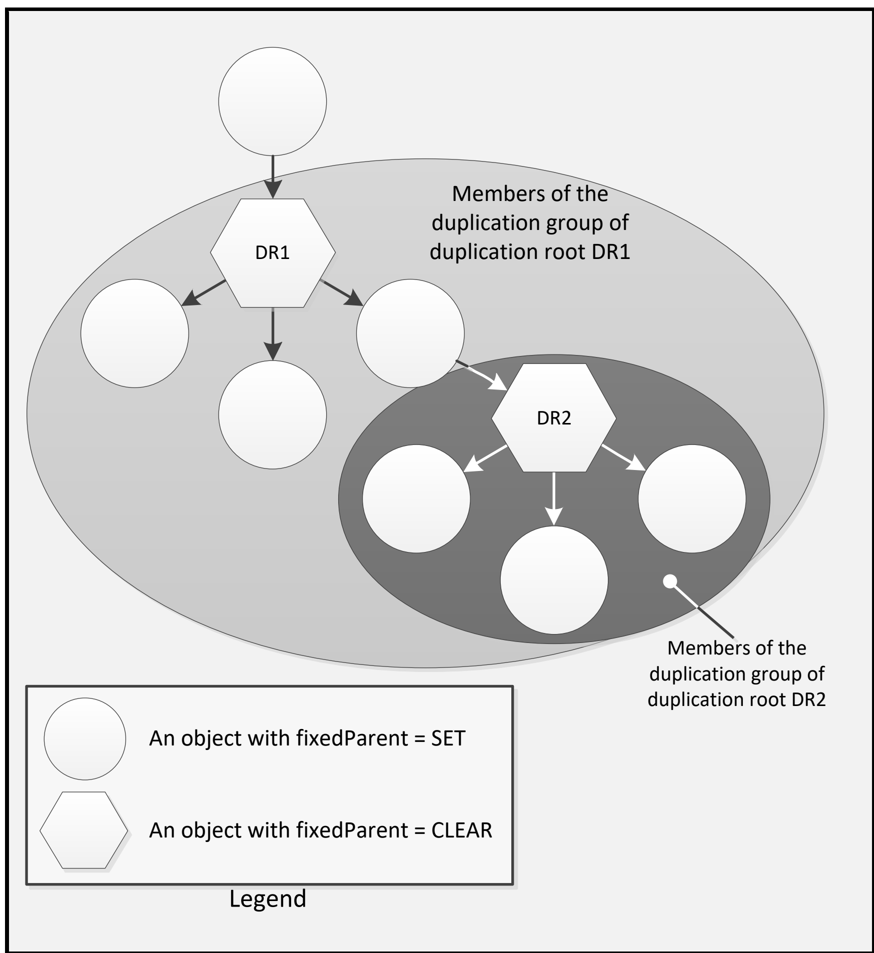


Figure 24 — Duplication Groups

23.5 Protection Group

The algorithms (asymmetric, symmetric, and hash) and key sizes used to protect child keys are consistent within a protection group. The protection group is all of the descendants of a duplication root not including other duplication roots or their descendants.

By requiring all of the non-duplicable Storage Keys to use the same algorithm, it is easier to determine the security properties of a hierarchy. If an object's *fixedTPM* attribute is SET, then all of the ancestor keys of that object use the same set of algorithms. If an object's *fixedTPM* is not SET, then the protections are determined by the duplication authority for each of the duplication roots in the object's hierarchy.

The reason that the protections are determined by the duplication authority and not by the algorithms of the key is that a duplication authority can attach a duplication root to a software-generated new parent. Inspecting the hierarchy in which an object exists does not guarantee the protections of the object unless the object's *fixedTPM* is SET.

Change of the algorithm set at a duplication root is illustrated in Figure 25.

 Copyright © TCG 2006-2020

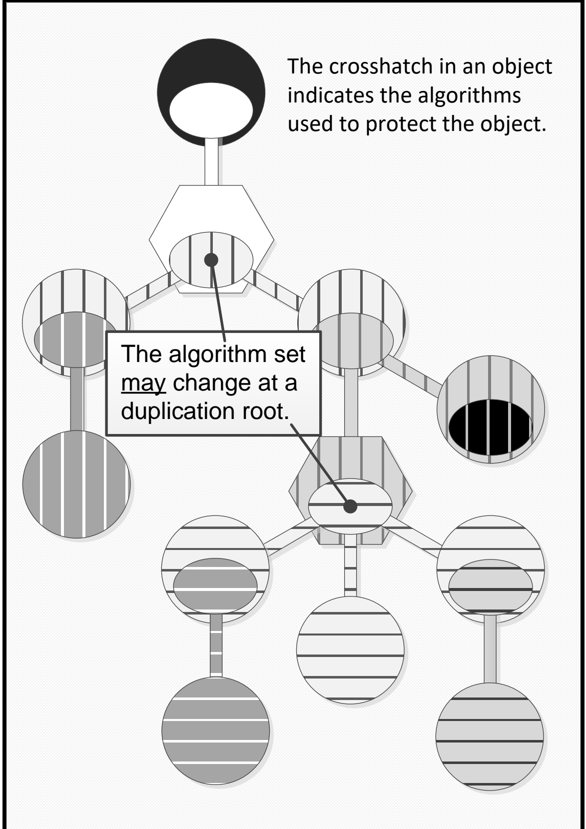


Figure 25 — Protection Groups

23.6 Summary of Hierarchy Attributes

The hierarchy attributes of an object indicate how the object is connected to the hierarchy. They indicate if the object could be extant in other hierarchies and if the object may be duplicated directly by TPM2_Duplicate().

Table 22 lists the possible combinations of an object's hierarchy attributes and the interpretation of each combination.

Table 22 — Mapping of Hierarchy Attributes

fixedParent	***fixedTPM***	**Description**
0	0	This combination represents a duplication root.
0	1	This combination is not allowed.
1	0	This combination indicates an object that is permanently in the protection group of its Storage Parent. It cannot be the *objectHandle* reference in TPM2_Duplicate().
1	1	This combination indicates an object that was created on a specific TPM and no duplicate of the object is possible.

 Copyright © TCG 2006-2020

23.7 Primary Seed Hierarchies

A Primary Object is an object that is derived from a Primary Seed value. The sensitive area of a Primary Object is not returned in the TPM2_CreatePrimary() or TPM2_CreateLoaded() response. The Primary Object will need to be regenerated each time it is needed, or it can be made persistent in NV memory on the TPM (TPM2_EvictControl()).

NOTE 1 A Primary Object may be duplicated in which case its sensitive area will be stored off of the TPM.

NOTE 2 The reason for not allowing a Primary Object to be returned is to prevent certain types of power analysis attacks on the primary seed values.

Once created, a Primary Object may be context-saved/restored.

A Primary Object may have *fixedParent* SET or CLEAR. If a Primary Object has *fixedParent* SET, then *fixedTPM* is required to be SET.

Hierarchy Attributes Settings Matrix

Table 23 shows the combinations of hierarchy settings allowed for an object. In the table, the check marks ("✓") indicate that the combination is allowed.

Table 23 — Allowed Hierarchy Settings

***fixedTPM* setting in**		**Object's *fixedParent***		
parent	**object**	**CLEAR**	**SET**	**Comments**
CLEAR	CLEAR	✓	✓	if the parent's fixedTPM attribute is CLEAR, the child's fixedTPM is required to be CLEAR
CLEAR	SET			
SET	SET		✓	if the parent of an object has fixedTPM SET, then fixedParent and fixedTPM must have the same setting in the child(1)(2)
SET	CLEAR	✓		

NOTE

1) For purposes of this table, the parent of a Primary Object is considered to have a *fixedTPM* attribute that is always SET.

2) If the parent has fixedTPM SET, then a child may be duplicable (*fixedParent* == CLEAR) or not (*fixedParent* == SET). If the child is not duplicable, then it is required to have the same setting of *fixedTPM* as its parent.

The consistency of the hierarchy settings is checked in object templates (TPM2_Create() and TPM2_CreatePrimary()) and in public areas for loaded objects (TPM2_Load()) or duplicated objects (TPM2_Import()).

 Copyright © TCG 2006-2020

24 Credential Protection

24.1 Introduction

The TPM supports a privacy preserving protocol for distributing credentials for keys on a TPM. The process allows a credential provider to assign a credential to a TPM object, such that the credential provider cannot prove that the object is resident on a particular TPM, but the credential is not available unless the object is resident on a device that the credential provider believes is an authentic TPM.

24.2 Protocol

The initiator of the credential process will provide, to a credential provider, the public area of a TPM object for which a credential is desired along with the credentials for a TPM key (usually an EK). The credential provider will inspect the credentials of the “EK” and the properties indicated in the public area to determine if the object should receive a credential. If so, the credential provider will issue a credential for the public area.

The credential provider may require that the credential only be useable if the public area is a valid object on the same TPM as the “EK.” To ensure this, the credential provider encrypts a challenge and then "wraps" the challenge encryption key with the public key of the “EK.”

NOTE “EK” is used to indicate that an EK is typically used for this process but any storage key may be used. It is up to the credential provider to decide what is acceptable for an “EK.”

The encrypted challenge and the wrapped encryption key are then delivered to the initiator. The initiator can decrypt the challenge by loading the “EK” and the object onto the TPM and asking the TPM to return the challenge. The TPM will decrypt the challenge using the private “EK” and validate that the credentialed object (public and private) is loaded on the TPM. If so, the TPM has validated that the properties of the object match the properties required by the credential provider and the TPM will return the challenge.

This process preserves privacy by allowing TPM objects to have credentials from the credential provider that are not tied to a specific TPM. If the object is a signing key, that key may be used to sign attestations, and the credential can assert that the signing key is on a valid TPM without disclosing the exact TPM.

A second property of this protocol is that it prevents the credential provider from proving anything about the object for which it provided the credential. The credential provider could have produced the credential with no information from the TPM as the TPM did not need to provide a proof-of-possession of any private key in order for the credential provider to create the credential. The credential provider can know that the credential for the object could not be in use unless the object was on the same TPM as the “EK”, but the credential provider cannot prove it.

24.3 Protection of Credential

The credential blob (which typically contains the information used to decrypt the challenge) from the credential provider contains a value that is returned by the TPM if the TPM2_ActivateCredential() is successful. The value may be anything that the credential provider wants to place in the credential blob but is expected to be simply a large random number.

The credential provider protects the credential value (CV) with an integrity HMAC and encryption in much the same way as a credential blob. The difference is, when *seed* is generated, the label is “IDENTITY” instead of “DUPLICATE”.

 Copyright © TCG 2006-2020

24.4 Symmetric Encrypt

A seed is derived from values that are protected by the asymmetric algorithm of the “EK”. The methods of generating the seed are determined by the asymmetric algorithm of the “EK” and are described in an annex to this TPM 2.0 Part 1. In the process of creating *seed*, the label is required to be “INTEGRITY.”

NOTE If a duplication blob is given to the TPM, its HMAC key will be wrong and the HMAC check will fail.

Given a value for *seed*, a key is created by:

$$symKey \coloneqq \mathbf{KDFa}\ (ekNameAlg, seed, \text{“STORAGE”}, name, \text{NULL}, bits) \tag{44}$$

where

ekNameAlg	the *nameAlg* of the key serving as the “EK”
seed	the symmetric seed value produced using methods specific to the type of asymmetric algorithms of the “EK”
“STORAGE”	a value used to differentiate the uses of the KDF
name	the Name of the object associated with the credential
bits	the number of bits required for the symmetric key

The *symKey* is used to encrypt the CV. The IV is set to 0.

$$encIdentity \coloneqq \mathbf{CFB}_{ekSymAlg}\ (symKey, 0, CV) \tag{45}$$

where

$CFB_{ekSymAlg}$	symmetric encryption in CFB mode using the symmetric algorithm of the key serving as “EK”
symKey	symmetric key from (44)
CV	the credential value (a TPM2B_DIGEST)

24.5 HMAC

A final HMAC operation is applied to the *encIdentity* value. This is to ensure that the TPM can properly associate the credential with a loaded object and to prevent misuse of or tampering with the CV.

The HMAC key (*HMACkey*) for the integrity is computed by:

$$HMACkey \coloneqq \mathbf{KDFa}\ (ekNameAlg, seed, \text{“INTEGRITY”}, \text{NULL}, \text{NULL}, bits) \tag{46}$$

where

ekNameAlg	the *nameAlg* of the target “EK”
seed	the symmetric seed value used in (44); produced using methods specific to the type of asymmetric algorithms of the “EK”
“INTEGRITY”	a value used to differentiate the uses of the KDF

$bits$	the number of bits in the digest produced by *ekNameAlg*

NOTE Even though the same value for label is used for each integrity HMAC, *seed* is created in a manner that is unique to the application. Since *seed* is unique to the application, the HMAC is unique to the application.

$HMACkey$ is then used in the integrity computation.

$$identityHMAC \coloneqq \mathbf{HMAC}_{ekNameAlg}(HMACkey, encIdentity \,||\, Name) \tag{47}$$

where

$\mathbf{HMAC}_{ekNameAlg}$	the HMAC function using *nameAlg* of the “EK”
$HMACkey$	a value derived from the “EK” symmetric protection value according to equation (46).
$encIdentity$	symmetrically encrypted sensitive area produced in (45)
$Name$	the Name of the object being protected

The integrity structure is constructed by placing the *identityHMAC* (size and hash) in the buffer ahead of the *encIdentity*.

24.6 Summary of Protection Process

Marshal the CV into a TPM2B_DIGEST	size | Credential Value
Using methods of the asymmetric “EK”, create a *seed* value	
Create a symmetric key for encryption: $symKey \coloneqq \mathbf{KDFa}$ ($ekNameAlg, seed,$ “STORAGE”, $Name$, NULL , $bits$)	
Create $encIdentity$ by encrypting the CV $encIdentity \coloneqq \mathbf{CFB}_{ekSymAlg}(symKey, 0, CV)$	
Compute the HMAC key $HMACkey \coloneqq \mathbf{KDFa}$ ($ekNameAlg, seed$, “INTEGRITY”, NULL, NULL, $bits$)	
Compute the HMAC over the $encIdentity$ from step 4 $outerHMAC \coloneqq \mathbf{HMAC}_{ekNameAlg}(HMACkey, encIdentity \,||\, Name)$	size | outerHMAC

Figure 26 — Creating a Identity Structure

 Copyright © TCG 2006-2020

25 Object Attributes

25.1 Base Attributes

25.1.1 Introduction

Three attributes are used to determine how the TPM may use an object. These attributes are designated as *restricted*, *sign*, and *decrypt*. The Boolean combinations of these attributes are used to express the full range of behaviors for objects.

25.1.2 *Restricted* Attribute

When the *restricted* attribute of a key is SET, the key may only operate on other objects that follow strict, but simple, format rules. A restricted key is not usable in all commands that use a key of that type. The restrictions on each type of key are explained in the clauses describing the *sign* and *decrypt* attributes.

The *restricted* attribute has no meaning when applied to an object that has both *sign* and *decrypt* CLEAR and *restricted* is required to be CLEAR for those objects.

25.1.3 *Sign* Attribute

This attribute may apply either to symmetric or asymmetric keys. A signing key uses its sensitive area key to sign data. The signature is returned by the TPM.

An asymmetric signing key may perform signing according to the key family (such as, RSA or ECC) and the signing method selected. An external entity may use the public portion of an asymmetric key to validate that the information was signed by someone with knowledge of the private portion of the key.

For a symmetric cipher object, this attribute and the object's mode determines whether the key can encrypt or sign (SMAC).

A symmetric key that can sign is used for performing an HMAC or an SMAC computation. This signature can be checked by another entity that knows the HMAC or SMAC secret key in order to validate the source of the information.

A restricted signing key may only sign a digest that has been produced by the TPM. The digest may be over externally supplied data or an internally generated structure. An internally generated structure that is to be signed will have the characteristic TPM_GENERATED_VALUE as the first octets in the structure to be hashed and signed. When the TPM generates a digest over externally provided data, the TPM validates that the first octets of the data are not equal to the TPM_GENERATED_VALUE. When a digest is signed by a restricted signing key, there is no ambiguity about whether or not the signed data was generated by the TPM.

A restricted signing key is occasionally referred to in this specification as an Attesting or Attestation Key.

25.1.4 *Decrypt* Attribute

An asymmetric decryption key uses the private asymmetric key in its sensitive area to decrypt data blobs that have been encrypted using the public portion of the key. A symmetric decryption key uses the key in its sensitive area to decrypt data that has been encrypted by that key.

A key that has both *decrypt* and *restricted* attributes SET only accepts data that has a specific structure. The encrypted data block must have as its first element an integrity value for the remainder of the

 Copyright © TCG 2006-2020

structure. This integrity value is an HMAC of the encrypted data. This format allows the TPM to prevent misuse of the restricted decryption keys that are the basis of the protected storage hierarchy.

If the sensitive data is part of a child object, the symmetric and HMAC keys are derived from the symmetric seed value (*seedValue*) in the sensitive area of the Storage Parent. If the sensitive data is a duplication or certification blob, the symmetric and HMAC keys are derived from a single use seed. That seed is then protected using the asymmetric public key of the intended recipient of the protected blob.

When loading a protected blob, the TPM validates the integrity value before decrypting the data. The only way that the integrity value can be correct is if it were created by some entity with access to the unencrypted sensitive data.

NOTE The specific threat scenario that is addressed by this scheme is that an attacker will use a protected blob in a command that is not appropriate for that blob. For example, an attacker may load the sensitive portion of an asymmetric key and attempt to access the sensitive area using TPM2_Unseal(). The TPM will unseal data, but not a key. The attacker may attempt to modify the public area of the key in order to trick the TPM into thinking that the protected blob contains a sealed data rather than a private key. The integrity value prevents these deceptions.

A restricted decryption key is often referred to in this specification as a Storage Key.

 Copyright © TCG 2006-2020

25.1.5 Uses

Table 24 shows the combinations of an object's functional attributes and describes the resulting properties.

Table 24 — Mapping of Functional Attributes

sign	*decrypt*	*restricted*	Description
0	0	0	A data blob. Can be accessed using TPM2_Unseal(). NOTE: This attribute set may only be used for a *keyedHash* object.
0	0	1	Not allowed. The TPM will not load or create an object with this setting.
0	1	0	A key that can be used in any operation that requires a decryption key, except that the key may not be a storage key.
0	1	1	Indicates that only the default schemes and modes of the key may be used In this specification, key with these properties is referred to as a Parent Key. Asymmetric keys and symmetric keys with these attributes are Storage Parents, and *keyedHash* objects with these attributes are Derivation Parents. The TPM only allows keys with these attributes to be used on objects that have a specific structure. For Storage Parents, use includes create, load, and activate credential.
1	0	0	Indicates a key that may be used with any signing operation including quote, certify, and sign. The recipient of signatures generated by this key should be aware that quotes and certifications can be forged so the trust would not be in the key but in the entity that knows the key authorization value. If use with object type TPM_ALG_KEYEDHASH, then the key may be used for HMAC operations.
1	0	1	This combination indicates a key that can sign any digest that the TPM has created. The TPM only signs a digest over externally provided data that did not have as its first octets TPM_GENERATED_VALUE. This key can be used reliably for quoting, certifying, and signing. No signing command is prohibited for this type of key. Only the default schemes and modes of the object may be used.
1	1	0	A general-purpose key that can be used with any command that requires a key as long as the command is compatible with the key algorithm. However, this key may not be a Storage Key (the parent of other keys).
1	1	1	This type of key is currently not supported because use of a signing key as a storage node could prevent an application from being able to use the TPM in a way that is compliant with FIPS.

Table 25 shows the correspondence between the TPM 1.2 method of identifying key properties and the method in this specification.

Table 25 — TPM 1.2 Correspondence

TPM 1.2 Name	sign	decrypt	restricted	Comments
TPM_KEY_SIGNING	1	0	0	In TPM 1.2, keys had restricted schemes. In this specification, the scheme is defined in the command.
TPM_KEY_STORAGE	0	1	1	The functional properties are nearly the same as TPM 1.2. This key could only be used to protect and unprotect items in a Protection hierarchy.
TPM_KEY_IDENTITY	1	0	1	In TPM 1.2, an Identity key was highly constrained and could not, for example, sign a structure that was not produced by the TPM. In this specification, the restricted signing key can sign (within the limits defined in clause 25.1.3) a digest produced by the TPM. This allows, for example, an Attestation Key to sign a PKCS#10 certificate request.
TPM_KEY_AUTHCHANGE	-	-	-	This is not used in this specification and its use was deprecated in TPM 1.2. The functionality is provided by session encryption.
TPM_KEY_BIND	0	1	0	Functionality is roughly equivalent between the TPM 1.2 type and the unrestricted decryption key. The specification would use TPM2_RSA_Decrypt() in place of the TPM 1.2 TPM_Unbind().
TPM_KEY_LEGACY	1	1	0	Use of these keys is only constrained by the key family properties. For example, an ECC key will not perform TPM2_RSA_Decrypt().
TPM_KEY_MIGRATE	0	1	1	A Storage Key may be the object of a re-wrap if the new parent is allowed within the policy for the object. The policy for duplication of the object is always visible in the public area.
Sealed Data				A blob containing user defined data

25.2 Other Attributes

25.2.1 fixedTPM and fixedParent

These attributes are described in detail in clause 23.

25.2.2 stClear

This attribute indicates an object that will need to be reloaded after any Startup(CLEAR). Objects may be loaded into the TPM and their context saved by the TPM resource manager. Normally, these saved contexts may be reloaded at any time before the next TPM Reset. However, if this attribute is SET, then the saved context associated with the object will be invalidated on each TPM Restart as well as on TPM Reset.

An object that has this attribute SET may not be made persistent.

 Copyright © TCG 2006-2020

25.2.3 sensitiveDataOrigin

The meaning and allowed settings for this attribute are different for Created and Derived Objects. For a Derived Object, this attribute is required always to be CLEAR. For a Created Object, this attribute is SET if the sensitive data of the object is to be generated by the TPM.

NOTE 1 The reason that *sensitiveDataOrigin* is to be CLEAR for a derived object is that it is impractical to use it to indicate anything about the provenance of the seed value used in deriving an object. The only case in which the *sensitiveDataOrigin* of the Derivation Parent might reasonably be reflected in the derived key is when *sensitiveDataOrigin* and *fixedTPM* are both SET in the parent. For all other cases, there is no way for the TPM to provide any assurance about the setting of *sensitiveDataOrigin.* However, for a derived key with fixedTPM SET, it is a relatively simple matter to check the setting of this attribute in the Derivation Parent. Rather than add to the TPM the complexity of validating that a Derivation Parent has the correct combination of attributes to allow this attribute to be SET, it was decided to require that this attribute be CLEAR rather than ignored. This is because this attribute does not change the way that Object derivation takes place as it does with Object creation.

When a symmetric object (TPM_ALG_KEYEDHASH or TPM_ALG_SYMCIPHER) is created, the caller may provide the secret data or have the TPM generate it. If the TPM is to be the source of the data, then the caller will SET this attribute. Otherwise, this attribute will be CLEAR, and the caller-provided data will be used.

When an asymmetric object is created, this attribute must be SET. The public part of an asymmetric object is determined by its private key. If the caller has control over both the public and sensitive areas, then the TPM cannot ensure that the key is statistically unique. This is not an issue unless the object also has *fixedTPM* SET. One of the assumptions of a *fixedTPM* object is that it is statistically unique. This would not be the case for an asymmetric key if the caller provided the data. To avoid the possibility of creating a *fixedTPM* object on multiple TPMs, an asymmetric key is required to have its private key generated by the TPM or the object may be imported. If it is imported, *fixedTPM* will not be SET.

“NOTE 2 The requirement that *sensitiveDataOrigin* be SET for asymmetric objects is enforced indirectly. When an asymmetric key is created, the caller is not allowed to provide the sensitive data of the key. Because the caller does not provide the sensitive data, *sensitiveDataOrigin* is required to be SET. Since this relationship is only checked when the object is created, *sensitiveDataOrigin* is allowed to have any setting when an object is loaded or imported.

25.2.4 userWithAuth

This attribute indicates that the object's *authValue* may be used to provide the USER role authorizations for the object. If this attribute is CLEAR, then USER role authorizations may only be provided by satisfying the object's *authPolicy* in a policy session. A policy session may be used for USER mode authorizations when this attribute is SET or CLEAR.

25.2.5 adminWithPolicy

This attribute indicates that authorization for an action requiring the ADMIN role requires that the *authPolicy* of the object be satisfied. If this attribute is CLEAR, then the *authValue* may be used in an HMAC session to perform operations that require ADMIN role.

As with USER role authorizations, any ADMIN role action may be authorized with a policy session that satisfies the *authPolicy*.

The primary reason for having a set of operations that require ADMIN role is to allow each of the actions to be individually controlled. When a policy is used for an ADMIN role action, the policy must contain a command that sets the *commandCode* for the policy to the specific command. This allows each ADMIN role action to be individually enabled and controlled without having to group them.

 Copyright © TCG 2006-2020

25.2.6 noDA

If this attribute is SET in an object, then authorization failures of the object will not invoke dictionary attack protections. In addition, actions on an object with this attribute SET are not subject to lockout. This attribute is used to ensure that access to objects used by the OS is not blocked due to actions by users. An OS would be expected either to use objects with well-known values or to use high-entropy authorization values. In neither case is dictionary attack protection required.

25.2.7 encryptedDuplication

If this attribute is CLEAR, then an object may be duplicated with *newParentHandle* set to TPM_RH_NULL, which means that there is no outer wrapper for the object. If the caller does not specify an inner wrapper, then the object may be exported with this sensitive area in the clear.

While the entity that controls duplication is expected to be trusted to maintain the confidentiality of the sensitive area of a key during duplication, conformance to some standards requires that the sensitive area be encrypted when it leaves the TPM and reliance on the caller is not adequate for those standards. This attribute provides a method of producing objects that conform to those standards.

NOTE It is understood that the duplication authority can still arrange to have access to the sensitive area of the key by creating a software key and having the TPM duplicate to that key.

 Copyright © TCG 2006-2020

26 Object Structure Elements

26.1 Introduction

The TPM is intended to provide a means of creating a Storage hierarchy to protect data and keys (keys generated by the TPM or some other entity). Each of these objects (keys and data) has two components. The first is a public area that contains the attributes of the object and a public identity. The second is the sensitive area that contains the elements of the object that require TPM protections. These elements include an authorization value, one or more secret key values, and, in some cases, sealed data values.

The structure definitions for both the public and sensitive areas of an object define how the information is to be arranged when it crosses the TPM interface. The organization of these structures as they exist within the TPM is at the discretion of the TPM vendor. However, the actions of commands in this specification are defined in terms of these presumptive structures and any implementation will need to produce equivalent results.

26.2 Public Area

The public area contains the information for identification of an object and its properties. The fields of the public are listed and described in Table 26.

Table 26 — Public Area Parameters

Parameter	Description
type	This identifies the type of the object. An algorithm ID is used as the type identifier because the structures contain parameters that are specific to the types of operations that can be performed on or with the object. For example, an RSA type would contain an RSA key pair that could be used for operations defined for RSA. An AES type would be used for symmetric encryption or decryption.
nameAlg	This is a second algorithm ID that identifies the hash algorithm used for computing the Name of the object.
objectAttributes	This contains the set of attributes of the object. These attributes are in five classes: 1) usage (*sign*, *encrypt*, *restricted*); 2) authorization (*userWithAuth, adminWithPolicy, noDA*); 3) duplication (*fixedParent, fixedTPM, encryptedDuplication*); 4) creation (*sensitiveDataOrigin*); and 5) persistence (*stClear*).
authPolicy	This will contain the authorization policy for the object if one is defined. *nameAlg* is used as the *authPolicy* hash algorithm, NOTE An object that is intended to be duplicated must have an *authPolicy* enabling the duplication.
[type]parameters	The parameters of an object are dependent on the object *type*. For symmetric key object, the parameters would indicate the size of the key and the default encryption mode. For an asymmetric object (RSA or ECC), the parameters would indicate the key size, signing scheme, and symmetric encryption methods associated with the key.
[type]unique	The unique value of an object is also dependent on the object *type*. For an asymmetric object, this will be the public key. For a symmetric object, this will be a value computed by hashing values in the sensitive area.

26.3 Sensitive Area

The sensitive area is related to the public area and contains the data that are required to be encrypted when not in a Shielded Location on the TPM. It contains the authorization value and the item-specific

Copyright © TCG 2006-2020

information such as the private or secret portion of a key. If an object is a Storage Key, it contains the symmetric key that is used to encrypt its child object.

The structure of the sensitive area is shown in Table 27.

Table 27 — Sensitive Area Parameters

Parameter	Description
sensitiveType	This identifies the type of the object for this sensitive area. This value and the type parameter of the public area are the same.
authValue	This is the authorization value for the object. It is an octet array of zero or more octets. The authorization value for an object may not have more octets than the digest produced by the object's *nameAlg*.
seedValue	This value is required for Storage Keys and is the seed used to generate the protection values for the child objects of the Key. This is optional for asymmetric keys that are not Storage Keys and is not used if present. For all other object types, this is an obfuscation value. It is hashed with the *sensitive* field to produce the *unique* value in the public area. Including this value in the computation obfuscates *unique* so that the *sensitive* value cannot be determined from the *unique* field.
[sensitiveType]sensitive	The contents of this parameter are dependent on *sensitiveType.* For an asymmetric key, this will contain the private key. For a symmetric key, this will be the key. For an HMAC key this is the HMAC key value. For a data object, this will be the sensitive data.

Each sensitive area created by the TPM contains some TPM-created data that makes each sensitive area statistically unique. This will be either an asymmetric key or a large random number. The unique values in the sensitive area are cryptographically linked to values in the public area in a way that makes each public area statistically unique. The fact that a sensitive area is statistically unique and cryptographically linked to a public area ensures that a TPM can detect any attempt to substitute the sensitive area associated with a public area.

NOTE Such a substitution would allow subversion of secrets-based policy authorization. If an attacker could use an arbitrary sensitive area with a public area with a known Name, the attacker could perform TPM2_PolicySecret() and cause the *policyDigest* to be updated with the chosen Name even though the attacker does not know the authorization value of the correct sensitive area. Cryptographic linking of the sensitive area to the public area ensures that this type of attack is not practical.

26.4 Private Area

When a sensitive area is not in a Shielded Location on a TPM, it is integrity-protected and symmetrically encrypted. There is more than one format for a protected sensitive area but the loadable (TPM2_Load()) form of the protected sensitive area is called a "private" area.

NOTE 1 Another format is a saved context.

The process of converting a sensitive area to a private area requires that the sensitive area be marshaled to its canonical form. This marshaled structure is then encrypted using a key derived from the Storage Parent's symmetric seed (*seedValue*). An HMAC is performed over the data with the Name of the associated sensitive area include in the HMAC. The combination of the HMAC and the encrypted sensitive area is a key's private area.

NOTE 2 Similar protections are used when an object is context saved or duplicated.

 Copyright © TCG 2006-2020

26.5 Qualified Name

The Qualified Name (QN) of an object is the digest of all of the Names of all of the ancestor keys back to the handle of the Primary Seed at the root of the hierarchy. The QN of an object includes the Name of the object. The QN uses the Name hash of the current object to compute the QN for the object.

EXAMPLE 1 Assuming that key A is the Storage Parent of object B, then the Qualified Name of B (QN_B) is:

$$QN_B \coloneqq \mathbf{H}_B\ (QN_A \mid\mid NAME_B)$$

The QN is not a digest of all of the entities loaded into the TPM. It is a digest of all of the entities in a chain.

EXAMPLE 2 Assume two entities with public areas of A and B and different Name hash algorithms (H_A and H_B). Also assume that they share the same parent P with a QN of QN_P. The QN for A is $QN_A \coloneqq H_A(QN_P \mid\mid H_A(A))$ and the QN for B is $QN_B \coloneqq H_B(QN_P \mid\mid H_B(B))$.

The primary purpose of the Qualified Name is to supplement the environmental information relating to object creation and object use. The environment of an object includes its hierarchy. The hierarchy starts at a Primary Seed and includes all ancestor keys for the object. The Qualified Name of an object is included in its creation data. The Qualified Name permits validation that a list of ancestor Names is correct. It is then possible to determine if, for example, all ancestor keys use sufficient cryptographic strength. The Qualified Name of an object is included in its certification to indicate that the key is being used in a different environment (ancestry) than the one in which it was created.

Both the Name and Qualified Name for a Primary Seed are the handle of the Primary Seed. If the parent handle is TPM_RH_NULL, Name and QN are also TPM_RH_NULL. This makes the QN of a Primary Object or Temporary Object equal to:

$$QN \coloneqq \mathbf{H}_{nameAlg}\ (A\ hierarchy\ handle \mid\mid Primary\ Object\ Name) \tag{48}$$

NOTE The creation data for an object includes both the Name and QN of the Storage Parent of that object.

26.6 Sensitive Area Encryption

When a sensitive area is in a loadable format (a private area), the symmetric encryption key is derived from the secret seed (*seedValue*) of the parent.

When a sensitive area has been encrypted for duplication, the sensitive area is symmetrically encrypted with a key that is protected using asymmetric methods associated with the new parent. Before a duplicated object may be loaded, it must be “imported” (TPM2_Import()) and encrypted using the symmetric key derived from the secret seed of the new parent.

NOTE Clause 30.3 describes the protections that are applied to a sensitive area when it is part of a saved context.

All symmetric encryption of the sensitive area uses Cipher Feedback (CFB) mode.

The method of generating the encryption key and IV for the encryption is described in clause 22.

26.7 Sensitive Area Integrity

When an object is not in a Shielded Location, it is susceptible to modification through means other than through a Protected Capability. An HMAC-based integrity scheme allows these modifications to be detected. The integrity HMAC includes the sensitive data and some representation of the public area. Inclusion of the public area preserves the binding between the two elements of the object.

 Copyright © TCG 2006-2020

The HMAC key is generated from the same seed that is used for generating the symmetric encryption key and IV. The HMAC of the protected structure is required to be checked before the sensitive area is decrypted.

 Copyright © TCG 2006-2020

27 Object Creation

27.1 Introduction

TPM2_Create(), TPM2_CreatePrimary() and TPM2_CreateLoaded() are used to create the objects (keys and data) that are part of a TPM's Storage hierarchy. TPM2_CreatePrimary() is used to create Primary Objects that are derived from a Primary Seed. TPM2_Create() is used to create Ordinary Objects that are generated with values from the TPM RNG. TPM2_CreateLoaded() can be used to create a Primary or Ordinary Object.

NOTE 1 TPM2_CreateLoaded() may also be used for Derived Objects. This is covered in more detail in clause 28.

Table 28 — Creation Commands

Creation command	**TPM2_CreatePrimary()**	**TPM2_Create()**	**TPM2_CreateLoaded()**		
Parent Handle Type	Primary Seed	Storage Parent	Primary Seed	Storage Parent	Derivation Parent
Created Object Type	Primary	Ordinary	Primary	Ordinary	Derived
Public Area Returned	yes	yes	yes	yes	yes
Sensitive Area Returned	no	yes	no	yes	no
***creationData* Returned**	yes	yes	no	no	no

Table 28 compares and contrasts the creation of objects by TPM2_CreatePrimary(), TPM2_Create(), and TPM2_CreateLoaded(). In particular, when creating keys:

- TPM2_CreatePrimary() – creates and loads Primary Objects for immediate use and provides creationData.
- TPM2_Create() – creates Ordinary Objects for later use (via TPM2_Load()). TPM2_Create() returns a BLOB containing the sensitive area of an Ordinary Object and provides creationData.
- TPM2_CreateLoaded()– depending on the type of the parent, generates and loads a Primary Object, an Ordinary Object; or Derived Object.

Authorization to use the Parent is required in order to generate a child. Authorization to use a Primary Seed is required in order to create a Primary Object.

All of the objects created by these commands are similar in most respects. For TPM2_Create() and TPM2_CreatePrimary(), the parameters required to create an object are the same for both commands. They are:

- a public area template,
- the sensitive values,
- optional user-provided identification data, and
- the optional creation PCR selection.

For TPM2_CreateLoaded(), the only parameters are the public area template and the sensitive values.

NOTE 2 The user-data and PCR parameters are not used for TPM2_CreateLoaded() as it does not return the *creationData* used for creation certification. For objects where the creation certification is necessary, the TPM2_Create() or TPM2_CreatePrimary() functions are available.

 Copyright © TCG 2006-2020

Any type of object that can be created with TPM2_Create() can be created with TPM2_CreatePrimary() or TPM2_CreateLoaded().

NOTE 3 TPM2_CreateLoaded() can be used for creation of asymmetric keys but it may not be used for derivation of certain types of asymmetric keys. This limitation is because of the variability in algorithms for some asymmetric key types (such as RSA).

The sensitive area of an Object created from a seed does not leave the TPM except in a saved context or by duplication. If a Primary Object is not context saved (and not persistent), it will need to be recreated after the next TPM2_Startup(). Even if context saved, if a Primary Object is not made persistent in the TPM (TPM2_EvictControl()), it will need to be recreated after each TPM Reset.

27.2 Public Area Template

27.2.1 Introduction

A public area template describes the desired attributes of the object to be created. The TPM uses this template to guide the creation of the new object.

The format of the template has to match the desired format of the object to be created, in all details. The item-specific information (the *unique* field) will be replaced by the TPM in the creation process, but all other fields in the created object will be identical to those in the template.

In general, the fields in the public area are checked as if the object were being loaded under the Storage Parent indicated in the creation command.

27.2.2 type

This parameter indicates the basic type of the object and determines the format of the *parameters* and *unique* fields. The type may indicate a symmetric key, an asymmetric key, or a data value.

The allowed values for *type* are: TPM_ALG_SYMCIPHER, TPM_ALG_KEYEDHASH, TPM_ALG_RSA, or TPM_ALG_ECC.

NOTE The list of types may change. If an algorithm ID is allowed for use as a public area type, it is denoted by an "O" in the "Type" column of the TPM_ALG_ID constants table published by the TCG.

27.2.3 nameAlg

The *nameAlg* parameter in the template is set according to the object type. If the object is a restricted-decryption key, then the object is required to have the same *nameAlg* as the Storage Parent. For all other cases, the *nameAlg* may be any supported hash algorithm.

In the case of TPM2_LoadExternal(), *nameAlg* is allowed to be TPM_ALG_NULL. When this value is selected, the TPM does not validate the cryptographic linkage between the public and sensitive portions of the object. Since the *nameAlg* is TPM_ALG_NULL, the object has no Name.

NOTE Certification of the key with no Name has no meaning as the certification will have no Name for the certified object.

27.2.4 objectAttributes

These flags must be set according to the rules appropriate for loading the object. The required settings are found in TPM 2.0 Part 2, in the definition of TPMA_OBJECT.

 Copyright © TCG 2006-2020

27.2.5 authPolicy

If use of an object is to be gated by a policy (including PCR), the template will contain the policy hash. Otherwise, this entry will be set to the Empty Policy.

27.2.6 parameters

This field contains parameters that describe the details of the object indicated in *type*.

For a Storage Key that has *fixedParent* SET in its *objectAttributes*, these parameters will be identical to the parameters of the Storage Parent. For other objects, these parameters may vary according to the *type* and application.

27.2.7 unique

The *unique* field of the template is the only field in the public area that is replaced by the TPM during the object creation process. The caller may place any value in this field as long as the structure of the value is consistent with the *type* field. That is, this field should be structured in the same way as the data that will be placed in this field by the TPM. The caller may also set the size of this field to zero and the TPM will replace it with a correctly sized structure.

27.3 Sensitive Values

27.3.1 Overview

The sensitive values that are provided when the object is created allow initial setting of the *authValue* for the object and may provide some other object-sensitive value. The sensitive value may be an encryption key or sealed data.

The sensitive values provided to the TPM in TPM2_Create() and TPM2_CreatePrimary() (the inSensitive parameter) may optionally be encrypted using standard session-based encryption techniques. Since session-based encryption allows use of a different session for authorization and encryption, the session used for encrypting the authorization and other sensitive data does not have to be the same as the authorization session for the Storage Parent of the newly created object. This ensures that the entity that controls the Storage Parent does not automatically gain access to the secret values of a child.

27.3.2 userAuth

The *userAuth* value is the initial *authValue* for the created object. This value may be no larger than the digest produced by the *nameAlg* of the object.

NOTE This limitation ensures that any valid *authValue* will be usable on any TPM that can load the key. If this limitation were not imposed, then some TPM might not be able to load a duplicated object because the *authValue* was too large for the implementation.

27.3.3 data

This contains information that the caller wants to be incorporated in the sensitive part of the created object. This may be either a symmetric key or user data. If *data* is an Empty Buffer, then the *sensitiveDataOrigin* attribute of the template is required to be SET. If data is not empty, then *sensitiveDataOrigin* is required to be CLEAR.

 Copyright © TCG 2006-2020

If the object type is TPM_ALG_KEYEDHASH and both *sign* and *encrypt* are CLEAR, then the created object is a Sealed Data Object and the TPM will return an error (TPM_RC_SIZE) if *data* is an Empty Buffer.

If the created object is an asymmetric key and not a primary key, then *data* is required to be an Empty Buffer and *sensitiveDataOrigin* in the template is required to be SET. For a primary key, *data* permits personalization of the key with private data, data that can be provided as an encrypted parameter.

NOTE If the caller were allowed to specify the private key, then for some types of asymmetric algorithms (such as, ECC) the actions of the TPM would not determine the Name of the object. Since the TPM has no effect on the creation of such an object, the preferred means of having such a key become part of a hierarchy is to import it with TPM2_Import().

27.4 Creation PCR

The PCR selection that is present in TPM2_Create() or TPM2_CreatePrimary() parameters is used to select the PCR values that will best represent the environment in which the object was created. The selection and the PCR are hashed according to the creation data algorithm and included in the creation data (a TPM2B_CREATION_DATA) that is returned in the command response.

NOTE When an Object is created, the TPM produces a ticket that it (the TPM) can use to verify that it created the Object. This allows the TPM to certify that it created the Object (TPM2_CertifyCreation()).

27.5 Public Area Creation

27.5.1 Introduction

This clause describes how the TPM uses the parameters of TPM2_Create() and TPM2_CreatePrimary() to set the values in the public area of the created object.

This clause does not describe the error conditions if the parameters are bad. That information is provided in the description of TPM2_Create() and TPM2_CreatePrimary() in TPM 2.0 Part 3.

27.5.2 type, nameAlg, objectAttributes, authPolicy, and parameters

The TPM will validate that these parameters are consistent in the template and then copy them from template into the created structure without modification.

27.5.3 unique

27.5.3.1 Introduction

This parameter will contain a *type*-specific structure. It is used to ensure that each object has a statistically unique identity. The methods used to create *unique* ensure that it is cryptographically bound to the contents of the sensitive area. Creation of *unique* from the sensitive data uses non-invertible processes (such as, a hash) so that the *unique* value does not compromise the confidentiality of the sensitive area.

The computation of *unique* uses one or more values in the sensitive area of the object. At least one of the sensitive area values will be provided by the TPM to ensure that *unique* is, in fact, unique. For asymmetric keys, uniqueness is provided by the public key and the public key is mathematically linked to the private key in the sensitive area.

 Copyright © TCG 2006-2020

For symmetric objects (symmetric keys, HMAC keys, and data blobs), the key (or data) is hashed with a TPM-generated obfuscation value and the resulting digest is used as the *unique* value.

There are two reasons for generating the *unique* parameter for symmetric objects in this way. The first is that it protects the contents of the user-provided data. If the secret data has low entropy, then making the *unique* parameter a simple digest of that data would allow an offline attack to determine what the secret data might be. The large, random, obfuscation value generated by the TPM is not known to an attacker, which mitigates this threat.

The second reason for this method is that it prevents an attacker from stealing an object's identity. If the identity were not based on the contents of the sensitive area, then an attacker could create a sensitive structure and associate it with the public area of any symmetric object. Having the sensitive area contain information that can cryptographically link the sensitive area to the public area prevents this kind of substitution.

The methods for producing *unique* for each of the object types are described in the remainder of 27.5.3.

27.5.3.2 TPM_ALG_KEYEDHASH

This type is used for an HMAC key or data block. The computation for *unique* for a KeyedHash object is:

$$unique \coloneqq \mathbf{H}_{nameAlg}\left(obfuscate \mid\mid key\right) \tag{49}$$

where

$\mathbf{H}_{nameAlg}$	hash using *nameAlg* from the object template
$obfuscate$	the contents of *seedValue.buffer* in the object's sensitive area
key	the contents of *sensitive.bits.buffer* in the object's sensitive area; this will be either an HMAC key, a data blob, or a symmetric key.

27.5.3.3 TPM_ALG_SYMCIPHER

This type is used for a symmetric block cipher key. The *unique* value is computed as shown in (49).

27.5.3.4 TPM_ALG_RSA

For an RSA key, *unique* is the public modulus of the key. It is computed as described in B.8.

27.5.3.5 TPM_ALG_ECC

For an ECC key, *unique* is the public point computed as described in C.5.

27.6 Creation Entropy

27.6.1 Introduction

The reference code uses common algorithms for generating keys of a specific type. That is, there is one algorithm for generating RSA keys, one for ECC keys, one for HMAC keys and one for symmetric keys.

NOTE RSA and ECC are "if implemented."

 Copyright © TCG 2006-2020

When calling these functions, the caller is allowed to indicate where the function should get entropy for use in the algorithm. This allows these functions to be used for Primary, Derived, and Ordinary Objects simply by changing the source of “entropy.”

Table 29 — Deriving Object Entropy

Object Type	Source of Object Entropy	Described in Clause
Primary	DRBG initialized using a hierarchy seed and the hash of the input template	27.6.3
Ordinary	TPM’s default DRBG	27.6.2
Derived	KDF	28.4

Table 29 compares and contrasts the methods used to create the cryptographic values of primary, ordinary and derived keys.

- A Primary Object is intended to be created multiple times, in the absence of any other key, with the minimum amount of persistent storage. As a result, the cryptographic values of primary keys are created by instantiating a DRBG.
- An Ordinary Object is intended to be created exactly once and persistently stored. As a result, the cryptographic values of ordinary objects are created by the DRBG that is used by default when a TPM requires random data. This DRBG is seeded with entropy when the TPM was created and topped-up with additional entropy added at intervals.
- A derived key is intended to be derived multiple times from a parent key, and not persistently stored. As a result, the cryptographic values of derived keys are created by applying a KDF and hash algorithm specified in the Derivation Parent to the Derivation Parent’s symmetric key, using label and context values provided by the caller.

27.6.2 Entropy for Ordinary Objects

For an Ordinary Object, the caller would pass a NULL pointer. When the key generation function needs a random number, it would pass that NULL pointer to the random number generator. Because the pointer is NULL, the random number generator will use the default random number generator of the TPM which produces numbers that are as random as the TPM is able to produce.

27.6.3 Entropy for Primary Objects

For a Primary Object, the caller would instantiate a deterministic random number generator (DRBG) and seed the DRBG with a primary seed, a template hash, and a use string. The key generation function would pass this pointer to the random number generator which would use this state instead of the TPM’s default state. This produces a sequence of bits that have as much entropy as the primary seed and which have a property that is required for generating a Primary Object – the DRBG state can be reinstantiated each time the same Primary Object is created.

Choice of the entropy generation for Primary Objects is a vendor option.

NOTE The reference implementation uses a DRBG based on SP800-90A in order to minimize compliance issues.

27.7 Sensitive Area Creation

27.7.1 Introduction

This clause indicates how the TPM creates the sensitive portion of an object (a TPMT_SENSITIVE).

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

The process for computing the contents of a sensitive area is determined by the type of the object, indicated in the *type* field of *template*.

Some of the sensitive area fields may contain data that is provided by the caller. Some of the fields are always provided by the TPM. When a TPM-provided field is in a Primary Object, the TPM-provided data is always derived, in some way, from the associated Primary Seed such that the same Primary Object can be reproduced as long as the associated Primary Seed remains unchanged. For Ordinary Objects, an implementation may either get the TPM-provided data from the RNG or compute the fields of the object as if it were a Primary Object, but with a random number used in place of a Primary Seed.

The performance difference between the two methods of producing asymmetric objects is negligible as the majority of the work is in validating the choices rather than in generating them. For symmetric objects, the difference might be worth having different methods for Primary and Ordinary Objects but there is an added cost in development and testing that could offset the benefit of any slight performance advantage.

For Ordinary Objects, the method used for generating *sensitive* should be used for generating *seedValue*. That is, if *sensitive* is generated by taking values from the RNG, then *seedValue* should be generated by taking values from the RNG. If *sensitive* is generated by creating a random seed and using the methods used for Primary Keys, then that same seed should be used for generating *seedValue*.

27.7.2 type

The *type* parameter of the object's sensitive area is a copy of the type parameter from the object's public-area template.

27.7.3 authValue

The *authValue* of the object is copied from the *userAuth* field of the *inSensitive* parameter of commands such as TPM2_Create(), TPM2_CreateLoaded, or TPM2_CreatePrimary(), or from *newAuth* in commands such as TPM2_ObjectChangeAuth.

When the TPM returns a TPM2B_PRIVATE structure, the TPM pads the TPM2B_AUTH to its maximum size.

NOTE This prevents the TPM from leaking the size of the authorization value in cases where trailing zeros are stripped.

27.7.4 *seedValue*

For a symmetric object, *seedValue* field is used as an *obfuscation* value. It is also used to hold the symmetric seed value for Storage Keys.

For an asymmetric key that is not a Storage Key, *seedValue* is not needed and the TPM will ignore the value if it is present.

For a Storage Key, *seedValue* is used as a seed for generating the integrity and confidentiality values for protecting child objects of the key.

For all object types, when the TPM generates *seedValue*, it is the size of the digest produced by the *nameAlg* of the object.

NOTE 1 Presuming that the protection algorithms of a Storage Key are reasonably balanced (a requirement), then this size of seed will provide adequate entropy required for protection of the child Object.

For an imported symmetric object, *seedvalue* is required to be the size of the digest produced by the *nameAlg* of the object.

For an imported Storage Key, *seedvalue* is required to be at least to be at least half the size of the digest produced by the nameAlg.

NOTE 2 This requirement is for backward compatibility.

For Imported asymmetric non-Storage Keys, *seedValue* is not required.

NOTE 3 The rationale for these requirements derive from the use of *seedValue*. When *seedValue* is used in a hash, it must be the full size. When used in an HMAC, it can be half the size.

seedValue is generated using the “entropy” source used for the object type (see 27.6).

When creating a Primary Object in the Endorsement Hierarchy, it is required that the entropy source be updated to reflect the current SPS. This allows the *sensitiveValue* to remain the same after a change of the SPS but prevents any previously-generated Child Objects in the Endorsement Hierarchy from being loaded after the SPS changes.

NOTE 3 In the reference implementation, this is accomplished by reseeding the DRBG state with the proof value of the storage hierarchy.

27.7.5 sensitive

27.7.5.1 Symmetric Objects

Symmetric objects have a *type* of TPM_ALG_SYMCIPHER or TPM_ALG_KEYEDHASH. For a symmetric object, the sensitive object data may be provided by the caller or generated by the TPM.

If *sensitiveDataOrigin* attribute in the object template is CLEAR, then the sensitive data is provided by the caller. If provided by the caller, the sensitive data will be in the *data* field of the *inSensitive* parameter of TPM2_Create() or TPM2_CreatePrimary(). For TPM2_CreateLoaded(), if the Parent is a Derivation Parent, then *sensitiveDataOrigin* is required to be CLEAR in the template.

If *sensitiveDataOrigin* is SET, it indicates that the TPM is the source of the sensitive data and the data field of the *inSensitive* parameter is required to be an Empty Buffer.

A user provided symmetric key is required to be the size indicated by *parameters.symDetail.keyBits.sym* in the template. It is the number of octets required to hold the number of bits indicated.

NOTE 1 If the key has fewer significant digits than necessary, pad octets of zero are required. The pad octets are added to the high-order end of the key.

A user provided HMAC key is not allowed to be larger than the smaller of the block size of the hash algorithm or 128 octets. Limiting the size to 128 octets is for compatibility of structures between TPM.

NOTE 2 The HMAC algorithm requires that keys larger than the hash block size be hashed before use. This may result in fewer bits of entropy in the HMAC key than expected by the caller. The TPM will not allow the caller to specify an overly large value for the HMAC key. If the caller desires to use a larger value, they should perform the digest externally and pass the resulting digest to the TPM for use as the HMAC key.

If not provided by the caller, *sensitive* is generated by the TPM. For a TPM_ALG_KEYEDHASH object, the size is the digest size of the nameAlg of the object. For a TPM_ALG_SYMCIPHER object, the size is equal to (*parameters.symDetail.keyBits.sym* + 7) / 8.

 Copyright © TCG 2006-2020

27.7.5.2 Asymmetric Objects

The *sensitive* field in an asymmetric key object is the private key. The key is generated in a way that is specific to the algorithm and is described in an algorithm-specific annex of this TPM 2.0 Part 1.

EXAMPLE RSA key generation is described in B.8 and ECC key generation is described in C.5.

27.8 Creation Data and Ticket

When it creates an object, the TPM also creates a data structure that describes the environment in which the object was created. This data includes:

- a digest of selected PCR at the time of object creation and a bit-map indicating the PCR that were included in the list. The PCR selection is those PCR indicated in the call to TPM2_Create() and TPM2_CreatePrimary().
- the locality at which the object was created
- the *nameAlg* of the Storage Parent. If the parent is a Primary Seed, then the algorithm will be TPM _ALG_NULL.
- the Name of the Storage Parent. If the parent is a Primary Seed, then the Name will be the handle of the seed.
- the Qualified Name of the Storage Parent. If the parent is a Primary Seed, then the Qualified Name will be the handle of the seed.
- some additional data provided by the caller that is to be associated with the new object

In addition to these values, the TPM will create a ticket that will allow the TPM to validate that the creation data was generated by the TPM.

The creation data will act as a form of certification of the object that is most useful when *fixedTPM* is CLEAR in the created object. Without this information, it would not be possible to determine how the object came to be in the hierarchy where it is found. When the object is moved, it would be up to the duplication authority to provide some certification of the duplication process. If there is no creation data indicating that the object was created in the place where it was found, and there is no certificate from the duplication authority for the object, then it may be difficult to establish the trustworthiness of the object.

NOTE In this case, the trustworthiness of the object refers to determining that the sensitive area of the object has only ever been accessible by trusted entities such as other TPMs.

27.9 Creation Resources

When a Primary Object is created, it is also loaded in a TPM object slot and the handle is returned. If no free object slot is available, the TPM will return TPM_RC_OBJECT_MEMORY.

When creating an ordinary object, the TPM may use an object slot as scratch memory in which it builds the object. If the implementation does use this scheme and no object slot is available, then the TPM will return TPM_RC_OBJECT_MEMORY.

28 Object Derivation

28.1 Introduction

This section describes the differences between Object creation and Object derivation. If no difference is stated, then there is none.

The TPM2_CreateLoaded() command is used for derivation. This command can be used to create or derive any type of object with the type of Object determined by the type of the entity referenced by the *parentHandle* parameter. If *parentHandle* references a Primary Seed, then a Primary Object is created; if *parentHandle* references a Storage Parent, then an Ordinary Object is created; and if *parentHandle* references a Derivation Parent, then a Derived Object is generated.

NOTE For a given template (*inPublic*), the same Primary Object is created by both TPM2_CreatePrimary() and TPM2_CreateLoaded().

28.2 Derivation Parameters

For object derivation the TPM uses the *sensitive* value in a Derivation Parent as a key in a key derivation function (KDF). The KDF that is to be used in Object derivation is a property of the Derivation Parent and may include the hash algorithm to use in the derivation process.

NOTE **KDFa** (TPM_ALG_KDF1_SP800_108) is the only KDF that is currently supported by the reference code,

Most KDFs require additional parameters in order to have different types of keys derived for different applications. The TPM allows two additional parameters (*label* and *context*) to be provided in TPM2_CreateLoaded(). These additional parameters can be provided in two ways: in the *unique* field of the *inPublic* value, or in the *data* field of the *inSensitive* parameter. If provided in the *unique* field, the corresponding value in the *inSensitive.data* field is ignored.

28.3 Public Area Template

For TPM2_CreateLoaded(), a TPM2B_TEMPLATE is used for the *inPublic* parameter instead of a TPM2B_PUBLIC. The difference in parameters is to allow overloading of the *unique* field in the *inPublic* parameter. For a TPM2B_PUBLIC, the *unique* field is unmarshaled based on the *type* of *inPublic.* For a TPM2B_TEMPLATE, the *inPublic* is unmarshaled as a byte array and passed to the TPM2_CreateLoaded() action code where it is unmarshaled based on the type of parent and *type* of *inPublic.*

When using TPM2_CreateLoaded() to create a Primary or Ordinary Object, the caller should use the same format for the unique field that would be used when creating the Object with TPM2_CreatePrimary() or TPM2_Create(). The derivation-specific format is required when *parentHandle* references a Derivation Parent.

For object creation, *sensitiveDataOrigin* indicates to the TPM whether the caller is providing the sensitive data or if the TPM is to generate it. For Object Derivation, the caller provides values that influence the derivation process, but the caller does not explicitly set the sensitive value. For this reason, *sensitiveDataOrigin* is required to be CLEAR in the template for a Derived Object.

 Copyright © TCG 2006-2020

28.4 Entropy for Derived Objects

28.4.1 Conceptual Description

The ‘entropy’ for a Derived Object is provided by a protected value in the sensitive area of the derivation parent (the sensitive value). That entropy, along with caller-provided values, is used in a KDF to spread the entropy across values in the derived object. Those derived values are the *sensitive* and *seedValues*. The remainder of the Derived Object is provided by the template or computed from the two derived values.

KDFa is used for the derivation. The parameters for the derivation are:

$$\mathbf{KDFa}\ (hashAlg, sensitive, [label,]\ [context,]\ 0, 8192) \qquad (1)$$

where:

hashAlg	the *nameAlg* of the derivation parent
sensitive	the *sensitive* value in the sensitive area of the derivation parent
label	an optional string provided by the caller
context	an optional string provided by the caller

KDFa has a counter and a bits parameter that are set, as shown above, to 0 and 8192 respectively.

NOTE In order to be compliant with SP800-108, the KDFa function will increment counter to 1 before using it in the generation of the first HMAC block.

This call will cause the KDF to generate 1024 bytes of data, with the results of the first digest being the most significant bytes.

During the derivation process, the data is removed from the 1024-byte buffer as needed for each use. The data is used from most significant byte to least significant byte with no bytes skipped. For most key generations, a deterministic number of bytes will be removed for each of the derived fields (*sensitive* and *seedValue*).

Example 1 For a 128-bit AES key in a SYMCIPHER object having SHA-256 as its *nameAlg*, the most significant 16 bytes of the KDF data are used for the AES key and the next-most-significant 32 bytes are used for the *seedValue*.

Example 2 For ECC, the TPM uses the method of FIPS 186-4 B.4.1 Key Pair Generation Using Extra Random Bits. For a 256-bit ECC key, the most-significant 40 bytes are used to generate the private key and, if the *nameAlg* of the derived object is SHA-256, the next-most-significant 32 bytes will be used for the *seedValue*.

In some cases, the number of bytes used for the *sensitive* value is indeterminate. This is because some of the generated values are unsuitable for the application and may need to be discarded. In such cases, bytes are taken from the 1024-byte buffer until suitable values for *sensitive* have been found and the next most significant bytes are used for *seedValue*.

Example 3 Not all 64-bit values are suitable for use as DES keys. When the derivation process produces one of these values, the key will be discarded, and the next most significant bytes are taken from the 1024-byte KDF buffer.

When suitable values for both *sensitive* and *seedValue* have been extracted from the 1024-byte KDF buffer, the remaining bytes are discarded.

 Copyright © TCG 2006-2020

28.4.2 Implementation Alternatives

There are various ways to produce an implementation that is compatible with the conceptual description above without actually having to generate 1024 bytes of data. Some examples are given here:

- An implementation may compute the number of bytes that will be needed to produce the *sensitive* and *seedValue* and the KDF would only need to generate that number of bytes. The bits parameter in the call to generate the data would still need to be 8192. In order to facilitate this type of implementation, the *CryptKDFa*() function in the reference code has a blocks parameter that limits the number of returned blocks, regardless of the size of the *sizeInBits* parameter.
- An implementation may generate blocks on a demand basis. This is fairly complex but allows the derivation process to be used as if it were any other RBG. To implement this process, the calling parameters of the KDF are saved in a structure so that they are available for multiple calls. This structure is passed to the functions that use a random number generator. When random bits are needed, the generator checks the type of the random number generator context and, if it contains KDF parameters, they are used in a call to the KDF. After each call, the counter value is incremented so that the net effect of generating one block at a time is the same as generating all of them at the same time. The additional complexity of this implementation is that it is required that all of the bytes from the KDF be used in order, with none skipped. In order to deal with a call that does not use a full block, a buffer is added to the KDF structure in which residual bytes are saved. When a call is made to fetch bytes from the KDF, the residual buffer is checked first and any bytes in that buffer are returned before the KDF is called to produce additional bytes. If the KDF produces a block and not all bytes are returned, the residual bytes are placed in the buffer. This provides continuity of bytes as required by 28.4.1.

28.5 Derivation Process

The derivation process is required to be the same for all TPMs. That is, with the same inputs, all TPMs will generate the same Derived Objects.

When generating a Derived Object, the TPM will create the entropy structure for a KDF and pass a pointer to the structure to the function that creates Objects. The algorithm for generating an Object is as described in clause 27.7.

NOTE The method of generating RSA keys is highly variable and is normally chosen according to the constraints of the application. In some cases, compliance is the overriding factor and in others, performance may be the determining factor. Since no single algorithm seems to be optimum for all the constraints and it would not be acceptable to require that TPMs implement one RSA key generation for compliance and one for interoperability, the TCG has chosen not to support derivation for RSA keys.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

29 Object Loading

29.1 Introduction

An object is either a key or data that can be loaded into the TPM for use. An object must be loaded before the TPM can use or modify the object. Loading may require that the USER role authorization for the Storage Parent be provided

29.2 Load of an Ordinary Object

It is possible to load just the public portion of an object into the TPM (TPM2_LoadExternal()) or to load both the public and private portions (TPM2_Load()). If the sensitive area is to be manipulated or used, then both portions are required to be loaded.

When loading an object, multiple consistency checks are performed. Among these checks:

a) Is the HMAC of the encrypted private area correct – this ensures that the sensitive area was not modified, that the sensitive area and the provided public area are matched, and that the object is a descendant of the Storage Parent.

b) Is the unique parameter of the public area cryptographically bound to the sensitive data – this is required to prevent improper association of a public area with a sensitive area. If this check were not done, an attacker could use a public area that had a Name that is the same as a different object and associate a different sensitive area with the public area. If the object were used in TPM2_PolicySecret(), the attacker could get the TPM to create a *policyDigest* with any desired hash value.

   EXAMPLE A legitimate policy uses signature validation of a key with Name1. An attacker could create an object with Name1 (copy the data from the legitimate key) and then create a sensitive area that had an *authValue* known to the attacker, instead of using TPM2_PolicySigned() to create the policy.

c) Are the attributes consistent – these values need to be checked even if the integrity check indicates that the values were not modified. This is because the object may have been created by software using inconsistent values. The integrity may be good, but the values may be wrong.

   1) If *fixedTPM* is SET, *fixedTPM* must also be SET in the Storage Parent.

      NOTE If *fixedTPM* is properly SET, then the other checks need not be made because the object is verified to have been created on the TPM that loaded the object, so the other attributes are known to be correct.

   2) If *fixedParent* is CLEAR, then *fixedTPM* must also be CLEAR.

   3) If *restricted* is SET, only one of *sign* or *decrypt* may be SET.

29.3 Public-only Load

There are several cases, such as duplication or signature verification, when only the public portion of an asymmetric key can be loaded. The public-only load of an object requires that the caller associate the object with one of the hierarchies. This association is needed when the key is used for signature verification so that the TPM can determine which proof value to use in the ticket.

A public-only load occurs when the *inPrivate* parameter to TPM2_LoadExternal() has a size of zero.

29.4 External Object Load

External Objects allow the cryptographic processes of the TPM to be used on keys that are not part of a TPM hierarchy. The public portion of an asymmetric key may be loaded so that the TPM can be used to validate a signature. A symmetric key may be loaded so that the symmetric engines of the TPM may be used to encrypt or decrypt data.

TPM2_LoadExternal() is used to load an External Object. When only the public portion is loaded, the attributes of the object are arbitrary, but the structures are required to be consistent with the type. That is, if an RSA signing key is loaded, the signing scheme must be a valid scheme for an RSA key.

When the sensitive portion of the object is loaded (such as, a symmetric key), the sensitive area is not encrypted by a Storage Parent but may be encrypted using parameter encryption. The *fixedParent* and *fixedTPM* attributes are required to be CLEAR when both parts are loaded. This check allows the object to be used in any command that is valid for the type including certification.

NOTE If an entity has access to both the public and sensitive portions of a key, then the entity could import the key and then certify it.

An external object can be associated with a hierarchy when it is loaded. This allows creation of tickets that are specific to a hierarchy in commands such as TPM2_VerifySignature().

If the hierarchy with which an External object is associated is disabled, the object will be flushed. If the associated hierarchy is disabled when TPM2_LoadExternal() is called, the object will not load.

 Copyright © TCG 2006-2020

30 Context Management

30.1 Introduction

To allow the TPM to be shared among many applications, the TPM supports context management. The objects, sequence objects, and sessions used by an application may be loaded into the TPM when needed and saved when a different application is using the TPM. The TPM Resource Manager (TRM) is responsible for swapping the contexts so that the necessary resources are present in the TPM when needed.

There are two types of contexts: those associated with Transient Objects, and those associated with authorization sessions.

The four commands used to manage the contexts are

1) **TPM2_ContextSave()** – the TPM integrity protects, encrypts, and returns the context associated with a handle,
2) **TPM2_ContextLoad()** – allows a previously saved context to be loaded to TPM RAM and have a handle assigned,
3) **TPM2_FlushContext()** – the context information associated with the specified handle is erased from TPM RAM, and
4) **TPM2_EvictControl()** – allows the owner or the platform firmware to designate objects that are to remain TPM-resident over TPM2_Startup() events. This command will return a new handle.

A saved context is cryptographically bound to a specific TPM so that it may not be loaded on a different TPM. This binding is provided by using a statistically unique proof value in the generation of the protection values for a context (see 30.3 and 30.3.2). When the proof value of a hierarchy changes, saved object contexts belonging to that context can no longer be loaded into the TPM. The proof value for a context will change when its Primary Seed changes. Additionally, *ehProof* will change when either the SPS or EPS changes.

NOTE 1 In the reference implementation, *ehProof* is a non-volatile value from the RNG. It is allowed that the *ehProof* be generated by a KDF using both the EPS and SPS as inputs. If generated with a KDF, the ehProof can be generated on an as-needed basis or made a non-volatile value.

Saved contexts for all objects and sessions are invalidated on a TPM Reset. In the reference implementation, the encryption keys for contexts are changed by TPM Reset so previously saved contexts may no longer be loaded. Saved session contexts remain valid until the session is closed, or TPM Reset. If the *stClear* attribute of an object is SET, then saved contexts for the object are invalidated on either TPM Reset or TPM Restart (that is, any time the TPM does a Startup(CLEAR). If the *stClear* attribute of an object is CLEAR, then the saved contexts for that object are valid and may be loaded into the TPM until the next TPM Reset.

NOTE 2 In the reference design, when an object context is saved, the current value of *clearCount* is placed in the context. When the context is loaded, if the object is a *stClear* object, the value in the object is compared to the current value of *clearCount*. If they are not the same, then the context load fails.

Objects and sessions are not retained in TPM memory after a TPM2_Startup() and it is necessary for the TRM to save the contexts for any session or object that is to be useable after TPM Restart or TPM Resume.

NOTE 3 The TPM might lose power between a TPM2_Shutdown(TPM_SU_STATE) and the subsequent TPM2_Startup(). With respect to context preservation, the TPM behavior is defined to be the same whether the TPM loses power or not.

 Copyright © TCG 2006-2020

The structure of a saved context TPM2B_CONTEXT_DATA may be defined by the vendor, but a saved context is required to have its integrity and confidentiality protected by cryptographic means. Parts 3 and 4 of this specification implement the normative requirements for providing confidentiality and integrity protection for saved contexts. These protections are described in more detail in subsequent parts of this clause 30.

NOTE 4 The algorithms chosen for integrity and confidentiality protection of a saved context are vendor specific. However, the cryptographic strengths of the algorithms used are required to be the highest of any algorithm of the same type implemented on the TPM.

30.2 Context Data

30.2.1 Introduction

The data structure TPMS_CONTEXT returned by TPM2_ContextSave contains context metadata as well as the actual context TPM2B_CONTEXT_DATA. The context metadata contains:

- a sequence number,
- a handle, *savedHandle*, and

NOTE For Transient Objects, this *savedHandle* in a saved context data structure is not the same as the handle used by the TPM to reference loaded objects and by TPM commands to describe the object being operated on.

- a hierarchy selector.

The actual context contains:

- an integrity HMAC, and
- an encrypted data blob.

The structure of the metadata is normative. The internal structure TPMS_CONTEXT_DATA of the actual context is vendor specific. The encrypted data blob contains the data necessary to reconstruct the full object or session context in the TPM. The other fields are defined in the remainder of this clause 30.2.

The structure of the context contains both confidential and non-confidential data. This specification requires encryption of the confidential data. The TPMS_CONTEXT structure is normative. The structure of the enclosed TPMS_CONTEXT_DATA is vendor-specific, and its confidential data must be encrypted.

30.2.2 Sequence Number

New protection values are generated each time a context is saved. The protection values are an HMAC key, a symmetric key, and an initial value. The values are made unique by including a counter value in the generation process (see 30.3 and 30.3.2). The counter value used for the context is stored in the sequence number field of the context structure. Two counters are used for generating the sequence numbers. One counter is used for transient and sequence object contexts. A second counter is used for session contexts.

There are two counters used to provide sequence numbers. The counter (*objectContextID*) provides sequence numbers for Transient Objects. This counter is incremented each time an object context is saved. The counter (*contextCounter*) is used to provide sequence numbers for sessions and increments when a session context is created or loaded (its behavior is described in more detail in 30.5). When creating the context structure, the TPM sets the *sequence* parameter to the value of the counter used in the generation of the protection values for the context.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

When a context is loaded, (TPM2_ContextLoad()), the TPM checks that the *sequence* parameter is in a viable range before starting the operation. For an object, the viable range is any number that is less than the current value of the object sequence counter. For a session, the sequence number must also be less than the session sequence number, but it must also be greater that the sequence number minus the allowable range for session sequence number.

In the reference implementation, *objectContextID* is a 64-bit counter that is initialized to zero at startup and is expected to never overflow. The size is platform-specific.

EXAMPLE For purposes of this example, assume that the sequence counter value is only 16 bits and that the session counter indicates the last assigned session context had a value of $10\ 10_{16}$. It would then be an error if the *sequence* parameter in a loaded session context is greater than $10\ 10_{16}$. Assume further that the TPM only allows a range of 256 between session values (explanation in 30.5). Then it would be an error if the sequence parameter of the session in TPM2_ContextLoad() is less than $10\ 10_{16} – 01\ 00_{16} = 0F\ 10_{16}$; and the TPM will not load the context.

30.2.3 Handle

The *savedHandle* number for a context indicates the type of the context (object or authorization session). The type of the context is used to determine how to reconstruct the protection values for validation of the context. If the *savedHandle* value in the context is changed by software, the context will not load.

For a session, the same handle is assigned to the context whether the context is loaded in the TPM or in a saved context. That is, *savedHandle* is the same as the handle the TPM uses to refer to the session. A session handle will have an MSO of TPM_HT_HMAC_SESSION (02_{16}) or TPM_HT_POLICY_SESSION (03_{16}). The range of values in the handle index (the low-order three octets of the handle) is TPM dependent. In the reference implementation, the low order bits of the session context handles fall within a range from 0 to MAX_ACTIVE_SESSIONS – 1 and the TPM will generate an error and do no further processing of the context if the handle is outside of this range.

A *savedHandle* MSO of TPM_HT_TRANSIENT (80_{16}), indicates that the context is an Object or sequence object. For an object, the *savedHandle* parameter of the context structure does not indicate the handle value used by the TPM to reference the object (when a Transient Object context is not on the TPM, the TPM retains no information about that context). Therefore, the *savedHandle* value is not used for Transient Object contexts in the same way that it is used for session contexts. Instead, the *savedHandle* is used to indicate the type of the Transient Object context.

Three *savedHandle* values are defined for Transient Object contexts:

1) $00\ 00\ 00_{16}$ – indicates a Transient Object that does not have the *stateClear* property;

NOTE An Object has the *stateClear* property when *stClear* is SET in the Object or in any of its ancestor keys.

2) $00\ 00\ 01_{16}$ – indicates a sequence Object (see 32.4.5); and
3) $00\ 00\ 02_{16}$ – indicates a Transient Object that has the *stateClear* property.

EXAMPLE A sequence Object will have a 32-bit handle value of $80\ 00\ 00\ 01_{16}$.

If the *savedHandle* type is TPM_HT_TRANSIENT, the TPM will not generate or load a context with any other value besides the three values described above for the handle's index.

Objects that have the *stateClear* property are invalidated by Startup(CLEAR). To enforce this, the TPM will include *clearCount* in the integrity value of the Object.

TPM processing of contexts with *savedHandle* values of $80\ 00\ 00\ 00_{16}$ or $80\ 00\ 00\ 01_{16}$ is the same. The reason for differentiating sequence Objects is to identify the context for the convenience of the TPM

resource manager (TRM). The TRM needs to manage sequence objects differently from other Transient Objects. Because the context of a sequence object changes each time the sequence is updated, the context needs to be saved each time the context is used. The context of a Transient Object does not change on use. Therefore, the TRM can optimize by saving the Transient Object context only once.

30.2.4 Hierarchy

The hierarchy parameter of the context indicates which of the hierarchy proof values are used in the creation of the protection values for the context. For objects, this value is determined by the hierarchy of the object and may be TPM_RH_NULL for a Temporary Object. Sequence objects and sessions are in the Null hierarchy.

30.3 Context Protections

30.3.1 Context Confidentiality Protection

A symmetric block cipher is used to protect the confidentiality of a saved context. The algorithm is selected by the TPM vendor but is required to have the highest security strength of any symmetric block cipher implemented on the TPM.

When the context is created by TPM2_ContextSave(), the value of *sequence* is stored in the TPM2B_CONTEXT_SENSITIVE context before it is encrypted. When the context is loaded, the value of *sequence* is compared to the value in the loaded TPM2B_CONTEXT_SENSITIVE context after it is decrypted. If the values are not the same, then the TPM will enter failure mode as this is symptomatic of a specific type of power analysis attack.

The symmetric key and IV are regenerated when a context is loaded. It is required that the symmetric key and IV not be generated until the context integrity has been validated.

NOTE 1 This restriction prevents simultaneous power-analysis attacks on the integrity and encryption values of a context. Since the integrity is checked first, no attempt is made to create the symmetric key if the integrity check fails.

KDFa() is used to generate the symmetric encryption key and IV for context encryption. The parameters of the call are:

$$(symKey, symIv) \coloneqq \mathbf{KDFa}\,(hashAlg, hProof, vendorString, sequence, handle, bits) \quad (50)$$

where

hashAlg	a hash algorithm chosen by the vendor
hProof	the proof value associated with the hierarchy associated with the context
vendorString	a value used to differentiate the uses of the KDF
sequence	the sequence parameter of the TPMS_CONTEXT
handle	the handle parameter of the TPMS_CONTEXT
bits	the number of bits needed for a symmetric key and IV for the context encryption

NOTE 2 The value of *vendorString* is required to be different from any other label string used in a **KDFa**() call. The reference implementation uses "CONTEXT_ENCRYPT"

NOTE 3 The *nullProof* is used as the hProof value for a context in the Null hierarchy so that the encryption keys do not repeat and so that they change on each TPM Reset.

 Copyright © TCG 2006-2020

The key and IV produced in (50) are used to encrypt the object or session context

$$encContext \coloneqq \mathbf{CFB}_{symAlg}\,(symKey, symIv, context) \qquad (51)$$

where

$\mathbf{CFB}_{symAlg}$	symmetric encryption in CFB mode using a symmetric algorithm chosen by the TPM vendor
$symKey$	symmetric key from (50)
$symIv$	IV from (50)
$context$	the context being protected (a TPM2B_CONTEXT_DATA)

NOTE 4 The *size* field and the *buffer* field of *context* are encrypted.

30.3.2 Context Integrity Protection

The integrity of a saved context is protected by an HMAC using a hash algorithm selected by the TPM vendor. The hash algorithm chosen is required to have the highest security strength of any hash algorithm implemented on the TPM (see the description of TPM_PT_CONTEXT_HASH in TPM 2.0 Part 2).

The HMAC is constructed using the proof value associated with the hierarchy to which the object belongs. Since the proof value changes when the associated Primary Seed changes, HMAC validation for a previously saved context will fail when the associated Primary Seed changes; and that context may no longer be loaded. Other values in the HMAC computation serve to invalidate other context subsets without necessarily invalidating them all.

EXAMPLE The *clearCount* value is included in the HMAC of a context for an object with the *stClear* attribute so that the context will be invalidated on each TPM Restart as well as each TPM Reset.

The only TPM state change that invalidates all saved contexts is TPM Reset.

Sessions, Sequences, and Temporary Objects are in the “null” hierarchy.

The HMAC integrity computation for a saved context is:

$$contextHMAC \coloneqq \mathbf{HMAC}_{vendorAlg}\,(hProof, resetValue \,\{\, || \, clearCount\} \\ || \, sequence \, || \, handle \, || \, encContext) \qquad (52)$$

where

$\mathbf{HMAC}_{vendorAlg}$	HMAC using a vendor-defined hash algorithm
hProof	the hierarchy proof as selected by the hierarchy parameter of the TPMS_CONTEXT
resetValue	either a counter value that increments on each TPM Reset and is not reset over the lifetime of the TPM; or a random value that changes on each TPM Reset and has the size of the digest produced by vendorAlg
clearCount	a counter value that is incremented on each TPM Resume and may be incremented or set to zero on TPM Reset. This value is only included if the handle value is $80\ 00\ 00\ 02_{16}$.

NOTE the handle value is $80\ 00\ 00\ 02_{16}$ when the *stClear* attribute of the object is SET or when the *stClear* attribute is set in one of the object's ancestor keys.

 Copyright © TCG 2006-2020

sequence	the sequence parameter of the TPMS_CONTEXT
handle	the handle parameter of the TPMS_CONTEXT
$encContext$	the encrypted context blob

30.4 Object Context Management

When an object's context is saved, a copy of the object context is integrity protected, encrypted, and returned to the caller. The original context remains in the TPM and the TPM retains its handle. A saved object context may be reloaded into the TPM with TPM2_ContextLoad(). If the TPM has sufficient memory available, it will load the object and assign a handle. If other copies of the same object are in TPM memory, they are unaffected. An object context is only removed from TPM memory with TPM2_FlushContext(), deletion of the associated hierarchy seed, or TPM2_Startup().

The handle assigned to an object when it is loaded may not be assigned to any other TPM resource, object, or session. When the object is flushed from TPM memory, its handle may be assigned to another TPM resource when it is loaded or created.

Software may create as many copies of an object context as desired. When an object is not in TPM memory, it has no associated handle. If an object context is saved and subsequently reloaded, it is likely that a different handle will be assigned to the object.

When the Primary Seed is changed for the hierarchy associated with an object, all objects associated with that hierarchy are flushed from TPM memory. The TPM will no longer load saved contexts associated with the previous Primary Seed.

When an attempt is made to load an object or an object context (TPM2_Load(), TPM2_CreatePrimary(), TPM2_LoadExternal() or TPM2_ContextLoad()) and the TPM does not have sufficient RAM to hold the object, the TPM will return TPM _RC_OBJECT_MEMORY or TPM _RC_MEMORY. This warning code is normally handled by the TRM. It indicates that an object or a session needs to be unloaded from TPM memory before the command can complete. If the TPM returns TPM_RC_OBJECT_MEMORY, it indicates that an object must be flushed from TPM memory. If the TPM returns TPM_RC_MEMORY, then it is possible that removal from TPM RAM of either an object or a session would allow the command to complete.

When a command references a persistent object, the TPM may move the object from NV into an object slot. If no slot is available, the TPM will return TPM_RC_OBJECT_MEMORY.

An implementation is allowed to use an object slot for temporary memory in execution of TPM2_Import() and return TPM_RC_OBJECT_MEMORY if a slot is not available.

If the TPM uses an object slot for temporary memory, the slot will be freed at the end of the command in which the slot was allocated.

If a TPM receives Shutdown(STATE) before the _TPM_Init, then the saved object contexts will continue to be usable after a TPM Restart or TPM Resume. An exception is that an object may be created with the *stClear* attribute. If this attribute is SET in an object or an ancestor of an object, then the saved context will be invalidated on TPM Restart. All saved object contexts are invalidated by TPM Reset.

30.5 Session Context Management

A session context is created by TPM2_StartAuthSession(). The context associated with a session is unique. That is, the data describing the session's state may be either on the TPM or saved off the TPM, but not both. Further, a saved session context may only be loaded once. These limitations on the session context are intended to prevent possible attacks based on replay of authorizations.

 Copyright © TCG 2006-2020

The handle associated with a session does not change as long as the session is active. The session is active until closed by the *continueSession* flag being FALSE or until the session context is flushed from the TPM by TPM2_FlushContext().

The nominal implementation uses a volatile counter (*contextCounter*) that increments each time a session is created or context loaded. This count value is assigned to the created or loaded session context and serves as a version number for the session context. If the session context is saved and reloaded, it is assigned a new version number. *contextCounter* is saved by Shutdown(STATE) and reset on TPM Reset.

The TPM maintains a database of concurrent sessions so that it can validate that a reloaded session context is the most recent version. It is required that the TPM be able to ensure that the restored context is the correct context regardless of the number of contexts created.

The size of *contextCounter* affects the size of the memory required for tracking each of the contexts. It is therefore desirable that the counter only be large enough for the majority of applications, meaning that it will not be large enough for all applications. In those applications, a method is required to handle counter rollover.

One scheme for handling rollover is to maintain an even/odd interval. If, for example, a nonce were being used for each interval, then the TPM could maintain two nonces, one to be used when the MSb of the volatile counter is 0 and the other when the MSb is 1. When the counts of all the sessions have the same MSb, then a new nonce can be created for use when the MSb changes. This scheme works unless a session has a long lifetime. That is, if the session is created when the MSb is 0, and the session is still active when the counter reaches its maximum value with all bits equal 1, then the context with an MSb of 0 will need to be discarded.

Rather than have the old session be automatically flushed, the TPM provides an indication that it is reaching its limit and that one or more saved session contexts need to have their *sequence* number updated to the current interval in preparation for the context counter rollover.

The indication that the context counter is approaching its limit is provided when an authorization session is created or loaded. If the creation or loading of a session would make it impossible for the TPM to bring all contexts into the current interval, then it would return an error (TPM_RC_CONTEXT_GAP) and not create or load the new session. On receiving this error, the management software either would explicitly flush old session contexts or would load the old session contexts to update their associated counter values.

When the TPM returns TPM_RC_CONTEXT_GAP, it will not allow an authorization session to be created and it will only allow the oldest authorization session to be loaded. When the oldest session is loaded, its *sequence* number is updated. It may be used or saved with its new *sequence* number.

NOTE The TPM must provide the indication of the session-tracking limit being reached before the maximum count is reached. If there are three sessions in the 'odd' interval and the end of the 'even' interval is being reached, then the TPM must indicate the limit while there are still three available session sequence numbers in the 'even' interval. This allows the sessions in the 'odd' interval to be loaded and saved with an 'even' interval session sequence number and with no session in the 'odd' interval so that a new 'odd' interval identifier can be created.

Session contexts in TPM RAM are flushed on any TPM2_Startup(). Saved session contexts are not invalidated and may be reloaded after a TPM Restart or TPM Resume. Saved session contexts are invalidated on a TPM Reset.

30.6 Eviction

Eviction is the process of removing the context associated with an object or session from TPM RAM to allow for other sessions or objects to be loaded or created. Saving a session context removes the majority of the session context from TPM RAM. Saving an object context does not remove it from TPM

 Copyright © TCG 2006-2020

memory. When applied to an object, TPM2_FlushContext() will remove it from the TPM RAM but not invalidate the saved contexts of that object. When applied to a session, TPM2_FlushContext() will invalidate the session whether its context is in TPM RAM or saved.

An object may be copied to persistent TPM NV memory with TPM2_EvictControl(). When made persistent, TPM2_FlushContext() and TPM2_Startup(TPM_SU_CLEAR) have no effect on the persistent copy of the object.

A session may not be made persistent.

Use of TPM2_EvictControl() requires either Owner Authorization or Platform Authorization. An object made persistent using *ownerAuth* may be evicted from persistent memory using either Owner Authorization or Platform Authorization. An object made persistent using Platform Authorization may only be evicted from persistent memory using Platform Authorization.

30.7 Incidental Use of Object Slots

In most cases, the TRM will explicitly load and unload (flush) objects from the TPM's object memory. In three cases, the TPM will make use of object slots as a side effect and the TRM needs to deal with potential resource issues that may arise. The three cases are: TPM2_Import(), use of persistent objects, and _TPM_Hash_Start.

TPM2_Import() allows an implementation to use an object slot for its "scratch" memory while operating on the import blob. When the command completes the slot will be available. An implementation that uses this option may return TPM_RC_OBJECT_MEMORY if a needed slot is not available. This return code is in the group of response codes that are expected to be handled by the resource manager.

When a handle references a persistent object, a TPM implementation is allowed to return TPM_RC_OBJECT_MEMORY if an object slot is not available. This allows the TPM to keep the persistent image of the object in a compressed form and decompress it into an object slot for efficient processing. The version of the persistent object held in an object slot will be removed when the command completes.

When the TPM receives _TPM_Hash_Start, it will unconditionally create an Event Sequence context. If an object slot is available, the TPM will use the available slot. If an object slot is not available, the TPM will flush an arbitrary object context and use that slot. At the end of the event sequence (_TPM_Hash_End), the slot used for the Event Sequence will be vacant. The TRM should be aware that the _TPM_Hash_Start sequence may cause loss of a loaded object.

31 Attestation

31.1 Introduction

Attestation is the action of having the TPM sign some internal TPM data. Confidence in the attestation is related to the confidence in the key that is used to sign. The highest confidence is provided by a *fixedTPM*, restricted signing key that is created on a TPM with a certificate from the TPM manufacturer.

The TPM may be used to attest to several different types of data:

- PCR data – TPM2_Quote()
- *Clock* and *Time* data – TPM2_GetTime()
- Audit digests – TPM2_GetCommmandAuditDigest() and TPM2_GetSessionAuditDigest()
- Other TPM Objects – TPM2_Certify()

For all of these commands, the TPM produces a standard attestation structure and appends the command-specific data. The resulting data block is then hashed and signed by the selected signing key. The selected key may be any key that has the *sign* attribute SET. If the signing key is unrestricted, then the caller may indicate the signing scheme to be used. If the signing key is restricted, the TPM will return an error (TPM_RC_SCHEME) unless the scheme selector in the attestation command is TPM_ALG_NULL.

31.2 Standard Attestation Structure

The contents of the standard attestation structure are described in Table 30.

Table 30 — Standard Attestation Structure

Parameter	Type	Description
magic	TPM_GENERATED	This unique value (TPM_GENERATED_VALUE) occurs as the first octets in any TPM-generated attestation structure. This field is used to prevent use of a restricted signing key to sign a forgery of an attestation. A TPM will not allow a restricted signing key to sign any external data if that data starts with this unique value. The way that the TPM enforces this restriction is that a TPM will not use a restricted key to sign a digest that the TPM did not produce. Since the TPM produced the digest, it can ensure that any external data did not start with this value.
type	TPMI_ST_ATTEST	This identifies the type of the attestation structure and indicates the contents of the *attested* parameter.
qualifiedSigner	TPM2B_NAME	This is the Qualified Name of the key used to sign the attestation data. A key that can be duplicated may be signing in different locations and this Qualified Name allows the Verifier to determine the environment in which the signature was produced.
extraData	TPM2B_DATA	external info supplied by caller (often in *qualifyingData* parameter) NOTE A TPM2B_DATA structure provides room for a digest and a method indicator to indicate the components of the digest. The definition of this method indicator is outside the scope of this specification.
clockInfo	TPMS_CLOCK_INFO	The values of *Clock*, *resetCount*, *restartCount*, and *Safe*
firmwareVersion	UINT64	This TPM-vendor-defined value changes when the firmware on the TPM changes, if that change is meaningful to the security of the TPM.
[type]attested	TPMU_ATTEST	the type-specific attestation information

 Copyright © TCG 2006-2020

31.3 Privacy

The attestation block contains information that could allow cross correlation of attestation values. The combination of a *firmwareVersion* and *clockInfo* could be used to identify that two attestations were signed by keys on the same TPM. This correlation is possible because the combination of *resetCount*, *restartCount*, and *firmwareVersion* could be unique. Even if the combination is not unique for all TPM, an imperfect correlation may be adequate for certain types of activity tracking.

The TPM prevents such tracking by adding obfuscation values to the reported values of *resetCount*, *restartCount*, and *firmwareVersion*. This obfuscation value is different for each key and TPM (see 36.7). Although the values are obfuscated, they do not lose any of their usefulness for indicating changes to the values. While the absolute values are not visible in the attestation, it is still possible to look at attestations signed by the same key and determine how many times the TPM was reset or restarted between the attestations and to see the delta in the firmware version number (if any).

It is sometimes necessary to have the non-obfuscated values of the *clockInfo* and *firmwareVersion* included in an attestation. Support for this is provided by allowing signing keys in the Endorsement hierarchy. When a key in the Endorsement hierarchy signs an attestation, no obfuscation is applied. The underlying presumption is that the TPM's Privacy Administrator controls the Endorsement hierarchy and it is possible, through policy, to limit the use of keys in that hierarchy so that authorization from the Privacy Administrator is always required.

31.4 Qualifying Data

Each of the attestation commands has a parameter called *qualifyingData*. This parameter is not interpreted by the TPM and may contain any data chosen by the caller. The most common use of this parameter is expected to be as a nonce to ensure "freshness" of an attestation.

31.5 Anonymous Signing

If an anonymous scheme (TPM_ALG_ECDAA) is used for signing in any attestation command, the *qualifiedSigner* parameter will be an Empty Buffer.

NOTE 1 If the *qualifiedSigner* field was properly populated (not the Empty Buffer), then the unique identity of the signing key would be disclosed.

For TPM2_Certify() using an anonymous signing scheme, both the *qualifiedSigner* and *qualifiedName* of the certified key are set to an Empty Buffer.

NOTE 2 If the *qualifiedName* field was not cleared, then it would be possible to establish a hierarchical relationship between to certified objects. This is not desirable for an anonymous scheme.

31.6 X.509 Certificate Signing

TPM2_CertifyX509() signs an X.509-formatted certificate. Prior to constructing and signing an X.509 certificate, TPM2_CertifyX509() verifies that the key-to-be-certified is loaded in the TPM, and that some of the permissions in the key's proposed X.509 certificate are compatible with the key-to-be-certified.

A typical use of TPM2_CertifyX509() is the enrollment of any TPM key into an X.509 Public Key Infrastructure. TPM2_CertifyX509() signs an X.509 formatted certificate describing the TPM key, rather than enabling the TPM key to self-certify by creating an X.509 formatted Certificate Signing Request (CSR). This is because a CSR doesn't provide a way for a TPM to communicate to a Certification Authority that the TPM key is protected by a TPM, or the restraints on the TPM key that are enforced by the TPM. The X.509 formatted certificate signed by the TPM is inspected by a Certification Authority prior to the CA signing its own certificate describing the operations on the key that are approved by the CA. A simpler CA will use conventional X.509 methods to just verify the TPM's certificate and verify the X.509

 Copyright © TCG 2006-2020

certificate of the key that signed the TPM's certificate. A more sophisticated CA that understands TPMs will also interpret the TPM certificate's Extensions element, which describes the certified key's detailed TPMA_OBJECT attributes and indicates precisely what operations a TPM can and cannot perform on the certified key.

In a typical usage of TPM2_CertifyX509():

- First, the CA (or its proxy) loads a certifying signing key in the TPM. The certifying signing key is typically a key with the x509Sign attribute SET.

- Next, the CA uses the certifying key with TPM2_CertifyX509() to certify a TPM key . The DER-encoded partialCertificate parameter for TPM2_CertifyX509() describes the validity time for the certificate as well as the X.509 Name for the certifying and certified keys. Additionally, partialCertificate is required to contain a KeyUsage in the Extension field and may contain a TPMA_OBJECT Extension. TPM2_CertifyX509() verifies that the key is compatible with approved operations, constructs a complete X.509 RFC 5280-defined certificate, and signs that complete certificate.

- TPM2_CertifyX509() returns the part of the complete X.509 certificate that was constructed by the TPM, plus the TPM's signature over the complete certificate. This data and partialCertificate are assembled outside the TPM into a complete X.509 RFC 5280-defined signed certificate for inspection by the Certification Authority using conventional X.509 tools.

- Finally, the Certification Authority recertifies the TPM key using the CA's normal certifying key, for consumption by entities that trust the CA.

Certification Authorities should be wary of certificates signed by TPM2_CertifyX509() with keys that do not have the x509sign attribute SET, because an X.509 certificate can be signed using TPM2_Sign(). In that case, the TPM will not have verified that the certified key is loaded in the TPM and will not have verified that the certified key is compatible with the X.509 certificate. Nevertheless, an X.509 certificate signed by TPM2_CertifyX509() with an ordinary signing key or a restricted signing key may be acceptable when the CA trusts the entity controlling usage of that signing key: the CA itself may have exclusive control over the signing key, for example.

An x509sign key is even more restricted than a restricted key: a restricted key won't use TPM2_Sign() to sign any data that starts with the "magic" parameter, and an x509sign key won't use TPM2_Sign() to sign any data at all. Hence the command TPM2_CertifyX509() is the only way that an x509sign key can sign an X.509 certificate. An x509sign key will also sign any of the non-X.509 certification commands (e.g., TPM2_Certify, TPM2_CertifyCreation, TPM2_NV_Certify), which sign data that start with the TCG "magic" value. This enables an x509sign key to be an Attestation Key with a certificate proving that the x509sign key is bound to a single TPM and is safe because none of the certification commands sign data that could masquerade as a fraudulent certificate.

TPM2_CertifyX509() verifies that a key-to-be-certified is compatible with permissions granted by the X.509 certificate. Broadly speaking, the TPM verifies that the private key can decrypt if the certificate says the key is approved for decryption and verifies that the private key can sign if the certificate says the key is approved for signing. The TPM verifies that a key has its fixedTPM attribute SET if the certificate approves the key for nonrepudiation/contentCommitment operations, because only the TPM that has a particular fixedTPM key can use that particular key.

TPM2_CertifyX509() cannot guarantee that a key will perform only the operations approved by an X.509 certificate. A TPM cannot control operations on the public part of a key and doesn't usually know why is key is being used, whereas a certificate can approve operations on the public part of a key and can approve the objective of operations on a key. Relying parties are cautioned that not all of the information in a certificate signed using TPM2_CertifyX509() is validated by the TPM. The TPM validates the public key of the target, the KeyUsage, and the TPMA_OBJECT extended attributes.

Some partialCertificate inputs to TPM2_CertifyX509 cannot be verified by the TPM but must be present because they are necessary to create a signature over an X.509 certificate. One such input is the certifying signing scheme OIDs that are necessary to create the signature. The caller does not need to supply OIDs if the TPM is able to generate the OIDs for the certifying signing scheme. However, there are so many OIDs that the TPM is unable to generate all possible OIDs, and some OIDs have not yet been assigned. Thus, the caller must supply the OIDs for the certifying signing scheme if the TPM is unable to generate them. Should the caller supply OIDs, they take precedence over any OIDs that the TPM would have assigned.

 Copyright © TCG 2006-2020

32 Cryptographic Support Functions

32.1 Introduction

In TPM 1.2, the cryptographic primitives were not exposed for general purpose use. For example, the RSA engine could not be used for exponentiation. This specification provides commands that allow access to the primitive cryptographic processes of the TPM.

One assumption in TPM 1.2 was that the host processor usually had much greater performance than the processor used for the TPM so there was no point in having the TPM do something that the host could do much faster. In addition, TPM 1.2 was a passive device with limited bandwidth. While it is true that the host processor will usually have more capability than the TPM, this will not be true in all cases. In fact, on some systems, the main processor will be able to switch execution environments and perform the TPM operations. In others, the TPM may be built around a cryptographic coprocessor that has significantly greater processing capability for cryptographic operations than the host. These higher performance implementations will not be performance-limited by being attached to the system with a low-bandwidth interface. These performance differences mean that exposure of the cryptographic primitives of TPM 2.0 makes more sense that it did in TPM 1.2.

Another reason to make the cryptographic primitives available is that not all software will implement all the algorithms that may be in the TPM. For example, a BIOS may not implement the RSA algorithm but would want to check the RSA signature of some code.

This clause describes the commands and methods that may be provided by a TPM compliant to this specification.

32.2 Hash

TPM2_Hash() will create a digest of a block of data using the indicated hash algorithm. If the amount of data to be hashed exceeds that input buffer size of the TPM, then a hash sequence is used (see 32.4).

If the data used to create the digest does not have TPM_GENERATED_VALUE as its first octets, then the response to TPM2_Hash() or TPM2_SequenceComplete() will contain a ticket indicating that the digest may be signed with a restricted signing key.

NOTE The creation of the ticket may be suppressed by using TPM_RH_NULL as the hierarchy parameter in TPM2_Hash() or TPM2_SequenceComplete().

32.3 HMAC

TPM2_HMAC() will compute an HMAC over a block of data using a TPM-resident value for the HMAC key. In this command, the handle parameter is required to reference an object with a *type* of TPM_ALG_KEYEDHASH with the *sign* attribute SET.

32.4 Hash, MAC, and Event Sequences

32.4.1 Introduction

When the amount of data to be included in a digest cannot or will not be sent to the TPM in one of the atomic hash/HMAC commands (TPM2_Hash(), or TPM2_HMAC()) then a sequence of commands may be used to provide incremental updates to the digest.

A sequence is started with either TPM2_HashSequenceStart() or TPM2_HMAC_Start(); increments of data are added to the sequence digest(s) using TPM2_SequenceUpdate(); and

TPM2_SequenceComplete() or TPM2_EventSequenceComplete() is used to complete a sequence. TPM2_SequenceComplete() and TPM2_EventSequenceComplete() may also provide the last data to be included in the sequence digest(s).

Three types of sequences are defined:

1) hash
2) Event
3) HMAC

32.4.2 Hash Sequence

In a hash sequence, the TPM will perform a hash over all the data in the sequence using the selected algorithm.

TPM2_SequenceComplete() completes the hash sequence and returns a digest of the data. Additionally, if the data used to create the digest did not start with TPM_GENERATED_VALUE, then a ticket is produced indicating that the digest may be signed with a restricted key.

A hash sequence is:

a) TPM2_HashSequenceStart() (*hashAlg* is a supported hash algorithm), followed by
b) TPM2_SequenceUpdate() (zero or more), followed by
c) TPM2_SequenceComplete()

32.4.3 Event Sequence

For an Event Sequence, the TPM will potentially create multiple digests over the data (a digest for each PCR bank). TPM2_EventSequenceComplete() is used to complete the sequence and return a list of digests; and, If a PCR handle is provided, each digest is extended into the corresponding PCR bank.

EXAMPLE If a TPM implements both a SHA1 and a SHA256 bank, then the list will contain two digests.

An Event Sequence is:

a) TPM2_HashSequenceStart() (*hashAlg* is TPM_ALG_NULL), followed by
b) TPM2_SequenceUpdate() (zero or more) followed by
c) TPM2_EventSequenceComplete() (will do an Extend if *pcrHandle* is a PCR and not TPM_RH_NULL)

32.4.4 HMAC Sequence

For an HMAC sequence, the TPM will use the indicated key as the HMAC key and perform an HMAC computation over the data of the sequence using the specified hash algorithm.

TPM2_SequenceComplete() completes the HMAC sequence and returns the HMAC value.

NOTE The response for TPM2_SequenceComplete() also has a *validation* parameter. This parameter is used for a hash sequence to indicate if the digest is safe to sign with a restricted key. This parameter is not used for an HMAC sequence so the TPM will set the *validation* parameter to a NUL Ticket

An HMAC sequence is:

 Copyright © TCG 2006-2020

a) TPM2_HMAC_Start() (*hashAlg* is a supported hash algorithm), followed by

b) TPM2_SequenceUpdate() (zero or more) followed by

c) TPM2_SequenceComplete()

32.4.5 Sequence Contexts

Sequences involve hashing of data and the intermediate hash state must be retained by the TPM in a protected location. This intermediate state is kept in a vendor-specific structure that may occupy an object slot on the TPM.

A sequence context is assigned a handle so that it may be saved and restored like any Transient Object. Its properties are not identical to other Objects because the sequence context is updated on each use. In addition, unlike other Objects, the public portion of a sequence is not readable with TPM2_ReadPublic(). A sequence context can be replayed if one has the authorization for the sequence.

If an authorization or audit for a sequence object requires computation of a *cpHash* and an *rpHash*, the Name associated with *sequenceHandle* will be the Empty Buffer.

When TPM2_EventSequenceComplete() or TPM2_SequenceComplete() completes successfully, the sequence context is flushed from the TPM.

A sequence is exempt from dictionary attack protection and authorization failures will not cause the TPM to enter lockout.

32.5 Symmetric Encryption

TPM2_EncryptDecrypt() is defined for symmetric encryption and decryption of blocks of data. Support for this command in a TPM may cause the TPM to be subject to different jurisdictions' legal import/export controls than would apply to a TPM without these commands.

The command supports chaining of encryption so that the encryption/decryption may be done incrementally as the data arrives or to handle the cases where the block of data is larger than will fit into a single TPM buffer.

32.6 Asymmetric Encryption and Signature Operations

The annexes to this TPM 2.0 Part 1 contain descriptions of the cryptographic encryption/decryption and signature primitives that are defined for each of the asymmetric algorithms supported by the specification.

 Copyright © TCG 2006-2020

33 Locality

In some systems, accesses to the TPM are segregated by privilege level. The interface to the TPM may be able to discriminate the different privilege levels and provide an indication to the TPM when the access is at a privilege level other than the default level.

The indication of privilege level can be used in access control policy to ensure that the operation on an object is occurring at the right level. The privilege level of a command is called its Locality.

The method by which the TPM interface determines the Locality of an access is system-dependent. The TPM interface provides a Locality indication to the TPM each time the TPM is accessed. The contents of the command or response buffer are not changed by the Locality indication.

The definition of the modifier is platform-specific. Depending on the platform, the modifier could be a special bus cycle or additional input pins on the TPM. One example would be special cycles on the Low Pin Count (LPC) bus that inform the TPM it is under the control of a process on the PC platform. The assumption is that spoofing the modifier to the TPM requires more than just a simple hardware attack and would require expertise and possibly special hardware.

The locality value is represented as a byte and locality values have two separate representations. Localities 0 through 4 are represented as bits in the byte with $0000\ 0001_2$ representing locality 0 and $0001\ 0000_2$ representing locality 4. This representation allows multiple localities to be represented in a single byte as long as the localities are in the range of 0-4. This representation of locality is compatible with previous versions of this specification.

A second representation is for localities above 4. These are called *extended localities*. For extended localities, the locality byte is an integer value representing the locality. Because of the format for localities 0-4, the first extended locality is 32. The range of extended localities is 32-255. The locality value may indicate only one extended locality at a time.

NOTE Locality 5 through 31 cannot be selected.

 Copyright © TCG 2006-2020

34 Hardware Core Root of Trust Measurement (H-CRTM) Event Sequence

34.1 Introduction

A process that puts the system in a known state running known code creates the starting point for a chain of trust. A computer system reset puts the processor and chipset into a known state, and the processor (the root of trust for measurement) begins executing code provided by the platform manufacturer. This initial code is the core root of trust for measurement (CRTM). It is code that must be trusted as there is no way to tell what that code is other than to rely on the manufacturer. Usually, one of the actions of the CRTM is to extend a PCR with a value that represents the identity of the CRTM. This boot process starts the chain of trust with two different roots that are usually from different sources: the RTM from a CPU vendor and a CRTM from a platform manufacture.

Some system implementations support an alternative method of starting a chain of trust that makes the CPU the CRTM. For this method, the CPU is placed in a known state and measures the code that it will run. Before being measured, this code is protected so that it cannot be tampered with and there is assurance that the code that is measured is the code that is executed. Since the CPU is both executing the measured code and measuring it, it is both the RTM and the CRTM. This is called a hardware-based core root of trust for measurement or H-CRTM.

The TPM supports an H-CRTM by providing special interface indications that allow the TPM to determine when it is receiving data from the RTM acting as CRTM. These indications are:

- **_TPM_Hash_Start** – sent by the RTM to indicate the start of a H-CRTM Event Sequence. The TPM will initialize an H-CRTM Event Sequence context. The H-CRTM Event Sequence context contains hash state for each bank of PCR. This indication is only allowed from the RTM when it has been put into a known "good" state as defined by the RTM manufacturer. There is only one _TPM_Hash_Start per H-CRTM Event Sequence.
- **_TPM_Hash_Data** – sent by the RTM to update the digests in the H-CRTM Event Sequence contexts with H-CRTM data. An H-CRTM Event Sequence may have zero or more _TPM_Hash_Data indications.
- **_TPM_Hash_End** – sent by the RTM to indicate the end of the H-CRTM Event Sequence. On receipt of this indication, the TPM will take actions that are dependent on whether the H-CRTM occurred before or after TPM2_Startup(). The actions taken as the result of this indication will always include initialization of at least one PCR followed by a PCR being extended with the H-CRTM data.

During an H-CRTM sequence, if any indication other the _TPM_Hash_Data occurs between the _TPM_Hash_Start and _TPM_Hash_End indications (including receipt of a command), then the H-CRTM Event Sequence is abandoned, the H-CRTM Event Sequence context is flushed, and no change to any PCR occurs.

NOTE The interface may be designed such that it is not possible to interrupt this sequence.

34.2 Dynamic Root of Trust Measurement

When an H-CRTM occurs after TPM2_Startup() it is called the dynamic root of trust for measurement (D-RTM).

NOTE There is no special designation for when the H-CRTM occurs before TPM2_Startup()

NOTE The D-RTM sequence may be repeated one or more times after TPM2_Startup. On each invocation of the D-RTM sequence, the RTM must be in the same known state.

For D-RTM, the TPM will initialize one or more PCR to zero and then extend PCR[17] in each bank with the H-CRTM data accumulated in the H-CRTM Event Sequence.

 Copyright © TCG 2006-2020

$$\text{PCR}[17][hashAlg] \coloneqq \mathbf{H}_{hashAlg}(0\ldots0 \,||\, \mathbf{H}_{hashAlg}(hash_data)) \tag{53}$$

Where

$hash_data$	all the octets of data received in _TPM_Hash_Data indications

The PCR that are initialized and extended as a result of a D-RTM event are specified in a platform-specific TPM specification.

34.3 H-CRTM before TPM2_Startup() and TPM2_Startup() without H-CRTM

If the H-CRTM sequence occurs before TPM2_Startup(), then only PCR[0] will be affected. When _TPM_Hash_End is received, the TPM will complete the Event Sequence digests. It will then initialize PCR[0] to 4 and Extend the H-CRTM Event Sequence data. The value 0…4 represents evidence that the initial measurement was from an H-CRTM.

$$\text{PCR}[0][hashAlg] \coloneqq \mathbf{H}_{hashAlg}(0\ldots04 \,||\, \mathbf{H}_{hashAlg}(hash_data)) \tag{54}$$

where

0…04	denotes a numeric value of 4 with high-order bits of 0 to make the value the size of a digest computed with *hashAlg*
$hash_data$	all the octets of data received in _TPM_Hash_Data indications

If PCR[0] is initialized by an H-CRTM event before TPM2_Startup(), then TPM2_Startup(CLEAR) will not change the value of PCR[0]. Otherwise, TPM2_Startup(CLEAR) will set PCR[0] to the locality of the TPM2_Startup() command.

If there is an H-CRTM event before a TPM2_Startup(CLEAR), there must be an H-CRTM event before a subsequent TPM2_Startup(STATE). The locality of the TPM2_Startup(STATE) is not checked against the locality of the previous TPM2_Startup(CLEAR)

If there is no H-CRTM event before TPM2_Startup(CLEAR), there must be no H-CRTM event before a subsequent TPM2_Startup(STATE) and the TPM2_Startup(STATE) must have the same locality as the previous TPM2_Startup(CLEAR).

35 Command Audit

The command audit mechanism allows the TPM owner to create a verifiable log of each execution of selected commands.

TPM2_SetCommandCodeAuditStatus() is used either to change the list of commands being audited or to change the audit hash algorithm (it cannot change both in the same command). This command requires either Platform Authorization or Owner Authorization. The selection may change at any time.

NOTE 1 It is anticipated that a small number of commands will be selected for audit, most likely those commands that provide identities and control of the TPM. However, there are few restrictions on which commands may be audited.

The audit log, the list of executed TPM commands and responses, is maintained outside the TPM by an untrusted party. Enabling the audit function of a TPM does not guarantee that the log will be properly maintained. The TPM audit function simply provides a means to determine if the log was properly maintained.

It is not necessary to continuously maintain the audit log in order to use the audit capability. When an audit log is started, the current contents of the audit digest register can be read to establish the starting value for the log. At the end of the audit interval, the audit digest register can be read again and the contents of the audit log over the audit interval can be verified.

An audit can be used to track use of keys and, therefore, is potentially privacy sensitive. For this reason, the privacy administrator of the TPM must authorize access to the audit digest register. Authorization from the privacy administrator is expressed using Endorsement Authorization.

The update of the audit digest register occurs when the command completes successfully and the response has been created. The command audit update is:

$$audit_{new} \coloneqq \mathbf{H}_{auditAlg}(audit_{old} \,||\, cpHash \,||\, rpHash) \tag{55}$$

where

$\mathbf{H}_{auditAlg}$	hash function using the currently selected audit hash algorithm
$audit_{old}$	the previously computed audit digest
$cpHash$	the command parameter hash using the audit hash
$rpHash$	the response parameter hash using the audit hash

NOTE 2 Clause 18.7 describes the process for computing *cpHash* and clause 0 describes the process for computing the *rpHash*.

The audit mechanism uses two components: an audit digest register and an audit counter. The audit counter is a non-volatile register that counts the number of audit logs that are created. If the audit digest register contains all octets of zero when an audit event is recorded, then a new audit log is being created and the audit counter is incremented.

An audit log ends and the audit digest is cleared when the command TPM2_GetCommandAuditDigest() returns a signature.

NOTE 3 The audit counter is incremented when the new log starts so that a missing log cannot be dismissed as being irrelevant. Because a new audit log is started only when an auditable event occurs, any missing log is suspect.

 Copyright © TCG 2006-2020

The audit counter is non-volatile and is reset to zero by TPM2_Clear(). The audit digest register is reset when an unanticipated power event occurs (that is, loss of TPM power without an orderly shutdown). The audit digest is preserved over any orderly shutdown.

The audit digest register is reset by a TPM2_SetCommandCodeAuditStatus() that changes the audit digest algorithm *auditAlg*.

An audit report structure contains the current value of the audit digest register and the value of the audit counter.

NOTE 4 The signed audit structure is a TPM2B_ATTEST structure that contains other qualifying information about the signing environment.

Because the audit mechanism utilizes NV memory, endurance may be a factor. The endurance requirements of the audit mechanism are platform-specific.

NOTE 5 The command audit session counter is incremented on the first auditable command in a session. This should be infrequent so the endurance of the counter is not likely to be a major issue.

When the TPM is in Failure mode, command audit is not functional and command audit of TPM2_GetTestResult() and TPM2_GetCapability() will not occur.

TPM2_SetCommandAuditStatus() is audited when it changes the list of audited commands. It is not possible to disable audit of this command. If TPM2_SetCommandAuditStatus() is used to change the audit hash algorithm, then the command is not audited and evidence of this operation is provided by the change in the hash algorithm reported when the command audit value is read.

 Copyright © TCG 2006-2020

36 Timing Components

36.1 Introduction

The TPM has timing components for use in time-stamping of attestations and for gating policy

Time is a free-running hardware value that is not under software control. *Time* advances when the *Time* circuit is powered and is reset to zero when power to the *Time* hardware is lost.

NOTE 1 Typically, the *Time* hardware will be powered down when the rest of the TPM is powered down.

Clock is a value that is derived from *Time* and advances as *Time* advances. *Clock* may be advanced in order to bring it into alignment with real time. However, *Clock* may not be set back except by installing a new owner.

The *resetCount* and *restartCount* values allow detection of power loss that could cause discontinuities in the time recorded by *Clock*. The Safe flag indicates whether *Clock* might have been wound backwards, in which case the current *Clock* value would be unsafe. The timing components are exposed through commands that:

- read the value of *Clock*, *Time*, *resetCount*, and *restartCount* (TPM2_GetTime());
- time-stamp externally provided data using a signature key and *Clock*, *resetCount*, and *restartCount* (TPM2_GetTime(), TPM2_Quote(), TPM2_Certify(), and other restricted signing operations);

NOTE 2 TPM2_ReadClock() returns uncertified (not signed) values. TPM2_GetTime() returns a structure and an optional signature over the data. TPM2_ReadClock() is used by the OS to manage the timing resources of the TPM and TPM2_GetTime() is for attestation of time and is under control of the privacy administrator.

- allow *Clock* to be adjusted forward (TPM2_SetClock());
- allow the rate of advance of *Clock* to be adjusted (TPM2_ClockRateAdjust()); and
- allow objects to be lifetime-limited using authorization policy expressions that reference *Safe*, *Clock*, *Time*, *resetCount*, and *restartCount* (TPM2_PolicyCounterTimer()).

Potential use cases for the TPM timing components include:

- lifetime limits for keys when certificate revocation is impossible or undesirable;
- time-limited delegation of rights (such as, the right to use or duplicate a key for 1 hour);
- time-stamping of security event logs to ensure that events cannot be forged in the past;
- boot-counter stamping of event logs to ensure that a log associated with a particular reboot cannot be deleted without leaving a trace;
- boot-counter/PCR-counter stamping of keys to indicate they were created during OS installation;
- time-stamping of attestation values as an alternative to the use of a nonce in online protocols; and
- indication of whether a TPM/platform has rebooted since last checked.

Clock is *not* designed to be a replacement for other online or local time sources and is not appropriate for all uses. Later clauses describe the behavior of timing resources and their specific security properties. Implementers and relying parties should understand the limitations before using these features.

 Copyright © TCG 2006-2020

36.2 Time

Time is a 64-bit value that contains the time in milliseconds that the circuit providing *Time* has been powered.

NOTE Depending on the frequency of the TPM oscillator and the setting of the frequency divisor (TPM2_ClockRateAdjust()), the rate at which *Time* advances may be in error by as much as 32.5%.

Time is unaffected by TPM2_ClockSet().

The circuit providing *Time* may be powered independently from the rest of the TPM. However, *Time* must be powered whenever the TPM is powered. The *Time* hardware needs to provide a reliable indication that it has lost power or has been reset. *Time* should not be reset unless the TPM requires a _TPM_Init indication before resuming operation.

Time need not advance continuously when powered. The *Time* hardware is required to provide a reliable indication if *Time* has stopped advancing.

36.3 Clock

36.3.1 Introduction

Clock is a time value that can be advanced but never rolled back. It may increment in volatile memory. If so, it is periodically written to NV memory.

A non-orderly shutdown may cause a write to NV memory to be missed. Other values that are written to NV on an orderly shutdown will be advanced to a known safe value on the next startup. However, Clock is not advanced because power outages would cause the clock to be advanced to a time in the future and it could not be adjusted back to an accurate value. To indicate that a value reported in Clock may be a repeat of a previously reported value, a flag (*safe*) is CLEAR after a non-orderly shutdown. After the next NV update of *Clock,* safe is SET to indicate that *Clock* is not a repeat.

Clock is a volatile value that advances at the rate that *Time* advances. A non-volatile value (*NV Clock*) is updated periodically from *Clock*. *NV Clock* will always move forward as *Clock* advances. However, because of unexpected power loss, it is possible that the same value of *Clock* will be reported more than once. The mitigations for this are described in subsequent parts of this clause (36.3).

The accuracy of *Clock* is approximate. The causes of inaccuracy are

- the TPM's time reference may not be accurate, and
- the TPM must rely on external software to provide initial or periodic adjustments to *Clock* settings.

The interpretation of the time-origin (t=0) is out of the scope of this specification, although Coordinated Universal Time (UTC) is expected to be a common convention.

The value of *Clock* may be set forward by external software (TPM2_ClockSet()) to compensate for power interruptions or clock slew, but, except for changes in ownership (TPM2_Clear()), the TPM will not allow external software to set *Clock* backward.

The value of *Clock* may be advanced by TPM2_ClockSet() using either platform or owner authorization.

NOTE The value of *Clock* may not be advanced beyond FF FF 00 00 00 00 00 00_{16}. This restriction prevents any possibility of *Clock* rolling over during its lifetime and simplifies use of *Clock* in policies.

 Copyright © TCG 2006-2020

The TPM may be driven by an imprecise internal or external frequency source. To compensate, the TPM allows external software with a more reliable time source to make limited (+/-15%) adjustments to the rate of advancement of *Clock*.

36.3.2 *Clock* Implementation

The technology used for non-volatile storage may make the update rate for *NV Clock* an endurance issue. To mitigate this, the interval between updates of *NV Clock* from *Clock* are allowed to be as long as once per 2^{22} milliseconds.

NOTE If *NV Clock* is implemented in a technology that allows millisecond updates and has no endurance issues, then *Clock* and *NV Clock* may be the same.

Since *NV Clock* may be updated at a low rate, a power event may cause the value in *Clock* to appear to go backward. For example, assume that the update interval for *NV Clock* is the maximum allowed value (2^{22} milliseconds or approximately 70 minutes). Power may be removed from the TPM and *Time* just before an update of *NV Clock*. Then, when power is restored, *Clock* will be restored from *NV Clock* and *Clock* may have a value that is more than an hour older than the last reported value of *Clock*. This illustrates that the values of *Clock* reported by the TPM for the first hour of operation may have a lower value than values returned before the power outage.

The *Safe* flag in the TPMS_TIME_INFO structure is used to indicate if the reported value of *Clock* is guaranteed not to be a repeat of a previously reported value. The *Safe* flag is described in more detail in the following clause.

36.3.3 Orderly Shutdown of *Clock*

In order to reduce the amount of time that must pass before *Safe* is SET, the TPM supports an orderly shutdown. TPM2_Shutdown() is used to indicate to the TPM that software anticipates the loss of TPM power and that the appropriate state should be preserved. When the TPM receives TPM2_Shutdown(), it will copy all of the bits of *Clock* to *NV Clock.* After an orderly shutdown, the TPM will SET a non-volatile flag to indicate that an orderly shutdown has occurred.

NOTE 1 To allow the *NV Clock* to only have to record the upper bits of *Clock,* an alternate implementation is to keep *Clock* in memory that has a copy saved on an orderly shutdown and to restore *Clock* from that memory on the next power up.

After an orderly shutdown, *Clock* continues to count and *NV Clock* will be updated at the normal rate.

Any time a command is executed that uses the value of *Clock*, the flag indicating orderly shutdown will be CLEAR even if this command occurs subsequent to TPM2_Shutdown(). This flag may be SET when *NV Clock* is updated from *Clock*.

NOTE 2 It is possible for the TPM to perform multiple shutdowns before TPM power is actually lost.

If *Safe* is not SET when TPM2_Shutdown() is received, then *NV Clock* must not be set from *Clock* and *Safe* must not be SET on the subsequent startup.

It is permitted for the low-order 10 bits of *Clock* to come from *Time* and for *NV Clock* not to implement those bits. That is, *NV Clock* does not maintain resolution to better than 2^{10} milliseconds. If an implementation uses this option, then *Safe* will be CLEAR at least for the first 2^{10} milliseconds of TPM operation.

Clock remains safe as long as *Time* is powered. That is, if there is a non-Orderly shutdown and the TPM is powered down but *Time* is powered, then *Clock* will be updated the next time the TPM starts. Since time is not lost, *Clock* will not appear to go backwards and *Safe* can be SET. During the time that the

 Copyright © TCG 2006-2020

TPM is powered down it is not necessary for *Time* to advance, it simply needs not to be set to a lower time value.

36.3.4 *Clock* Initialization at TPM2_Startup()

On any TPM2_Startup() or _TPM_Init (vendor's choice), *Clock* is loaded from *NV Clock* and *Clock* begins incrementing at a one millisecond rate. *NV Clock* is then updated, no less frequently than the update interval. It is anticipated that the first update of *NV Clock* will occur when some number of low-order bits of the volatile *Clock* become zero, indicating the passage of the update interval. For example, assuming that the NV Clock update interval is 2^{12} (approximately every 4 seconds), the TPM may perform an update of *NV Clock* whenever the low-order 12 bits of volatile *Clock* are zero.

NOTE 1 If the TPM had an orderly shutdown, the low-order bits of the *NV Clock* will likely not be zero, so the first update of *NV Clock* after the _TPM_Init will occur in less than the normal update interval.

NOTE 2 If the TPM received TPM2_Shutdown() and a subsequent command that used *Clock*, then the NV value of *Clock* will likely be non-zero, but *Safe* will be CLEAR.

36.3.5 Setting *Clock*

The value in the volatile *Clock* may be set forward using TPM2_ClockSet(). The *newTime* parameter of TPM2_ClockSet() is required to have a greater value than the volatile *Clock*. So that policies that rely on *Clock* do not have to contend with the possibility of the value of *Clock* wrapping, *newTime* may not be greater than FF FF 00 00 00 00 00 00_{16}.

If TPM2_ClockSet() causes the volatile and non-volatile versions of *Clock* to differ by more than the implementation-dependent update interval, then *NV Clock* will be updated before TPM2_ClockSet() returns.

NOTE 1 It is not necessary that all the bits of *NV Clock* be updated. Only the bits of *NV Clock* that are updated in the normal update process need to be changed.

EXAMPLE Assume the update of *NV Clock* occurs every 2^{12} milliseconds (00 00 10 $00_{16)}$, that the low-order 32 bits of *NV Clock* are 00 00 00 00_{16} and *Clock* are 00 00 0F 00_{16}, and that a *newTime* advances *Clock* to 00 00 11 00_{16}. Since this makes the difference between *Clock* and *NV Clock* more than the update interval (2^{12}), *NV Clock* is updated to 00 00 11 00_{16}.

The expected management for *Clock* is for a coarse (large) update to be made after TPM2_Startup() in order to recover the time lost when the TPM was not powered. After that single large change, *Clock* is expected to be updated with relatively small values to keep it synchronized with real time. If software manages *Clock* in this manner, TPM2_ClockSet() will not have to be throttled in order to avoid NV wear-out.

NOTE 2 System software may purposely cause the rate of *Clock* advance to be slower than real time and just make minor adjustments when an attestation of some sort is required. If managed in this way, TPM2_ClockSet() may be executed many times between update intervals. Because update of the NV portion of *Clock* is not allowed unless the difference between the two versions is at least as large as the update interval, TPM2_ClockSet() will not need throttling to avoid wear-out.

NOTE 3 The specification could have been written so that TPM2_ClockSet() would never invoke NV throttling. That is, the value for *newTime* could have been set such that the rate of *NV Clock* update would be at an acceptable rate or TPM2_ClockSet() would fail. This logic is complex, and under normal circumstances, redundant. As a consequence, the specification does not place restrictions on the values of *newTime* other than those listed above. The fact that TPM2_ClockSet() requires Owner Authorization or Platform Authorization should provide some level of protection against an attacker using TPM2_ClockSet() for a wear-out attack on the TPM. TPMs may implement wear-protection if extraordinary rates of update are observed.

36.3.6 *Clock* Periodicity

The TPM clock may be driven by an internal or external frequency source or be derived from a time source supplied by its operating environment. TPM profiles shall specify the time source to be used and the required accuracy.

External software may make limited adjustments to the rate of advance of *Clock* to provide a better approximation to real time.

This specification requires that the nominal rate of advance of *Clock* when powered is within 15% of the rate of UTC. If the external clock is not reliable, the TPM must provide its own clock with the necessary accuracy. External software may indicate that *Clock* is not advancing at the rate of UTC and that the rate needs to be increased or decreased. The command to adjust the clock rate is TPM2_ClockRateAdjust(). The *newRate* parameter of this command allows fine or coarse upward or downward adjustments to the current counting rate. This specification does not define coarse or fine adjustment percentages, and software that manages the TPM must infer this from observed behavior.

The range of adjustment of the rate is dependent on the design of the TPM. It is required that the variation in the rate be large enough that it will allow software to adjust the rate of *Clock* advance to be the same as UTC. The TPM should not allow rate adjustments that are larger than the design tolerance of the TPM.

EXAMPLE 1 A TPM is designed to have a nominal internal oscillator frequency of 10 MHz with a tolerance of +/-15% and a presettable counter that is used to count the oscillator clocks and generate an output every second that is used to advance *Clock*. To cover the tolerance of the oscillator, the preset for the counter would have to be between 8,500,000 and 11,500,000.

EXAMPLE 2 A TPM is designed as above but with the additional ability to accept an outside frequency reference as long as that reference is at least +/-15%. If the external source is more accurate than +/-15%, then the TPM may still allow an adjustment over the 8,500,000 to 11,500,000 range.

NOTE 1 In the worst case, an attacker who knows either the Platform Authorization or Owner Authorization value may be able to make the TPM run 32.5% (1.15^2) fast or slow. However, an attacker who knows the Platform Authorization or Owner Authorization could also set *Clock* arbitrarily far into the future.

An error is returned if external software tries to adjust the clock rate outside specified bounds.

The TPM may store adjustments to the nominal clock rate in volatile memory. If it does, then adjustment should only be stored on an orderly shutdown and not during the actions of TPM2_ClockRateAdjust(). That is, the adjustment value should be in volatile memory and only saved to nonvolatile memory on an orderly shutdown.

NOTE 2 This constraint on TPM2_ClockRateAdjust() is so that software may make changes to the rate at arbitrarily high rates without causing an NV event that might require throttling.

36.4 resetCount

The *resetCount* is a non-volatile, 32-bit counter that is incremented on a successful TPM Reset. It may be read using TPM2_ReadClock() and be used in an authorization policy (TPM2_PolicyCounterTimer()). Additionally, the contents of the *resetCount* are included in the attestation data for any of the attestation commands.

NOTE 1 Depending on the hierarchy of the signing key, the value of *resetCount* may be obfuscated so that a verifier can tell that the counter has changed but cannot know the absolute value of the counter.

The purposes of *resetCount* are to indicate when the static trust state of the platform may have changed and to indicate a possible discontinuity in *Clock*.

 Copyright © TCG 2006-2020

EXAMPLE Without the *resetCount*, the sequence - (1) attest to trusted values (2) transition to an untrusted state (3) perform a transaction (4) TPM Reset (5) attest to trusted value - would hide the fact that the transaction may have occurred during an untrusted state.

NOTE 2 Since the volatile *Clock* is reloaded from the *NV Clock* on each _TPM_Init, the volatile *Clock* will lose some time in nearly all circumstances.

resetCount is incremented whenever the TPM starts up and all previous state is lost (i.e., on a TPM Reset). *resetCount* is set to zero in TPM2_Clear().

36.5 restartCount

In addition to TPM Reset, other events may cause a discontinuity in TPM-recorded time or in the Root of Trust for Reporting (RTR). A suspend-resume cycle will cause a time discontinuity. _TPM_Hash_Start can cause an RTR discontinuity in the dynamic Root of Trust for Measurement (D-RTM) PCR. The *restartCount* is used to provide an indication of these discontinuities.

The *restartCount* is a non-volatile, 32-bit counter that increments when the TPM executes TPM Resume, TPM Restart, or _TPM_Hash_Start. Since *resetCount* increments on each TPM Reset, the combination of *resetCount* and *restartCount* accounts for the cases when a discontinuity may occur, allowing TPM *Time* to fall behind real time.

NOTE When software sets *Clock* forward, that is a positive time discontinuity under control of software. The negative discontinuities of *Clock* are due to hardware actions that may be outside of the control of software.

The combination of *resetCount* and *restartCount* also accounts for the discontinuities of the RTR. A change in *resetCount* indicates a discontinuity in the static RTR, and a change in *restartCount* indicates a change in the dynamic RTR.

restartCount is reset to 0 on TPM Reset – when *resetCount* is incremented. This does not cause a loss of information about the dynamic RTR because a change to *resetCount* also implies a change to the dynamic RTR.

36.6 Note on the Accuracy and Reliability of *Clock*

Clock is designed to allow a managed environment, such as enterprise, to maintain a small deviation between *Clock* and real time. If the platform is not managed, if the platform falls into the hands of an adversary, or if the platform is controlled by malware, then accuracy of *Clock* is diminished. This note addresses considerations that influence the applicability of *Clock* for time-stamping and for time-limited objects.

This analysis assumes that the TPM is not physically attacked, but that adversaries may manipulate external software and local clocks like the CMOS clock on PC platforms.

It is assumed that, under normal operation, external software adjusts *Clock* at platform startup and subsequently makes occasional additional rate and forwarding adjustments to ensure that *Clock* remains within acceptable tolerances. Enterprise management servers or web services may occasionally request time-stamped nonces to check that *Clock* meets network policy.

If *Clock* is used to time-stamp event log entries, then server software should ensure that *Clock* is accurate (as described above), and client software may occasionally record TPM *Time* values counter-signed by external authoritative time-stamping services to provide fiduciary time markers. These services may include the *Clock* and *Time* values as well as the initialization counters (*resetCount* and *restartCount*). The minimal security guarantees provided by the TPM in this case are

 Copyright © TCG 2006-2020

- proper ordering of events logged at times greater than 1 millisecond apart (apart from when associated with discontinuities in the *resetCount* and *restartCount*), and
- that time stamps can never be forged to indicate a time in the past. If the value of *Clock* could be "stale," *Safe* will indicate as much. If *Clock* has occasionally been reported to other authorities or has been counter-signed, then the accuracy of the other time stamps can be interpolated more accurately.

If *Clock* is used to lifetime-limit objects, then when the platform is properly managed, objects will become inaccessible with temporal accuracy related to the precision of clock management and the update interval of *NV Clock*. If the lifetime has the granularity of *NV Clock* update, then once it becomes inaccessible, it cannot be recovered because, at that granularity, *Clock* will not move backward. If the granularity of the lifetime needs to be shorter than the update interval of *Clock*, then the *Safe* flag can be checked to see if the value of *Clock* may be "stale" or not.

If the platform falls into adversarial hands, the attacker will never be able to recover already revoked objects. However, for objects with lifetimes in the future, an adversary may effectively stop the passage of time so that objects never expire.

EXAMPLE To make TPM *Time* "stop," the platform should only be turned on briefly to access the time-limited object and then turned off in a way that prevents an orderly shutdown of the TPM. If the TPM is left on for less than the update interval and the platform does not have an orderly shutdown, *Clock* will continue to repeat values within the range of an update interval. In a managed environment, a platform with a *Clock* that has a value that is substantially different from real time will likely be denied further network services. For a system in an unmanaged environment, a more complex policy using *resetCount* and *Time* may be used to limit access to objects even if time does not advance (for example, the policy may allow access for 20 minutes or 2 reboots).

When the owner of the platform changes (new SPS generated) *Clock* is reset to zero. Using *Clock* to do time stamping with a non-duplicable key does not constitute a vulnerability because the signing key also becomes inaccessible when the owner changes, so no new events can be created. If the time-stamping key is duplicable, then a more detailed security analysis is needed — for instance, examination of the Qualified Name in the signing structure.

If *Clock* is used in other policy settings, similar considerations apply. If an object is destroyed when the owner is changed, then *Clock* reset is benign. However, if an object survives an owner change (such as, an NV Index created by the platform), then use of *Clock* in its access policy may not be appropriate.

36.7 Privacy Aspects of Clock

The attestation structures return several values that, when taken together, may be sufficiently unique to identity a specific platform. For example, the difference between *Clock* and *Time* is, during the interval of a boot, likely to be somewhat unique for a platform. When combined with *resetCount* and *restartCount*, the values can become very indicative of a specific platform. If these values allow signatures from two keys to be correlated, then those keys remain correlated as long as they are in use. The TPM uses authorizations and obfuscation values to prevent this type of unwanted correlation.

All attestations contain a TPMS_CLOCK_INFO structure. That structure contains *Clock*, *resetCount*, *restartCount*, and *Safe*. The attestation structure also contains a 64-bit value that is indicative of the firmware version number. When these values are going to be signed by a key that is not in the Platform or Endorsement hierarchy, *resetCount*, *restartCount*, and firmware version number have a key-specific value added to them before they are put into the attestation structure. The addition allows the determination of change in values but prevents disclosure of the exact value.

Each Attestation Key has a different 128-bit obfuscation value that is constant for the lifetime of the key. It is computed by:

 Copyright © TCG 2006-2020

$$obfuscation \coloneqq \mathbf{KDFa}\,(signHandle{\rightarrow}nameAlg, shProof, \text{“OBFUSCATE”}, signHandle{\rightarrow}QN, 0, 128) \tag{56}$$

37 NV Memory

37.1 Introduction

Each TPM is required to have some non-volatile memory. This memory is used to retain values across power events. The NV memory is used to hold:

- NV Index values,
- objects made persistent by TPM2_EvictControl,
- state saved by TPM2_Shutdown(), and
- Persistent NV data.

37.2 NV Indices

37.2.1 Definition

An NV Index is space that is defined by a user of the TPM. The Index is identified by a unique handle value. An NV Index handle has an MSO of TPM_HT_NV_INDEX.

The NV Index structure has:

- An identifying handle – this handle is assigned by the caller when the Index is defined and is used to reference the Index. The handle associated with an Index has an MSO of TPM_HT_NV_INDEX.
- A *nameAlg* – this parameter indicates the hash algorithm used in the computation of the Name of the Index (see clause 16).
- An authorization policy – this parameter is optional and is the digest of the policy for the NV Index. For the policy to apply to an operation, the corresponding TPMA_NV_POLICY_READ, TPMA_NV_POLICY_WRITE, or TPMA_NV_POLICY_DELETE attribute needs to be SET. Different policies for read, write, and delete can be achieved using policy OR terms and TPM2_PolicyCommandCode().
- A set of NV Index attributes – this parameter determines the nature of the Index and who may manipulate or read the Index.
- An authorization value that is no larger than the size of the digest produced by the *nameAlg* of the NV Index.
- A value indicating the size of the Index data – this parameter indicates the number of octets that are required to hold the NV data. For some Index types, the size is fixed.
- The NV Index data that may be modified according to the type of the NV Index.

All the parts of the NV Index structure, except for the *authValue* and Index data, constitute the public portion of the Index. They are hashed using the *nameAlg* to produce the Name of the Index.

The public area of the Index may be read using TPM2_NV_ReadPublic().

NOTE TPM2_NV_ReadPublic() also returns the Name of the NV Index.

An NV Index can be designated as a hybrid Index. A hybrid Index is intended for applications where frequent updates are expected. High frequency updates are generally not compatible with the technology currently used for nonvolatile storage on a TPM. A hybrid Index maintains a volatile (RAM) and a non-volatile copy of its Index data. A write to an ordinary Index is immediately written to NV memory but a

 Copyright © TCG 2006-2020

write to a hybrid Index may only update the copy of the Index data in RAM. The non-volatile copy of a hybrid NV Index is updated on TPM2_Shutdown().

If an NV Index has TPMA_NV_ORDERY SET, then it is a hybrid Index.

NOTE 1 The user of a hybrid NV Index must understand that data may be lost if the TPM does not shut down in an orderly fashion so that the volatile data can be written to NV memory.

Whether or not NV Index is a hybrid, when an NV Index is defined (TPM2_NV_DefineSpace()), the persistent values of the NV Index are written to NV if the command completes successfully.

Any NV Index type can be defined as a hybrid. The conditions under which the write to NV memory occur vary and are described below.

NOTE 2 An implementation is not required to support an arbitrary number of hybrid indices and is not required to support any ordinary hybrid Index with a size of more than eight octets.

37.2.2 NV Index Allocation

An NV Index is allocated with TPM2_NV_DefineSpace(). Either Platform Authorization or Owner Authorization is required in order to allocate an Index. The caller indicates the NV Index to assign to the NV location, the access controls for the Index, and the type and or size of the data buffer that should be reserved for writing. While the allocation process does write the meta-data for the Index to NV, it does not write to the data area of the Index data and a read of the NV location before it is written will return an error (TPM_RC_NV_UNINITIALIZED).

When an NV Index is defined (TPM2_NV_DefineSpace), its TPMA_NV_WRITTEN attribute will be CLEAR. Until the Index is written by a party that can satisfy the write policy, the Index is defined but has no data, and TPM2_PolicyNV() and TPM2_NV_Read() will fail.

TPMA_NV_WRITTEN is SET when an authorized party first writes the Index. This permits a relying party to know that the value in the Index was written by an authorized party. It is not simply a default value that was present when the Index was defined (or deleted and redefined to attempt a roll back.)

A replying party can read the Index attributes and policy, which are public, to determine the authorized party.

NOTE The meta-data of an NV Index is the data relating to the NV Index description (Index number, policy, attributes, data size, and *authValue*) along with any additional information that the TPM needs to manage the NV Index memory.

Different types of NV Index may be supported.

- **Ordinary** – an Index with an NV Index type of TPM_NT_ORDINARY contains data that is opaque to the TPM that is modified using TPM2_NV_Write().
- **Counter** – an Index with an NV Index type of TPM_NT_COUNTER contains a 64-bit counter that is modified using TPM2_NV_Increment().
- **Bit field** – an Index with an NV Index type of TPM_NT_BITS contains 64 bits that are initialized to 0 and are modified using TPM2_NV_SetBits().
- **Extend** – an Index with an NV Index type of TPM_NT_EXTEND contains a value that has behavior similar to a PCR and is modified using TPM2_NV_Extend().
- **PIN Fail** - an Index with an NV Index type of TPM_NT_PIN_FAIL that contains a TPMS_NV_PIN_COUNTER_PARAMETERS structure that is modified using TPM2_NV_Write() or by using the *authValue* of the Index. *pinCount* is reset when an authorization attempt using *authValue* succeeds. *pinCount* is incremented after an authorization attempt using *authValue* fails. *pinCount*

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

cannot increment beyond *pinLimit* because *authValue* authorization is locked out if *pinCount* >= *pinLimit*. A Pin Fail Index can be modified with TPM2_NV_Write.

- **PIN Pass** - an Index with an NV Index type of TPM_NT_PIN_PASS that contains a TPMS_NV_PIN_COUNTER_PARAMETERS structure that is modified using TPM2_NV_Write() or by using the *authValue* of the Index. *pinCount* is incremented after an authorization attempt using *authValue* succeeds. *pinCount* cannot increment beyond *pinLimit* because *authValue* authorization is locked out if *pinCount* >= *pinLimit*. A Pin Pass Index can be modified with TPM2_NV_Write.

TPM2_NV_DefineSpace() can fail if an Index with the requested handle already exists or if there is insufficient NV memory for the allocation. Creation of a hybrid Index will fail if there is insufficient RAM available for the allocation. The command will fail if an Index type is not supported.

EXAMPLE If the TPM does not implement TPM2_NV_Extend(), then the TPM will not allow creation of an NV Index that has the TPMA_NV_EXTEND attribute.

If the Index to be created has its TPMA_NV_POLICY_DELETE attribute SET, then platform authorization is required for allocation. This attribute is only allowed to be selected if TPM2_NV_UndefineSpaceSpecial() is implemented on the TPM.

NOTE This attribute indicates that a policy is required to delete the Index. It permits creation of an Index that can never be deleted, for example, by specifying an Empty Policy. Requiring platform authorization protects against the current TPM owner creating such an Index.

37.2.3 NV Index Deletion

An NV Index can be removed using either TPM2_NV_UndefineSpace() or TPM2_NV_UndefineSpaceSpecial().

If the TPMA_NV_POLICY_DELETE attribute is SET, then the Index can only be deleted if ADMIN role authorization is provided. ADMIN role authorization is provided by a policy session with the *commandCode* of the policy set to TPM2_NV_UndefineSpaceSpecial().

TPM2_NV_UndefineSpace is used to delete other Indices from the NV. The authorization given for deleting the Index is required to be the same as the authorization given to allocate the Index.

TPM2_Clear() will remove any NV Index that used Owner Authorization to define the Index. TPM2_Clear() uses either TPM_RH_LOCKOUT or TPM_RH_PLATFORM.

TPM2_ChangePPS() does not cause any NV Index to be removed.

NOTE To comply with FIPS-140, the data contents and authorization value must be zeroized when the NV Index is deleted.

37.2.4 High-Endurance (Hybrid) Indices

37.2.4.1 Description

Some applications need the ability to make frequent updates to non-volatile values such as monotonic counters. A high update rate is generally not compatible with the technology currently used for non-volatile storage on a TPM. To allow the TPM to support high-update rates while protecting the endurance of the NV memory, a hybrid Index type is defined.

When an NV Index is defined with the TPMA_NV_ORDERLY attribute SET, the TPM will allocate the required NV memory as well as space in TPM RAM for the data value. During normal operation, updates to the Index will modify the RAM copy of the Index data with updates to the NV on Shutdown() or

whenever the RAM copy of a counter is divisible by a set modulus. In some cases, the data write may never occur.

NOTE The value of the modulus is implementation specific and can be accessed using TPM2_GetCapability(*capability* == TPM_CAP_TPM_PROPERTY, *property* == TPM_PT_ORDERLY_COUNT). The returned value is the modulus – 1. This value is referred to as MAX_ORDERLY_COUNT.

If the TPMA_NV_ORDERLY attribute of an Index is SET, the TPM will perform special processing on the Index at TPM2_Startup(). The processing is dependent on the type of the Index.

37.2.4.2 Hybrid Indices Other than Counter Indices

For hybrid Indices that are not Counters, the NV Index data in volatile memory is copied to non-volatile memory on a Shutdown(STATE), The data need not be copied to non-volatile memory on Shutdown(CLEAR).

a) On TPM Resume, the non-volatile copy of the Index data is copied into the volatile version of the NV Index data.
b) On TPM Reset, the TPMA_NV_WRITTEN attribute will be initialized to CLEAR. On a subsequent update of the Index, it will be initialized before it is updated.
c) On TPM Restart, if TPMA_NV_CLEAR_STCLEAR is SET, the NV Index is initialized as in b) above. If TPMA_NV_CLEAR_STCLEAR is CLEAR, then the NV Index is initialized as in a) above.

NOTE TPMA_NV_CLEAR_STCLEAR may not be SET if the NV Index type is TPMA_NV_COUNTER. Counters are either restored (on an orderly startup) or set to a higher value (on a non-orderly startup).

37.2.4.3 Counter Hybrid Indices

The hybrid counter Index is designed so that it will be monotonically increasing and not miss an increment command regardless of the type of shutdown or startup.

For a Counter NV Index with the TPMA_NV_ORDERLY attribute, Index data in non-volatile memory is written to NV on any Shutdown().

NOTE 1 For a Counter (or any other Index) that has TPMA_NV_ORDERLY CLEAR, non-volatile memory is written on any update of the NV Index.

On any orderly startup of the TPM (TPM2_Startup() following an orderly shutdown), the NV value of a hybrid counter Index will be copied to the RAM version. The count will be able to continue without any discontinuity.

On a non-orderly startup, the value of the counter in NV is adjusted before it is copied to RAM. A counter is adjusted by logical OR of the value of MAX_ORDERLY_COUNT to the NV value. This sets the RAM version of the counter to the maximum value it could have had before being updated due to the modulus test. This ensures that the RAM counter value is no less than any previously used counter value.

EXAMPLE Assume that MAX_ORDERLY_COUNT contains 0F FF_{16} and that the TPM lost power without an orderly shutdown. On a startup, if an orderly counter is found to have a value of 00 00 00 00 00 01 73 $A1_{16}$, then the RAM version is updated to 00 00 00 00 00 01 7F FF_{16}.

NOTE 2 When the RAM version of the counter is set this way, it is not necessary to immediately update the counter to NV. If the counter is incremented, then it will be automatically saved to NV when the low bits become zero.

 Copyright © TCG 2006-2020

NOTE 3 If the RAM counter were initialized so that the low bits were zero and a subsequent un-orderly shutdown occurred, the counter would have to be advanced again, whether it had been incremented or not. By setting the counter to the maximum value before NV update, there is no need to advance the count on a subsequent unorderly shutdown unless the counter was used.

37.2.5 Reading an NV Index

Read access to an NV Index is provided with TPM2_NV_Read(), TPM2_NV_Certify(), and TPM2_PolicyNV(). For all of these commands, read authorization is required. The attributes of the Index determine what authorizations are allowed. TPMA_NV_PPREAD allows the Index to be read using Platform Authorization*;* TPMA_NV_OWNERREAD allows the Index to be read using Owner Authorization; TPMA_NV_AUTHREAD allows the Index to be read using the *authValue* of the Index; and TPMA_NV_POLICYREAD allows the Index to be read if the *authPolicy* of the Index is satisfied.

At least one of TPMA_NV_PPREAD, TPMA_NV_OWNERREAD, TPMA_NV_AUTHREAD or TPMA_NV_POLICYREAD needs to be SET or the TPM will not allocate the Index.

An access control (TPMA_NV_READ_STCLEAR) allows reading of the Index to be temporarily blocked. When this attribute is SET, TPM2_NV_ReadLock() may be used to temporarily disable read access to the Index. When the Index has been locked for read, the TPMA_NV_READLOCKED attribute of the Index will be SET. TPMA_NV_READLOCKED will be CLEAR on the next TPM Reset or TPM Restart. If the TPMA_NV_READLOCKED attribute is SET when the Index is read, the TPM returns TPM_RC_NV_LOCKED.

The *authPolicy* of the NV Index may be constructed such that it only applies for reading or for writing. It may be constructed to allow general reading and limited writing or general writing and limited reading. If reading or writing of the Index is to be restricted based on PCR values, then read authorization needs to use *authPolicy*.

37.2.6 Updating an Index

37.2.6.1 Introduction

The command used to update an Index is determined by the NV Index type. TPM2_NV_Write() is used to modify an Ordinary Index or a PIN Index, TPM2_NV_Increment() is used to modify a Counter Index, TPM2_NV_SetBits() is used to modify a Bit Field Index, and TPM2_NV_Extend() is used to modify an Extend Index. For all of these commands, write authorization is required.

The attributes of the Index determine what authorizations are allowed. TPMA_NV_PPWRITE allows the Index to be modified using Platform Authorization*;* TPMA_NV_OWNERWRITE allows the Index to be modified using Owner Authorization; TPMA_NV_AUTHWRITE allows the Index to be modified using the *authValue* of the Index; and TPMA_NV_POLICYWRITE allows the Index to be modified if the *authPolicy* of the Index is satisfied.

At least one of TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, TPMA_NV_AUTHWRITE or TPMA_NV_POLICYWRITE needs to be SET or the TPM will not allocate the Index. For a PIN Index, TPMA_NV_AUTHWRITE may not be SET and at least one of the other three write methods is required to be selected.

NOTE 1 A method other than TPMA_NV_AUTHWRITE is required for a PIN Index because the *authValue* of a PIN Index is not accessible until the Index is written.

If the access control attribute TPMA_NV_WRITEDEFINE is SET, TPM2_NV_WriteLock() or TPM2_NV_GlobalWriteLock() may be used to permanently disable modify access to the Index. When the Index has been locked for modify, the TPMA_NV_WRITELOCKED attribute of the Index will be SET. This attribute will remain SET until the Index is deleted (TPM2_NV_UndefineSpace()).

If TPMA_NV_WRITEDEFINE is CLEAR, the TPMA_NV_WRITELOCKED attribute can be SET using TPM2_NV_WriteLock() if TPMA_NV_WRITE_STCLEAR is SET or TPM2_NV_GlobalWriteLock() if TPMA_NV_GLOBALLOCK is SET. In this case, TPMA_NV_WRITELOCKED will be CLEAR on the next TPM Reset or TPM Restart.

NOTE If TPMA_NV_WRITELOCKED is SET, but TPMA_NV_WRITTEN is CLEAR, then TPMA_NV_WRITELOCKED is CLEAR by TPM Reset or TPM Restart. This is true even if the TPMA_NV_WRITEDEFINE attribute is set. It prevents an NV Index from being defined that can never be written and permits a use case where an Index is defined, but the user wants to prohibit writes until after a reboot.

If the TPMA_NV_WRITELOCKED attribute is SET when an attempt is made to modify the Index, the TPM returns TPM_RC_NV_LOCKED.

For a PIN Fail Index, the TPM will return TPM_RC_NC_ATTRIBUTES if TPMA_NV_NO_DA is CLEAR.

37.2.6.2 NV Ordinary Index Update

TPM2_NV_Write() is used to modify the contents of an ordinary Index. The modification may be to the entire Index or, if the Index attributes allow (TPMA_NV_WRITE_ALL CLEAR), the size of the data to write can be as small as zero octets.

When a partial write is allowed, the *offset* parameter of TPM2_NV_Write() may be non-zero or the *size* of the data parameter may be less than the *size* of the Index. The TPM checks the TPMA_NV_WRITTEN attribute. If it is CLEAR, then the TPM will initialize the remainder of the Index to either all zero or all one. Alternatively, the TPM can initialize the entire Index at the time the Index is defined.

If the sum of the size of the *data* parameter and the *offset* parameter in TPM2_NV_Write() is greater than the size of the Index, then the TPM will not perform the write and will return an error.

On any TPM2_NV_Write() (including a size of zero), if the modification is successful, then the TPMA_NV_WRITTEN attribute of the Index will be SET. Any octets not initialized by the first write will have a value of all zero or all one.

EXAMPLE If the Index is defined to contain 2 octets, and the first write of the Index is a single octet of 55_{16}, to offset 0, then the next read of the full Index will return 55 00_{16}.

If the Ordinary Index has the TPMA_NV_ORDERLY attribute, then only the RAM version of the Index is written. The data is preserved on a Shutdown(STATE).

37.2.6.3 NV Counter Index

When an Index has the TPMA_NV_COUNTER attribute, it behaves as a monotonic counter and may only be modified using TPM2_NV_Increment().

When an NV counter is created, it has no value and the TPMA_NV_WRITTEN attribute will be CLEAR.

On each TPM2_NV_Increment() the TPM checks the TPMA_NV_WRITTEN attribute of the Index. If it is CLEAR, then the TPM will initialize the 8-octet counter value such that the first increment will set a value that is greater than any value that a counter Index with the same Name has had over the lifetime of the TPM. The TPMA_NV_WRITTEN attribute will be SET.

NOTE 1 This check ensures that an Index cannot be deleted and another Index with the same Name defined with a lower value.

NOTE 2 The reference implementation implements this by tracking and using the largest count of any deleted NV Counter. An alternative implementation could track the largest count of any NV Counter, deleted or currently defined.

 Copyright © TCG 2006-2020

After checking TPMA_NV_WRITTEN and performing any required initialization operations, the TPM will increment the Counter.

NOTE 3 The TPM will need to maintain a largest-count value. It is not necessary to update this value except when a NV Index is deleted. If the NV Index being deleted has the largest value held by an NV Index, then this value would be copied to the largest-count value. The value of an NV Counter Index after the first increment is larger than the largest-count value.

NOTE 4 Since no counter can ever repeat a previous value ever contained in any NV Counter Index, a counter with a particular Name cannot be rolled back by deleting it and redefining it.

If the TPMA_NV_ORDERLY attribute is CLEAR, the increment will occur on the NV version of the counter (no RAM version exists). If the TPMA_NV_ORDERLY attribute is SET, the increment will occur on the RAM version of the counter, and if this causes a rollover, the NV version of the counter is updated. However, if TPMA_NV_WRITTEN is CLEAR, the NV version of the counter is also written. Once SET, TPMA_NV_WRITTEN of a counter is never CLEAR.

An Index may be defined with the TPMA_NV_ORDERLY attribute to indicate that the Index is expected to be modified at a high frequency and that the data is only required to persist when the TPM goes through an orderly shutdown process. For a counter, it also means that it will be written to NV when the counter has reached some threshold value. The threshold value for counters (MAX_ORDERLY_COUNT) is implementation dependent and can be read using TPM2_GetCapability(capability = TPM_CAP_PT, property = TPM_PT_ORDERLY_COUNT). This property has one of 32 values that can be expressed as (2^N-1) where N is between 1 and 32.

EXAMPLE If MAX_ORDERLY_COUNT is 00 00 0F FF_{16}, then whenever the RAM version of a counter is incrementing, causing the low-order 12 bits to be zero, the NV version of the counter is updated.

The meaning of this threshold value is that when the counter is incremented such that the counter value ANDed with MAX_ORDERLY_COUNT is zero, then the NV version of the counter will be updated.

NOTE 5 Another way to express this is to simply say that the NV version of the counter will be updated when the low order bits of the counter "roll-over".

The TPM is required to ensure that, when an NV Counter is read, its value is not less than a previously reported value of the counter. That is, it may not go backward. If the shutdown was orderly, then, regardless of the type of the NV Counter, the NV value of a counter will not be less than the last reported value. If the shutdown was not orderly and the NV Counter has the TPMA_NV_ORDERLY attribute, then a value of the Counter may have been read from the RAM version of the counter but the NV version may not have been updated. To handle this case, if the TPMA_NV_ORDERLY attribute of an NV Counter is SET, and the TPM shutdown was not orderly, then, at TPM2_Startup() the TPM will OR the value of MAX_ORDERLY_COUNT to the contents of the non-volatile counter and set that as the current count in the RAM version of the counter.

NOTE 6 The TPM must prevent a rollback attack caused by a counter being deleted and then being recreated with a lower value. To do this, the TPM may keep track of the value of the highest count of a deleted counter using a phantom counter. When a counter is deleted, the current value of the counter is compared to the current phantom counter and other counters. If the value is larger than the phantom counter and other counters, the phantom counter is updated. When a new NV counter is created, it starts with the highest value of all the counters, including the phantom counter.

For a Counter with the TPMA_NV_ORDERLY attribute SET, the NV copy of the data will be updated whenever a specified number of low order bits of the RAM copy become all zeros. That number of low order bits is TPM implementation-dependent. The setting for a TPM may be found using TPM2_GetCapability(TPM_CAP_TPM_PROPERTIES, TPM_PT_ORDERLY_COUNT). That capability is MAX_ORDERLY_COUNT.

The TPMA_NV_CLEAR_STCLEAR attribute has no effect on an NV Counter Index and it may be SET or CLEAR in the template.

 Copyright © TCG 2006-2020

37.2.6.4 NV Bit Field Index

When an Index has the TPMA_NV_BITS attribute it may only be modified by TPM2_NV_SetBits().

When an NV Bit Field Index is created, it has no value and the TPMA_NV_WRITTEN attribute will be CLEAR.

On each TPM2_NV_SetBits(), the TPM will check the TPMA_NV_WRITTEN attribute of the Index. If it is CLEAR, the TPM will set the 64 bits of the Index to zero. The TPM will then SET the TPMA_NV_WRITTEN attribute for the Index.

After checking TPMA_NV_WRITTEN and doing any necessary initialization, the TPM will OR the *bits* parameter to the Index.

If the TPMA_NV_ORDERLY attribute is not SET, the NV value of the Index is written with the modified value. If no bits were SET in the *bits*, the NV Index data will only be updated if TPMA_NV_WRITTEN was CLEAR when the command execution was started.

If TPMA_NV_ORDERLY is SET, the RAM version of the Bit Field data is updated but it is not written to NV. The data is only preserved to NV on a Shutdown(STATE), and on TPM Reset, the TPMA_NV_WRITTEN attribute of the Index will be CLEAR.

37.2.6.5 NV Extend Index

When an Index has the TPMA_NV_EXTEND attribute, it may only be modified by TPM2_NV_Extend().

When an NV Extend Index is created, it has no value and the TPMA_NV_WRITTEN attribute will be CLEAR.

On each TPM2_NV_Extend(), the TPM will check the TPMA_NV_WRITTEN attribute of the Index. If it is CLEAR, the TPM will initialize the Index to a Zero Digest that is the size of the digest produced by the *nameAlg* of the Index. The TPM will then SET the TPMA_NV_WRITTEN attribute for the Index.

After checking TPMA_NV_WRITTEN and doing any necessary initialization, the TPM will extend the Index using:

$$nvIndex{\rightarrow}data_{new} \coloneqq \mathbf{H}_{nameAlg}\left(nvIndex{\rightarrow}data_{old} \,||\, data.buffer\right) \tag{57}$$

where

$\mathbf{H}_{nameAlg}$	the hash algorithm indicated in *nvIndex→nameAlg*
$nvIndex{\rightarrow}data$	the value of the data field in the Index
$data.buffer$	the data buffer of the command parameter

If the TPMA_NV_ORDERLY attribute is not SET, the NV value of the Index is written with the modified value.

If TPMA_NV_ORDERLY is SET, the RAM version of the Index is updated but it is not written to NV. The data is only preserved on a Shutdown(STATE), and on TPM Reset, the TPMA_NV_WRITTEN attribute of the Index will be CLEAR..

 Copyright © TCG 2006-2020

37.2.6.6 NV PIN Index

TPM2_NV_Write() is used to modify the contents of a PIN Index. The modification may be to the entire Index or, if the Index attributes allow (TPMA_NV_WRITE_ALL CLEAR), the size of the data to write can be as small as zero octets.

When a partial write is allowed, the *offset* parameter of TPM2_NV_Write() may be non-zero or the *size* of the data parameter may be less than the *size* of the Index. The TPM checks the TPMA_NV_WRITTEN attribute. If it is CLEAR, then the TPM will initialize the remainder of the Index to either all zero or all one. Alternatively, the TPM can initialize the entire Index at the time the Index is defined.

If the sum of the size of the *data* parameter and the *offset* parameter in TPM2_NV_Write() is greater than the size of the Index, then the TPM will not perform the write and will return an error.

On any TPM2_NV_Write() (including a size of zero), if the modification is successful, then the TPMA_NV_WRITTEN attribute of the Index will be SET. Any octets not initialized by the first write will have a value of zero.

EXAMPLE If the Index is defined to contain 2 octets, and the first write of the Index is a single octet of 55_{16}, to offset 0, then the next read of the full Index will return 55 00_{16}.

If the Index has the TPMA_NV_ORDERLY attribute SET, then only the RAM version of the Index is written. The data is only preserved to NV on a Shutdown(STATE), and on TPM Reset, the TPMA_NV_WRITTEN attribute of the Index will be CLEAR.

If the *authValue* of an PIN Index is used for authorization, then the authorization will fail if the *pinCount* field of the Index is not less than the *pinLimit* field or if the TPMA_NV_WRITTEN attribute of the Index is CLEAR.

When the *authValue* of a PIN Index is used for authorization and the authorization succeeds, the *pinCount* field is set to zero if the Index is PIN Fail and incremented if the Index is PIN Pass. If the authorization fails, *pinCount* is incremented for a PIN Fail Index and left unchanged for a PIN Pass Index.

37.2.7 NV Index in a Policy

TPM2_PolicyNV() may be used to include the contents of an NV Index in a policy command. TPM2_PolicyNV() allows various comparisons of the value of the NV data with a reference value.

TPM2_PolicyNV() is an immediate assertion (see 19.7.7.2). If the comparison succeeds, the TPM will update the *policyDigest* with the comparison values and the access controls on the referenced Index, including the *authPolicy*. Inclusion of the update policy of the Index provides a means of identifying the update properties of the Index. To make effective use of this command, writing of the Index should be dependent on *authPolicy.* If the policy must be met in order to write the Index, then it is possible to ensure that only the correct entity may recreate the Index. If other write authorizations are allowed, then it is not possible to know if the Index was written by a known entity.

If an NV Index is used in TPM2_PolicyNV() after it is defined but before it is first written, then the TPM will return an error.

The nominal use of a PIN Index is to reference the Index in an entity's policy in TPM2_PolicySecret(). The TPM2_PolicySecret() will succeed if *pinCount* is less than *pinLimit* and the caller is able to provide the *authValue* of the Index in the authorization. If the rest of the policy is satisfied, access to the PIN-protected entity will be allowed.

NOTE 1 A PIN Fail Index provides a form of individual Dictionary Attack defense that is not affected by the TPM's global Dictionary Attack mechanism. In particular, it can be used to allow the TPM to emulate the behavior of a smart card.

NOTE 2 A PIN Pass Index allows count-limited use of a TPM object. An example use would be to only allow access to a decryption key for protected content.

37.2.8 PIN Index Considerations

37.2.8.1 Restricting the number of uses of an object with PIN Pass

It is possible to limit the number of *authValue* (PIN) authorizations of a particular key or entity.

A key or object has a limited number of authorizations when its policy has a TPM2_PolicySecret assertion pointing to a PIN Pass NV Index.

A PIN Pass's *pinLimit* is the number of correct authorization attempts that are permitted before authorization via *authValue* is locked out. If *pinCount* is less than its *pinLimit*, *pinCount* is incremented immediately by the TPM after *authValue* authorization succeeds. There is no automatic reset or decrement method for *pinCount*. Once *pinCount* equals *pinLimit*, an administrator must reduce *pinCount* and/or increase *pinLimit* using TPM2_NV_Write or delete the Index.

37.2.8.2 Localized Dictionary Attack protection with PIN Fail

It is possible to authorize a particular key or object via an *authValue* (PIN) that has its own individual Dictionary Attack defense and does not use (and is not affected by) the TPM's global Dictionary Attack defense mechanism. This may be useful when a TPM is used to emulate a smartcard, for example.

A key or object has localized Dictionary Attack protection if its policy has a TPM2_PolicySecret assertion pointing to an PIN Fail NV Index.

A PIN Fail's *pinLimit* is the number of incorrect authorization attempts that are permitted before authorization via *authValue* is locked out. If *pinCount* is less than its *pinLimit*, *pinCount* is incremented immediately by the TPM after *authValue* authorization fails. *pinCount* is reset to zero by the TPM whenever *authValue* authorization succeeds.

37.2.8.3 PIN Index Attributes

A PIN Index may be read or write locked. If read or write locked, the Index may still be referenced by TPM2_PolicySecret(). An Index disabled using *phEnableNV* (if platform created) or *shEnable* (if owner created) cannot be used in a policy. If a policy points to an unwritten PIN Pass or PIN Fail Index, the Index's authorization check must fail because *pinLimit* is not written.

NOTE 1 Allowing a PIN Index to be used when write locked allows it to be used as a PIN but prevents writing of the *pinLimit*.

TPMA_NV_ORDERLY may be SET or CLEAR, however, if SET the Index will revert to unwritten on TPM Reset and possibly on TPM Restart (depending on TPMA_NV_CLEAR_STCLEAR). This will cause the Index to not be usable for PIN authorization until it is reinitialized.

TPM2_PolicyAuthValue() and TPM2_PolicyPassword() cannot be used in the policy that does the initial write to a PIN Index. This is because these policy commands require that the *authValue* of the PIN Index to be used and the *authValue* of a PIN Index cannot be used until it is first written. Therefore, it may be desirable that TPMA_NV_POLICYWRITE is SET so that the PIN Index value may be initialized.

If TPMA_NV_POLICYREAD, TPMA_NV_PPREAD, or TPMA_NV_OWNERREAD is SET then the Index may read using TPM2_NV_Read (with those authorizations) without affecting the contents of the Index. If TPMA_NV_AUTHREAD is the only method of reading the Index, then the act of reading the Index could change its *pinCount.*

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

NOTE 2 Using the NV Index authorization value for the read would consume a PIN Pass Index authorization or reset the PIN Fail *pinCount*. In addition, *authValue* can't be used for authorization once *pinCount* >= *pinLimit*.

NOTE 3 In a PIN Fail Index, it may be desirable that TPMA_NV_AUTHREAD is SET, so *pinCount* can be reset by reading the NV Index with valid *authValue* authorization.

TPMA_NV_AUTHREAD is SET, so pinCount can be reset by reading the NV Index with valid authValue authorization.

It is recommended that the Index have a policy that includes a PolicySigned assertion, to unambiguously identify the Index and the entity authorized to initialize the Index.

NOTE 4 This prevents covert attacks where an Index is secretly deleted and replaced.

If the *authObject* parameter of TPM2_PolicySecret() references a PIN Pass Index, then the command may succeed, but a NULL ticket will be returned. The reason is that the ticket could allow more accesses to a count limited object than allowed by the PIN Pass Index.

NOTE 5 Without this restriction, a caller could get a ticket for a count limited object and use the ticket instead of using the PIN Pass Index. This could, potentially, allow unlimited access to a PIN Pass entity.

If a PIN Pass or PIN Fail Index is referenced as a bind object, the TPM must return TPM_RC_HANDLE. Otherwise, the sequence in which the TPM processes authorizations would enable a hammering attack on the Index.

Restrictions on PIN Pass and PIN Fail Indexes are specified in Part 3 TPM2_NV_DefineSpace.

37.3 Owner and Platform Evict Objects

In some applications, it is desirable for an object to be made persistent in the TPM so that it is always available. An example of when this would be useful is for a Primary Key. Having the Primary Key be always available avoids the time penalty of re-computing the Primary Key after each TPM Reset.

TPM2_EvictControl() is used to make a loaded object persistent by saving it to the TPM's NV memory. This command is also used to remove a persistent object.

To be made persistent, an object needs to have both public and private portions loaded; the object cannot be in the NULL hierarchy, the object cannot have the *stClear* attribute SET, and the object cannot be a descendant of a key with the *stClear* attribute SET.

The type of the *objectHandle* parameter of TPM2_EvictControl() determines if the Object is to be made persistent or to be removed from persistent memory. If *objectHandle* is a Transient Object, it is made persistent and, if *objectHandle* is a persistent object, it is deleted. The Transient Object is not affected.

When making a Transient Object persistent, the *persistentHandle* parameter of TPM2_EvictControl() indicates which handle is to be assigned to the persistent version of the object. The TPM will not allow assignment of a persistent handle if that handle is already assigned to a persistent object.

If *objectHandle* is a Transient Object in the Platform Hierarchy, Platform Authorization must be provided. If *objectHandle* is in the Endorsement or Storage Hierarchy, Owner Authorization is required.

The persistent handle space is divided evenly between the Platform and the Owner. The persistent handles that may be assigned when Owner Authorization is provided are in the range $81\ 00\ 00\ 00_{16}$ to $81\ 7F\ FF\ FF_{16}$. Handles in the range $81\ 80\ 00\ 00_{16}$ to $81\ FF\ FF\ FF_{16}$ may be assigned when Platform Authorization is provided. When removing a persistent object, the authorization used to persist the object is required to remove it.

 Copyright © TCG 2006-2020

37.4 State Saved by TPM2_Shutdown()

37.4.1 Background

TPM2_Shutdown() is used for an orderly shutdown of the TPM. When doing an orderly shutdown, the TPM will save some state to NV memory. In the reference implantation, the state saved is separated into three groups:

1) NV Orderly Data – data that is saved on any Shutdown and is not reset,
2) NV Clear Data – data that is saved on Shutdown(STATE) and is reset on TPM Restart or TPM Reset (such as, PCR), and
3) NV Reset Data – data that is saved on Shutdown(STATE) and is reset on TPM Reset (such as session context tracking information).

37.4.2 NV Orderly Data

The data in this structure is saved to NV on any Shutdown type and restored on any Startup. It may have special initialization performed if the Startup is not orderly. In the reference implementation, this data is collected into a special data structure (ORDERLY_DATA) the contents of which are illustrated in Table 31.

Table 31 — Contents of the ORDERLY_DATA Structure

Parameter	Description	Changed By:
clock	This is the version of *Clock* that is updated on any Shutdown and on any rollover of the RAM version of *Clock*.	TPM2_Clear(), TPM2_Startup(), passage of time
clockSafe	used to determine the *Safe* value reported in the TPMS_CLOCK_INFO structure. This value is CLEAR when a Startup is not orderly and once CLEAR, is not SET until the RAM value of *Clock* rolls over.	TPM2_Clear(), TPM2_Startup(), passage of time

37.4.3 NV Clear Data

Data in this structure is saved to NV on any Shutdown(STATE) but is set to its default initialization value if the subsequent Startup is either TPM Reset or TPM Restart. In the reference implementation, data of this type is collected into a single data structure (STATE_CLEAR_DATA) as illustrated in Table 32.

NOTE The default reset value is applied on either TPM Reset or TPM Restart. These change conditions are not listed in the “Changed By” column.

 Copyright © TCG 2006-2020

Table 32 — Contents of the STATE_CLEAR_DATA Structure

Parameter	Description	Changed By
shEnable	the enable for the storage hierarchy. The default initialization value is SET.	TPM2_HierarchyControl()
ehEnable	the enable for the endorsement hierarchy. The default initialization value is SET.	TPM2_HierarchyControl()
phEnableNV	the enable for the platform hierarchy NV indices. The default initialization value is SET.	TPM2_HierarchyControl()
platformAlg	the hash algorithm used for *platformPolicy.* The default initialization value is TPM_ALG_NULL	TPM2_SetPrimaryPolicy()
platformPolicy	the policy used if the authorization session is a policy session and the authorized handle is TPM_RH_PLATFORM. The default initialization value is an Empty Buffer.	TPM2_SetPrimaryPolicy()
platformAuth	the authorization value used if the authorization handle is TPM_RH_PLATFORM and the authorization is provided by password or an HMAC session. . The default initialization value is an Empty Buffer.	TPM2_HierarchyChangeAuth()
pcrSave	a data structure that holds the PCR that are preserved across Startup(STATE). The PCR in this structure are determined by a platform-specific TPM specification. . The default initialization value for each PCR is determined by the relevant platform-specific specification but is normally a Zero Digest for each PCR in the structure.	TPM2_PCR_Extend(), TPM2_PCR_Event()

37.4.4 NV Reset Data

Data in this structure is saved to NV on any Shutdown(STATE) and restored by a subsequent Startup of any type. In the case of a TPM Reset, the values are set to their specified initialization value. In the reference implementation, data of this type is collected into a single data structure (STATE_RESET_DATA) as illustrated in Table 33.

Table 33 — Contents of the STATE_RESET_DATA Structure

Parameter	Description	Changed By[(1)]
nullProof	proof value used with entities associated with the TPM_RH_NULL hierarchy (including all session contexts, sequences, and Temporary Objects); initialization value is from the RNG	
nullSeed	seed value used for creating Temporary Objects with TPM_RH_NULL as a parent; initialization value is from the RNG	
clearCount	a value that is incremented each time the TPM performs a TPM Restart; used to tag contexts for *stClear* objects so that they may not be reloaded after a TPM Restart; initialization value is zero	TPM2_Startup(CLEAR)
objectContextID	counter that is incremented each time an object is context saved; used to ensure that the encryption key and IV for each saved object is unique; initialization value is zero	TPM2_ContextSave()

Parameter	Description	Changed By(1)
contextArray	an array for keeping the version numbers of the associated saved session contexts; used to prevent replay of authorization sessions; each element is initialized to zero indicating that it is not assigned	TPM2_ContextLoad(), TPM2_ContextSave(), TPM2_StartAuthSession()
contextCount	the value used to set the version number for each saved context; initialization value is 0.	TPM2_ContextSave(), TPM2_StartAuthSession()
commandAuditDigest	the current command code audit digest; initialization value is an Empty Digest.	Any audited command, TPM2_GetCommandAuditDigest()
restartCount	counts the number of TPM Resume, TPM Restart, or D-RTM events. Initialization value is zero.	TPM2_Startup(), _TPM_Hash_End
pcrUpdateCounter	counts the number of changes to PCR; because this value is used in policy sessions, it is not reset until the context protections for saved session contexts are changed. Initialization value is zero	TPM2_PCR_Extend(), TPM2_PCR_Event(), TPM2_PCR_Reset()
commitCounter	the number of times TPM2_Commit() is executed; initialization value is zero.	TPM2_Commit()
commitNonce	value used to create the pseudo-random values used in two-phase signing operations; initialization value is from the random number generator.	
commitArray	bit vector used to indicate that only one first phase of a two phase signing operation has occurred; initialization value is all bits CLEAR.	sign-phase of two-phase sign, TPM2_Commit()
NOTE (1) The default reset value is applied on each TPM Reset. This change condition is not listed in the “Changed By” column.		

37.5 Persistent NV Data

The data in this category is data that is always present in the TPM. This does not mean that the data cannot be changed, but that there is always a value associated with the location. The data can be changed by a Protected Capability.

In the reference implementation, the persistent NV data is in the PERSISTENT_DATA structure. It contents are listed in Table 34. While this table shows the context of the structure in the reference implementation, it is only illustrative. An implementation may change the contents in order to satisfy the requirements of the implementation.

Table 34 — Contents of the PERSISTENT_DATA Structure

Parameter	Description	Changed By
disableClear	This value is CLEAR if TPM_RH_OWNER is allowed for authorization of TPM2_Clear().	TPM2_ClearControl(), TPM2_Clear()
ownerAlg	the hash algorithm used for the *ownerPolicy*	TPM2_SetPrimaryPolicy(), TPM2_Clear()
ownerPolicy	the policy used if the authorization session is a policy session and the authorized handle is TPM_RH_OWNER	TPM2_SetPrimaryPolicy(), TPM2_Clear()
endorsementAlg	the hash algorithm used for the *endorsementPolicy*	TPM2_SetPrimaryPolicy(), TPM2_Clear()
endorsementPolicy	the policy used if the authorization session is a policy session and the authorized handle is TPM_RH_ENDORSEMENT	TPM2_SetPrimaryPolicy(), TPM2_Clear()

Parameter	Description	Changed By
ownerAuth	the authorization value used if the authorization handle is TPM_RH_OWNER and the authorization is provided by password or an HMAC session	TPM2_HierarchyChangeAuth(), TPM2_Clear()
endorsementAuth	the authorization value used if the authorization handle is TPM_RH_ENDORSEMENT and the authorization is provided by password or an HMAC session	TPM2_HierarchyChangeAuth(), TPM2_Clear()
lockoutAuth	the authorization value used if the authorization handle is TPM_RH_LOCKOUT and the authorization is provided by password or an HMAC session	TPM2_HierarchyChangeAuth(), TPM2_Clear()
lockoutAlg	the hash algorithm used for the *lockoutPolicy*	TPM2_SetPrimaryPolicy(), TPM2_Clear()
lockoutPolicy	the policy used if the authorization session is a policy session and the authorized handle is TPM_RH_LOCKOUT	TPM2_SetPrimaryPolicy(), TPM2_Clear()
epSeed	the seed value for the Endorsement Hierarchy	TPM2_ChangeEPS()
ehProof	the proof value for the Endorsement Hierarchy. It is used to tag tickets and saved object contexts for objects in the Endorsement Hierarchy.	TPM2_ChangeEPS() TPM2_Clear()
spSeed	the seed value for the Storage Hierarchy	TPM2_Clear()
shProof	the proof value for the Storage Hierarchy. It is used to tag tickets and saved object contexts for objects in the Storage Hierarchy.	TPM2_Clear()
ppSeed	the seed value for the Platform Hierarchy	TPM2_ChangePPS()
phProof	the proof value for the Platform Hierarchy. It is used to tag tickets and saved object contexts for objects in the Platform Hierarchy.	TPM2_ChangePPS()
resetCount	a counter that increments on each TPM Reset	TPM Reset, TPM2_Clear()
totalResetCount	a value that increments on each TPM Reset. This value is used as *resetValue* in equation (52) to tag saved contexts.	TPM Reset
pcrPolicies	This structure is used when a platform-specific specification requires that update of certain PCR requires policy authorization.	TPM2_PCR_SetAuthPolicy()
pcrAuthValues	This structure is used when a platform-specific specification requires that update of certain PCR requires HMAC or password authorization	TPM2_PCR_SetAuthValue()
pcrAllocated	This structure is used when a platform-specific specification requires support for TPM2_PCR_Allocate() to change the algorithms used for PCR and the population of the PCR in each bank.	TPM2_PCR_Allocate()
ppList	In the reference implementation, this is an array of bits that is used to indicate the commands that require assertion of Physical Presence when TPM_RH_PLATFORM is used for authorization.	TPM2_PP_Commands()

 Copyright © TCG 2006-2020

Parameter	Description	Changed By
failedTries	count of the number of authorization failures for objects that are subject to Dictionary Attack protection. This value can count down if no authorization failures occur for *lockoutRecovery* time.	TPM2_DictionaryAttackLockReset(), authorization failures, passage of time (*recoveryTime*)
maxTries	the maximum value for *failedTries* before the TPM enters lockout	TPM2_DictionaryAttackParameters()
recoveryTime	the time that must pass before *failedTries* is decremented	TPM2_DictionaryAttackParameters()
lockoutRecovery	the time that must pass after an authorization failure using TPM_RH_LOCKOUT	TPM2_DictionaryAttackParameters()
lockoutAuthEnabled	when CLEAR, TPM_RH_LOCKOUT may not be used for authorization	TPM_RH_LOCKOUT auth failure, passage of time (lockoutRecovery)
orderlyState	between a TPM2_Shutdown() and _TPM_Init, no TPM command caused a change to the TPM's state to make the state in NV inconsistent with the state in TPM RAM	many
auditCommands	in the reference implementation, a bit array indicating which commands are audited	TPM2_SetCommandCodeAuditStatus()
auditHashAlg	the hash algorithm used for the command audit	TPM2_SetCommandCodeAuditStatus()
auditCounter	a counter that increments on the first audited command following a reset of the command audit digest. The count is only incremented if the command completes with TPM_RC_SUCCESS.	audited command
algorithmSet	this is a vendor-specific value that indicates the algorithm set that is in use on the TPM. This value may be used selectively to disable algorithms implemented in the TPM.	TPM2_SetAlgorithmSet()
firmwareV1	the more significant 32-bits of the vendor-assigned, firmware revision	TPM2_FieldUpgradeStart(), TPM2_FieldUpdradeData()
firmwareV2	the less significant 32-bits of the vendor-assigned, firmware revision	TPM2_FieldUpgradeStart(), TPM2_FieldUpdradeData()

37.6 NV Rate Limiting

The TPM is allowed to limit the rate at which updates are made to NV memory. An update occurs when an NV Index is defined or undefined, when an NV Index is modified, and when the persistence of an object is changed with TPM2_EvictControl(). An NV modification is allowed for other commands in an implementation dependent way. The rate for limiting the updates is TPM dependent.

When the TPM will prevent execution of a command because it is rate-limiting NV updates, the TPM will return TPM_RC_NV_RATE. This code is in the group of warning return codes meaning that the command might succeed if retried later.

NOTE 1 Checking to see if the NV is being rate limited may occur at any part of the command execution. This means that the TPM may return TPM_RC_NV_RATE before it has validated all of the parameters of the command. As a consequence, when the command is retried when the TPM is not rate limiting, it may fail due to incorrect parameters.

 Copyright © TCG 2006-2020

TPM2_GetCapability() with *capability* = TPM_CAP_PROPERTIES and *property* = TPM_PT_NV_WRITE_RECOVERY will provide an estimate of the number of milliseconds before the TPM will be able to accept a command that will modify the TPM NV.

NOTE 2 After TPM2_Shutdown(), any command is allowed to cause a change of the TPM's orderly shutdown state and the TPM may return TPM_RC_NV_RATE in response to commands that are not normally allowed to make modifications to the TPM NV state.

37.7 NV Other Considerations

37.7.1 Power Interruption

A TPM is not required to maintain the integrity of the data in an NV Index if a power loss interrupts the write. After the interruption, the TPM should indicate that the Index no longer exists. The interruption of a write to one Index is not allowed to affect the integrity of other Indices.

37.7.2 External NV

37.7.2.1 Introduction

An implementation is allowed to use an external device for storing non-volatile TPM data. This may include all application defined NV (NV Indices and persistent objects) as well as all TPM state data. When stored in an external device, the data is required to be encrypted, integrity checked, and rollback protected using algorithms that have the highest security strength of any algorithm implemented on the TPM.

The encryption keys used to encrypt the data in the NV shall be protected in a manner which is defined by the TPM profile which is being implemented. The level and manner of protection for these keys shall also be specified and shall be at least as strong as the keys themselves. For a chip-based implementation, the encryption keys used to encrypt the data stored in NV are not allowed to be exposed outside of the TPM, even if encrypted.

The protection keys used to protect external NV data will be contained in or derived from a persistent value that does not leave the physical TPM. That persistent value must not be a global secret.

NOTE In many implementations, it is expected that the persistent values will be stored in fuses.

37.7.2.2 Access Interruptions

When an external device is used for non-volatile storage, that device may not always be accessible to the TPM command execution engine. When the memory is not accessible, operations that require update of NV will return TPM_RC_NV_UNAVAILABLE.

NOTE When updates to NV are being rate limited (but the NV is accessible), the TPM will return TPM_RC_NV_RATE.

During the time when NV is not available for update, *Clock* should not advance and *Safe* should be NO when accessed.

When NV is not available, the implementation may or may not advance *Clock.* If *Clock* is not being advanced, the TPM will return TPM_RC_NV_UNAVAILABLE for commands that do comparisons to *Clock* or adjustments of *Clock.* These commands are:

- TPM2_PolicySigned() or TPM2_PolicySecret() with a non-zero *expiration*;

 Copyright © TCG 2006-2020

- TPM2_PolicyTicket(); and
- TPM2_PolicyCounterTimer() if any part of TPMS_TIME_INFO.*clockInfo.clock* is used in the operation.

When NV is not available, the implementation may or may not advance *Time*. If *Time* is not being advanced, then TPM2_PolicyCounterTimer() will return TPM_RC_NV_UNAVAILABLE if any part of TPMS_TIME_INFO.time is used in the operation.

37.7.3 PCR in NV

If a TPM implementation places PCR in NV space, it should also use a caching scheme to prevent NV wearout.

 Copyright © TCG 2006-2020

38 Multi-Tasking

An implementation of the TPM may use cycles of a host processor for execution. The operating system on the host processor may not be able to operate properly if the TPM uses large blocks of time to complete execution of a command. In such systems, the TPM may be designed to yield after completion of a portion of the command so that the command may be resumed later.

When the TPM yields before completion of a command, it may return TPM_RC_YIELDED. This code indicates that the exact command that the TPM was executing may be resubmitted later. If the next command to the TPM is not the yielded command, the TPM may lose any state associated with the command that yielded so that when the yielded command is restarted, it may restart from the beginning.

 Copyright © TCG 2006-2020

39 Errors and Response Codes

39.1 Error Reporting

When a command fails, the TPM will return a 10-octet response that indicates the response code. No auxiliary information is provided by an error other than what may be inferred from the context of the error.

39.2 TPM State After an Error

When the TPM returns an error that is related to command execution, the TPM is required to preserve the TPM state. Except for the possible effect on the dictionary attack logic, it should be as though the command had not been received.

In some cases, an otherwise asynchronous operation may cause the TPM to create an error. For example, if the TPM is doing self-test of functions on an as-needed basis, the TPM may return an error due to failure of the self-test. The TPM should preserve the fact that it has failed the self-test but it should not preserve any command-specific results.

When a command modifies NV RAM, the action of writing the NV may fail and it may not be recoverable. If the TPM cannot recover from the NV write failure, then it should disable the NV so that the affected NV locations cannot be accessed.

39.3 Resource Exhaustion Warnings

39.3.1 Introduction

The executable specification has been optimized for comprehension and correctness. In particular, the reference implementation has been designed to minimize the locations in the code where resource exhaustion can occur, so that recovery from these situations is simplified. This is known not to achieve an efficient use of limited RAM resources, and other implementations may choose methods that are more aggressive in their use of memory. These implementations will invariably have error conditions that are not covered in the normative clauses of the reference implementation. This clause describes the methods that are recommended for reporting of these errors.

Allocated resources are classified by their persistence relative to a command's execution. A transient resource is one that can be moved to or from TPM memory using a context management command (TPM2_ContextLoad(), TPM2_ContextSave()). These resources may continue to occupy TPM memory after completion of a command. A temporary resource is used in the processing of a command but is disposed of before the command completes. The following two clauses describe the expected behavior of the TPM when it is unable to create either of these resource types.

39.3.2 Transient Resources

The TPM reference implementation allocates space for a configuration-defined number of transient resources of the maximum size supported by the configuration parameters. This allocation occurs during the compilation process of the reference implementation. The maximum size of the objects is determined by the structure definitions in TPM 2.0 Part 2. The reference implementation presumes that, if a resource slot is available, then any object that might be stored in that slot will fit.

A practical consequence of this approach is that the only resource allocation failure for a transient resource occurs when all the dedicated slots of the appropriate type (object, sequence object, or session) are full. For objects, the number of available slots determines when the resources are all used. For sessions, there are two slot resources: handles and session contexts.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

When the TPM is out of object slots, it returns TPM_RC_OBJECT_MEMORY. When out of session context slots, it returns TPM_RC_SESSION_MEMORY. When the TPM is out of handle slots for sessions, the response code is TPM_RC_SESSION_HANDLES.

For a system using dynamic allocation of memory for transient resources, the TPM should return an error response code that indicates the type of resource that needs to be removed from the TPM for the command to complete. If removal of either an object or a session from TPM memory would free memory for the command, then the TPM may return TPM_RC_MEMORY. If removal of a specific resource is required, the TPM should return a code that indicates the specific resource (TPM_RC_OBJECT_MEMORY or TPM_RC_SESSION_MEMORY).

39.3.3 Temporary Resources

The TPM reference implementation is designed so that temporary resources are allocated on the execution stack. Static analysis of the code allows the maximum size of the stack to be determined so that resource exhaustion for a temporary resource cannot occur.

This construction vastly simplifies the control flow of the normative command actions, since no additional memory management code is required. However, other memory management schemes for temporary resources are allowed. Error handling for these implementations is complex and beyond the scope of this specification. However, the TPM is required to follow the standard error reporting rules.

- If the TPM returns an error, the state of the TPM is required to be restored to the state that existed before the command execution began.

NOTE 1 One exception is state that would change even if a command were not executed, such as *Clock*, *Time*, dictionary attack lockout recovery, and related state. Another exception is state deliberately changed as a result of the error, such as the count of authorization failures and NV PIN Fail index values.

- The TPM will return TPM_RC_MEMORY if removal of one or more transient resources will allow the command to complete.

NOTE 2 If the TPM requires the removal of a specific type of resource, then it should return the specific response code (TPM_RC_SESSION_MEMORY or TPM_RC_OBJECT_MEMORY) rather than the non-specific TPM_RC_MEMORY response.

- If a session must be flushed before a new session can be created, the TPM will return TPM_RC_SESSION_HANDLES.

The consequence of these requirements is that the TPM is required to be able to return the memory allocation to the same state that existed before the command execution began. It is also required that no change to NV memory be made before all temporary resources required for completion of the command have been allocated.

39.4 Response Code Details

The response code from the TPM is a 32-bit value but the TPM only uses the low-order 12 bits to communicate its warnings or errors, leaving the remaining 20 bits for use by software.

The response codes are encoded so that certain errors can be associated with the component in which the error occurred, and the specific element of the component. In cases where the error cannot be associated with a specific parameter of the command, the response code will be sufficiently differentiated to allow determination of the cause of the error.

EXAMPLE 1 If the second handle in the handle area was the wrong type for the command, the TPM would return TPM_RC_VALUE + TPM_RC_H + TPM_RC_2.

 Copyright © TCG 2006-2020

EXAMPLE 2 If the TPM can determine that the error was in the handle area but not the handle in error, the TPM would return TPM_RC_VALUE + TPM_RC_H.

The design of the response codes was constrained so that the response codes returned for commands defined in this specification would be different from the response codes defined by the previous version of the specification, TPM 1.2. This constraint leads to a layout that satisfies the requirements but is not intuitive.

An algorithm for evaluating the response code to determine the nature of the error and the command handle, session, or parameter value in error is shown in the Figure 27 flow chart.

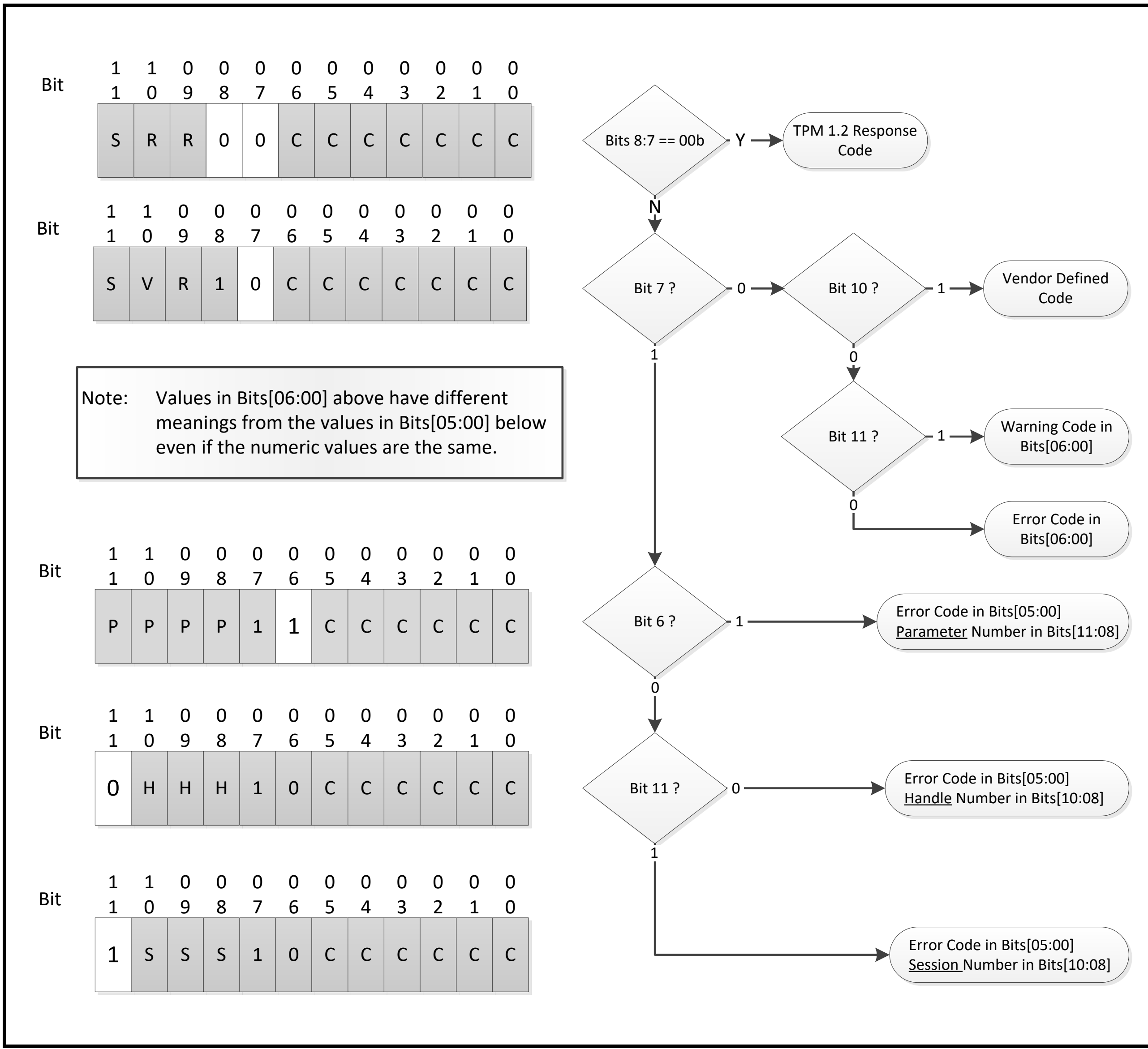


Figure 27 — Response Code Evaluation

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

40 General Purpose I/O

A TPM may have one or more I/O pins that inputs or outputs a logic state. TPM2_NV_Read and TPM2_NV_Write may be used to access the value of GPIO using normal access controls.

A platform-specific specification defines the mapping of NV Indices to individual General Purpose I/O (GPIO). Whether the TPM reserves any NV storage for the indicated GPIO is platform specific.

This specification does not require the NV Indices associated with GPIO pins to be pre-allocated. When one of the Indices reserved for GPIO pins is defined, it is automatically associated with the corresponding GPIO pin.

NOTE 1 The owner and platform space are segregated and it is expected that the GPIO pins will be assigned to Index values in the Index space reserved for the platform.

NOTE 2 The TCG maintains a registry of reserved NV Index values.

The controls that let the GPIO pin be used either as an input or an output are vendor or platform specific.

For outputs, if the Index has the TPMA_NV_ORDERLY attribute SET, the output state is volatile, and becomes non-volatile on an orderly shutdown. If the TPMA_NV_ORDERLY attribute is CLEAR, the output state is non-volatile.

For inputs, a read of the Index returns TPMI_YES_NO, where YES indicates a logic 1 and NO indicates a logic 0 on the input pin.

 Copyright © TCG 2006-2020

41 Minimums

41.1 Introduction

This clause lists the minimums for specific functional blocks where a minimum is needed to ensure proper TPM operation.

Platform-specific TPM specifications may impose other minimums but those minimums are not allowed to be less than the minimums in this specification.

41.2 Authorization Sessions

An active authorization session is a session that is currently loaded into TPM memory and can be addressed with a session handle in a command. A concurrent session is an authorization session that either is loaded on the TPM or has its context saved.

A command may require no more than three sessions divided according to the needs of the command. The TPM is required to be able to support execution of a command with three authorization sessions.

The management of sessions is different from the management of objects. Management software can keep the contexts for an indefinite number of objects and load them as required. The number of concurrent sessions, however, is limited by the resources that the TPM can devote to tracking those sessions.

The TPM should support a minimum of 64 concurrent sessions. Fewer sessions would impair the ability of the TPM to conduct concurrent operations with multiple users.

41.3 Transient Objects

In order to be able to execute all commands, the TPM needs to have two active, loaded objects of any type. A Transient Object is an object that occupies TPM memory and may be referenced by handle. The number of Transient Objects that the TPM supports does not include those objects that have been placed in persistent TPM memory.

NOTE A TPM implementation may copy an object from persistent storage into a Transient Object slot in order to speed up access to the object data.

41.4 NV Counters and Bit Fields

All TPM implementations should allow at least one NV Index to be allocated for use as a monotonic counter (TPMA_NV_COUNTER) or bit field (TPMA_NV_BITS). The number of these Index types determines how many different policies may include revocation as part of their logic. When the number of these Index types is too small, the software complexity of handling revocation becomes too complex to manage.

NOTE 1 This minimum (1) may be adequate for a TPM in a simple embedded system but is too low for a TPM in a complex system such as a PC. Platform-specific specifications for more complex systems should mandate support for at least sixteen (16) counter or bit field Indices.

NOTE 2 The requirement that a TPM support the TPMA_NV_COUNTER or TPMA_NV_BITS attribute implies that the TPM is required to implement either TPM2_NV_Increment() or TPM2_NV_SetBits().

 Copyright © TCG 2006-2020

42 Attached Components

42.1 Introduction

42.1.1 Purpose

The TPM has an extensive set of commands that allow implementation of flexible access control for objects contained in the Shielded Locations of the TPM. The pair of commands TPM2_AC_Send() and TPM2_Policy_AC_SendSelect() enable the access control mechanisms of a TPM to be used to manage sensitive data (such as keys) for other components (called the Attached Components) that are attached to the TPM.

Examples of Attached Components are encrypting disk controllers, crypto accelerators, and network adapters with crypto capabilities.

42.1.2 Concept

An Attached Component (AC) is physically connected to a TPM through a data channel other than the TPM's command and response buffer (the details of the TPM-to-AC data channel are beyond the scope of this document). A properly authorized TPM2_AC_Send() will cause the TPM to copy the selected TPM Object to a selected AC over the provided data channel. This avoids having to "bounce" the Object via an out of band transfer from the TPM through memory then to the component. Object data is, therefore, duplicated outside the TPM's "Shielded Location". The platform manufacturer provisions the connection between the TPM and the Attached Component, and hence determines the protection of an object in transit from the TPM to the Attached Component.

TPM2_AC_Send() uses the DUP auth role that requires authorization with a policy with *policySesssion→commandCode* set to TPM_CC_AC_Send.

NOTE TPM2_AC_Send() will only send objects that are constructed with a policy that allows TPM2_AC_Send().

42.2 TPM2_AC_Send()

TPM2_AC_Send() instructs the TPM to copy portions (possibly all) of the Object referenced by *sendObject* to the Attached Component referenced by *ac*. The methods used to send the Object and the properties of the Attached Component are outside the scope of this specification. The command does not cause the referenced object to be flushed.

The applications using this TPM and the creators of the TPM's objects are expected to understand the properties of the Attached Component and the connection between the TPM and the Attached Component.

The use of Enhanced Authorization allows the object creator to restrict sending the object using any combination of Policies. For example, the policy may only allow an Object to be sent when PCR and Locality have specific values. TPM2_AC_Send() requires DUP role authorization for the *sendObject.* This means that *sendObject* authorization requires a policy session that has *policySession→commandCode* set to TPM_CC_AC_Send. This requirement ensures that TPM2_AC_Send() is only used on Objects that are intended to be sent to an Attached Component.

The *acDataIn* parameter allows TPM2_AC_Send() to send qualifying data to an Attached Component along with the Object. For example, *acDataIn* may be used to identify a key slot in the AC into which the Object is to be loaded.

An Attached Component optionally returns *acDataOut* information to the caller. A possible use of *acDataOut* would be for the AC to indicate the key slot into which it has placed the Object.

After TPM2_AC_Send() completes, the Object in the TPM may be flushed without effecting the values in the AC. Similarly, the AC may modify (or delete) the Object data from its memory without affecting any Object in TPM Shielded Locations.

42.3 Send Object Types

The TPM does not restrict the type of Object that may be sent to an AC. However, an AC may not be able to process all types of TPM Objects. The AC may be designed to reject unknown Object types or it may be the responsibility of system software to ensure that only the proper Object types be sent to an AC.

42.4 Send Object Attributes

The *sendObject* shall have *fixedTPM*, *fixedParent* and *encryptedDuplication* CLEAR.

42.5 Attached Component Authorization

TPM2_AC_Send() requires authorization for the Object to be sent and for the AC that is the target of the send operation.

EXAMPLE If an Attached Component is an encrypting disk controller, unauthorized software, such as ransomware, cannot set the encryption key without getting access to the proper authorization. Use of the PCR as part of the policy for the object can ensure that the disk encryption key can only be set early in the boot phase.

Proper authorization to send to an AC is determined by the presence of an aliased NV Index. The range of AC handles is AC_FIRST to AC_LAST. The range for aliased AC Indexes is NV_AC_FIRST to NV_AC_LAST. If the Owner or Platform creates an NV Index in the range AC Indexes and a corresponding AC exists in the AC handle range, then the write authorization settings of the AC Index (TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, TPMA_NV_AUTHWRITE, and TPMA_NV_POLICYWRITE) determine how the send may be authorized. If no aliased AC Index exists, then either Owner Authorization or Platform Authorization may be used to allow the send to the AC.

The Platform and Owner may use TPM2_NV_DefineSpace() to delegate the rights to access a specific AC The Platform or Owner may change the access rights to the AC by calling TPM2_NV_UndefineSpace() and then TPM2_NV_DefineSpace() with new parameters.

TPM2_NV_ChangeAuth() allows the current key and object manager to change the *authValue* by using the current AC authorization policy.

The following is a summary of states and allowed actions:

a) Attached Component has no aliased NV Index defined
    1) Can use TPM2_AC_Send() to send objects to the Attached Component with *ownerAuth* or *platformAuth*
    2) Can use TPM2_AC_GetCapability() to get optional information about the AC
b) Attached Component has an aliased NV Index
    1) Can use TPM2_AC_Send() to send objects to the Attached Component with the write authorization types allowed by the aliased Index.
    2) Can change authorization using ADMIN role (TPM2_NV_ChangeAuth()).

TPM2_Clear will delete any AC Index alias defined by the Owner but not by the Platform.

42.6 Attached Component Object Management

42.6.1 Discovery

Applications may use information from the platform manufacturer (for example, a platform certificate) to determine whether a platform has an Attached Component. Software may also discover the number of attached components by calling TPM2_GetCapability(TPM_CAP_HANDLES) with a TPM_HT_AC Property Type to find available AC handles.

Specific information about each Attached Component may be provided by calling the TPM2_AC_GetCapability() command. In response to this command, the TPM returns vendor-specific information about a specific Attached Component. For example, *capabilitiesData* may be used to return a pairing value that can be also obtained from the AC itself by an AC-specific API (see Part 2, *Attached Component Structures*).

The association between ACs and AC handle is vendor specific, but the association is required to be constant during the platform's lifetime. If an AC is replaced, the new AC should have the same AC handle as the AC that was replaced.

After reboot or other TPM power state changes, TPM2_GetCapability() should be used to obtain the list of the available ACs.

NOTE 1 It is suggested that no handle be associated with a defective AC. However, it is also allowed that a defective AC be reported using TPM2_AC_GetCapability().

TPM2_GetCapability() can be used at any time to determine the current AC configuration.

NOTE 2 There is no mechanism for the TPM to proactively signal the change in state of an AC. However, a platform may have a method for the AC to signal software and for that indication to trigger an enumeration process.

42.6.2 Setup

The AC's configuration should be performed by the AC's Manufacturer. AC configuration is out of scope of the TCG.

42.6.3 Sending

If the desired Object is not already loaded, the application (such as a key manager) may load it using an existing TPM2 command such as TPM2_Load(). This removes the wrapping of the object and returns an object handle. However, if an object is evicted and then re-loaded, there is no assurance the TPM will assign the object the same object handle. For this reason, applications should not associate the TPM's assigned object handle with the object when sent to the Attached Component, or when the object is within the Attached Component's domain. Key and object managers using the TPM2_AC_Send() command may assign, if necessary, a handle for the transferred object within the Attached Component domain using the *acDataIn* parameter. Therefore, in common usages there is no association between the TPM generated object handle assigned to the object and any handle for that object within the Attached Component's domain. Any association between the TPM's object and the Attached Component's object is done by the application or the application's resource and object manager external to the TPM.

 Copyright © TCG 2006-2020

42.7 Power States

The AC authorization values and policies that are setup by TPM2_NV_DefineSpace() are persistent across all TPM power state changes, like the authorization values and policies of other NV Indexes.

TPM power state changes have no effect on the Attached Component's copy of objects, because that copy of the object is no longer in the TPM's "Shielded Location".

42.8 Attached Component Format

The format and security properties of the connection between the TPM and the Attached Component are outside the scope of the TPM Library Specification. A vendor may provide a proprietary connection between the TPM and an Attached Component where the format and the security properties are defined by that vendor. A TCG platform-specific Working Group may define a data format that may include a key exchange method so that independently manufactured Attached Components can interoperate with different TPMs.

 Copyright © TCG 2006-2020

43 Authenticated Countdown Timer (ACT)

43.1 Introduction

The functionality and commands described in this clause enable the TPM to manage multiple authenticated countdown timers (ACT).

43.2 Description

An ACT is a 32-bit counter that, when not already zero, will decrement by one each second that the TPM is powered.

The countdown timers are used to trigger events on a platform when they count down to zero, at which point they are said to timeout or expire. TPM2_ACT_SetTimeout() is used to set an ACT to a non-zero value and begin the timeout. On TPM Reset or TPM Restart, all ACT timeouts are set to zero with no side effects (no event triggered). ACT timeouts are preserved across TPM Resume.

The ACT timeouts are saved by TPM2_Shutdown(STATE). On TPM2_Startup(STATE), if the TPM shutdown was orderly, the saved ACT values are restored and the ACT resumes counting. If an ACT *startTimeout* has been written (TPM2_ACT_SetTimeout()) since the last TPM2_Startup(), then the current timeout of the ACT is saved by TPM2_Shutdown(STATE); otherwise, the saved value is one half of the current ACT timeout. If a TPM2_ACT_SetTimeout() occurs after the TPM2_Shutdown(), then the TPM state is no longer orderly, and a subsequent TPM2_Startup(STATE) will fail.

An ACT has an *authValue* and an *authPolicy*. The *authValue* is the same as the current *platformAuth* and can only be used if *phEnable* is SET. The *authPolicy* is ACT-specific and is neither enabled nor disabled by *phEnable*. Each ACT has its own *authPolicy*.

NOTE 1 A system might continue to operate after a TPM2_Shutdown(STATE). Therefore, saving half the timeout prevents an attacker from continually extending the timeout by doing TPM2_Shutdown(STATE) immediately after TPM2_Startup(STATE), and then restarting the system (TPM Resume) just before the timer expires.

TPM2_ACT_SetTimeout() must be properly authorized. Authorization may be provided either by *platformAuth* or by an ACT-specific *authPolicy*. The *startTimeout* parameter in TPM2_ACT_SetTimeout() is an integer number of seconds.

The *authPolicy* for an ACT can be changed by TPM2_SetPrimaryPolicy() using either *platformAuth* or the ACT-specific *authPolicy*.

The *authPolicy* of an ACT is initialized to an Empty Policy by TPM Reset or TPM Restart but is preserved during TPM Resume.

NOTE 2 After TPM Reset or TPM Restart, *phEnable* is SET, allowing the platform to initialize any ACT *authPolicy*.

43.3 Typical Use

A typical example for the use of an ACT is as a watchdog timer that will cause a platform reset when the timer reaches zero (expires). In a system using an ACT, a periodic platform action outside the TPM indicates that the timeout should be set anew using TPM2_ACT_SetTimeout(). The most common reason why timeout is not set anew is that the local system is not behaving properly because of some type of corruption (either inadvertent or malicious). The intent of the timer is that, in the absence of a properly authorized timeout extension, the platform would be reset, putting it back into a known state with the

 Copyright © TCG 2006-2020

expectation that the corruption can be removed. The reason for having an authenticated timeout is to allow an external entity to make a decision about the health of the system.

The example above is not the only one supported by an ACT. In fact, this specification does not mandate that any specific platform behavior occur as a result of a timer expiring. The action on timer expiration may be chosen by a platform-specific specification or be vendor specific.

Because *platformAuth* may be used to change the *authPolicy* or set a *startTimeout* for any ACT, the platform firmware has ultimate control of the ACT. On each TPM Reset or TPM Restart, the platform firmware is expected to set the *authPolicy* for all ACT using *platformAuth*. This specification mandates no specific policy for any ACT, but it is expected that, in most cases, the platform firmware will either:

a) Initialize an ACT *authPolicy* with a policy that can only be satisfied by an entity trusted by the platform manufacturer; or
b) Initialize the ACT *authPolicy* so that the *authPolicy* can be changed using *ownerAuth.*

In case a), the platform firmware may set an initial timeout to ensure that some corrective action will occur if malware prevents the trusted entity from setting the ACT.

NOTE 1 This is how the platform would typically initialize a watchdog timer

In case b), the system software is expected to take control of the ACT. The platform would not set an initial timeout as it is possible that the ACT will not be used by the system software.

NOTE 2 If the platform firmware does not initialize the ACT *authPolicy* before *phEnable* CLEAR, then the ACT cannot be used.

43.4 Failure Mode

If the TPM enters failure mode, the ACT should continue to count down and trigger the specified event should it expire.

NOTE 1 If the failure mode was caused by a timer failure or affects functionality which is required for the platform-specific event, the ACT might not trigger reliably.

TPM2_ACT_SetTimeout() shall not be usable while a TPM is in Failure Mode. This means that the timeout cannot be extended and that timed events will occur if the TPM is not powered down or Reset before the ACT expires. A platform-specific specification may specify that an event will have no effect if the TPM is in Failure Mode.

A TPM may allow reading of the remaining ACT time (TPM2_GetCapability(*capability* = TPM_CAP_ACT) when the TPM is in Failure Mode.

43.5 Field Upgrade

The behaviour of a TPM during Field Upgrade is undefined. However, it is preferred that ACT continue to operate normally during Field Upgrade except that the ACT may not be changed by TPM2_ACT_SetTimeout().

NOTE Since *platformAuth* is required to start a Field Upgrade, *platformAuth* can be used to set the *startTimeout* for any active ACT to a value that is sufficient to allow a Field Upgrade to complete.

 Copyright © TCG 2006-2020

43.6 Typical ACT authPolicy

This clause describes a typical ACT authorization policy that authorizes setting of *startTimeout* with an authentication credential (key). The signature created by the authentication key is used as a cryptographically protected deferral ticket for the ACT.

The ACT *authPolicy* is constructed using TPM2_PolicySigned() and may include other policy components. Authorization by multiple entities can be achieved by combining multiple TPM2_PolicySigned() commands using AND or OR terms.

The deferral ticket is provided to the TPM in TPM2_PolicySigned() as the *auth* parameter (the signed authorization). The signature verification key, the *authObject* in TPM2_PolicySigned(), may be a symmetric or asymmetric key.

NOTE The advantage of an asymmetric signing key is that only the public key needs to be provisioned into the TPM. In the case of a symmetric HMAC key, the HMAC key's *authPolicy* should restrict the key to be used only for TPM2_PolicySigned() and not for other commands like TPM2_HMAC() (i.e. the TPM should not be able to issue its own deferral tickets).

For the *nonceTPM* and *expiration* parameters of TPM2_PolicySigned(), the following is recommended:

c) *nonceTPM* present and not an Empty Buffer

d) *expiration* > 0

Both settings ensure that a deferral ticket is single-use. The presence of *nonceTPM* in TPM2_PolicySigned() prevents the same signature being used multiple times within a policy session to defer the ACT indefinitely.

NOTE The *nonceTPM* for the policy session changes at the end of TPM2_ACT_SetTimeout(). This invalidates the previous signature and prevents replay of TPM2_ACT_SetTimeout() without getting a new signature from the authorized entity.

The non-negative *expiration* prevents TPM2_PolicySigned() creating a policy ticket which may be reused with TPM2_PolicyTicket() over a period of time to defer the ACT.

The ability to change *startTimeout* of TPM2_ACT_SetTimeout() should be limited by including *cpHash* in the ACT *authPolicy*. This can be achieved in two ways:

1) The ACT authorization policy includes TPM2_PolicyCpHash(). In this case, the entity setting the policy determines the *startTimeout* value.

2) The ACT authorization policy includes TPM2_PolicySigned() with *cpHashA* set. In this case, the signer (of the deferral tickets) determines the startTimeout value.

 Copyright © TCG 2006-2020

Annex A
(informative)
Policy Examples

A.1 Introduction

This clause compares authorization between TPM 1.2 and this specification.

A.2 TPM 1.2 Compatible Authorization

A TPM 1.2 key may have its use gated by PCR and *authValue*. To select this authorization, the key would be created with a *pcrSelection* with at least one bit SET and the *digestAtRelease* set to indicate the digest of the selected PCR. Additionally, the key's TPM_AUTH_DATA_USAGE would be set to TPM_AUTH_ALWAYS. To perform the authorization, an authorization session is created and used to prove knowledge of the *authValue* in the authorization HMAC. If the HMAC check is successful and the digest of the selected PCR matches the *digestAtRelease*, the action is approved.

For a TPM compatible with this specification, use of PCR for access control requires a policy. The policy should be created at the time of object creation so that the policy requires selected PCR to have a specific value. This is similar to determining the *digestAtRelease* in TPM 1.2. The policy will use two factors: PCR and an *authValue*. The first policy command will be TPM2_PolicyPCR() and it will modify the *policyDigest* by:

$$policyDigest_1 \coloneqq \mathbf{H}_{contextAlg}\left(policyDigest_0 \mathbin{||} \text{TPM_CC_Policy_PCR} \mathbin{||} PCR\ Selection \mathbin{||} PCR\ digest\right) \quad (58)$$

where

$\mathbf{H}_{contextAlg}$	hash function using the context hash algorithm
$policyDigest_0$	an array of octets of zero equal in length to the size of the policy digest
TPM_CC_Policy_PCR	a constant indicating the command modifying the *policyDigest*
$PCR\ Selection$	a TPML_PCR_SELECTION that indicates the PCR that will be included in the PCR digest
$PCR\ digest$	the expected digest of the PCR selected by the PCR Selection; the PCR are hashed using the hash algorithm of the policy session

To cause the TPM to compute an HMAC using the *authValue* of the object, a TPM2_PolicyAuthValue() would be included in the policy. It would modify the *policyDigest* as:

$$policyDigest_2 \coloneqq \mathbf{H}_{contextAlg}\left(policyDigest_1 \mathbin{||} \text{TPM_CC_PolicyAuthValue}\right) \quad (59)$$

where

$\mathbf{H}_{contextAlg}$	hash function using the context hash algorithm
$policyDigest_1$	the result of performing the operation in equation (58) above
TPM_CC_PolicyAuthValue	the command code for TPM2_PolicyAuthValue()

The value of *policyDigest*$_2$ would be included in the template of the object in the *authPolicy* parameter.

To use the object, a policy authorization session would be started using TPM2_StartAuthSession(). Then a TPM2_PolicyPCR() and TPM2_PolicyAuthSession() would be executed using the handle of the authorization session. If the PCR were the same as those used when performing the operation of

equation (58), then the *policyDigest* of the policy session will match the *authPolicy* of the object. Because the policy sequence contained TPM2_PolicyAuthValue(), the TPM will check that the HMAC in the authorization indicates that the caller knows the *authValue* of the object (same computation as performed on an HMAC session). If both checks succeed, the object is properly authorized.

Annex B
(normative/informative)
RSA

B.1 Introduction

The RSA asymmetric algorithm is used for digital signatures, secret sharing, and encryption.

A TPM that supports RSA should support a public modulus size of at least 2,048 bits. Support for other key sizes is permitted.

NOTE 1 The reference implementation supports key sizes of 1024, 2048, and 3072.

When the size (k) of the public modulus (n) of an RSA key is given, then $\lfloor log_2 n \rfloor = (k - 1)$. Additionally, for a two-prime system, the primes (p and q) satisfy $\lfloor log_2(p^2) \rfloor = (k - 1)$ and $\lfloor log_2(q^2) \rfloor = (k - 1)$.

The RSA algorithm requires the methods of encryption and signing defined in IETF RFC 8017. This includes support for RSAES-OAEP, RSAES-PKCS1-v1.5, RSASSA-PKCS1-v1.5, and RSASSA-PSS.

The RSA structures in this specification support only public keys that are the product of two primes. Support for other numbers of primes is allowed, but it is performed in a vendor-specific manner and thus beyond the scope of this specification.

A TPM is required only to support a public exponent (e) of $2^{16}+1$. Support for other exponents is allowed but discouraged.

NOTE 2 The reference implementation does not support an exponent size smaller than 7 nor does it allow keys to be created on the TPM with a public exponent less than $2^{16} + 1$.

When loading an RSA key, the TPM validates that its public and private portions are properly paired by dividing the public modulus by the single private prime and requiring that the remainder be zero. The TPM does not validate whether input values are primes.

NOTE 3 Validating the pairing of the public and private key portions need not be performed when the key is being loaded. However, this check is performed before the authorization value of the key or the private portion of the asymmetric key may be used.

The TPM will also validate that the provided and computed prime factors are in an acceptable range. To be acceptable, the square of the prime is required to have the same number of significant bits as the public modulus.

 Copyright © TCG 2006-2020

B.2 RSAEP

This is the RSA public key primitive defined in IETF RFC 8017, clause 5.1.1. It is a modular exponentiation of a message (m) with the public exponent (e), modulo the public modulus (n) to produce the cipher text (c). This is expressed as:

$$c \coloneqq m^e \pmod n \tag{60}$$

where

c	the encrypted message
m	a value $0 < m < n$ to be encrypted
e	the public exponent (default is $2^{16} + 1$)
n	the public modulus

B.3 RSADP

This is the RSA private key primitive defined in PSCS#1v2.1, clause 5.1.2. This clause describes the private key in two forms: as a pair and as a quintuple. The reference implementation uses the pair form with a private exponent (d). Using this form, the RSADP operation recovers a message from a cipher text by:

$$m \coloneqq c^d \pmod n \tag{61}$$

NOTE The reference implementation also supports use of the CRT form of the private exponent.

B.4 RSAES_OAEP

This encryption scheme is defined in IETF RFC 8017. It is the only scheme used with an RSA-restricted decryption key. The algorithm identifier for this scheme is TPM_ALG_OAEP.

For RSA keys protecting a secret value (such as, an encryption key or a session secret), the *L* parameter is a byte stream, the last byte of which must be zero, indicating the intended use of the encrypted value. A command that accepts or creates an RSA-encrypted secret indicates the value of the string to use for *L*. The RSA key's *scheme* hash algorithm (or, if it is TPM_ALG_NULL, the RSA key's Name algorithm) is used to compute $lhash \coloneqq \mathbf{H}(L)$, and the null termination octet is included in the digest.

MGF1 (as defined in IEEE Std 1363™-2000) computes *dbMask* and *seedMask.* The mask-generation function uses the Name algorithm of the RSA key as the hash algorithm.

B.5 RSAES_PKCSV1_5

This encryption scheme is defined in IETF RFC 8017. It has no parameters. The algorithm identifier for this scheme is TPM_ALG_RSAES.

B.6 RSASSA_PKCS1v1_5

This signing scheme is defined in IETF RFC 8017. The algorithm identifier for this scheme is TPM_ALG_RSASSA.

 Copyright © TCG 2006-2020

An RSA-restricted signing key may use either this algorithm or RSASSA_PSS, but not both. An unrestricted signing key may select as its default either this algorithm or RSASSA_PSS. If TPM_ALG_NULL is selected, the caller will specify the scheme in the signing command.

This signature scheme prepends an OID to a digest before signing with the private key. It may be used in any command that allows an asymmetric signing operation.

For signing commands that use restricted signing keys, the TPM provides the OID that corresponds to the digest algorithm, and the OID provided by the caller is discarded.

For commands that use unrestricted signing keys, the TPM uses the caller-provided OID.

NOTE 1 If the command does not provide a parameter for the OID, then the TPM provides the OID even if the key is not restricted.

For hash algorithms where the TCG defines a TPM_ALG_ID, the TCG provides the OID to use with restricted signing keys. Currently, the defined values are:

- SHA1

  30 21 30 09 06 05 2B 0E 03 02 1A 05 00 04 14_{16}

- SHA256

  30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20_{16}

- SHA384

  30 41 30 0d 06 09 60 86 48 01 65 03 04 02 02 05 00 04 30_{16}

- SHA512

  30 51 30 0d 06 09 60 86 48 01 65 03 04 02 03 05 00 04 40_{16}

NOTE 2 These values are from IETF RFC 8017.

NOTE 3 The listing above is not normative. TCG maintains the normative list.

B.7 RSASSA_PSS

This signing scheme is defined in IETF RFC 8017. The algorithm identifier for this scheme is TPM_ALG_RSAPSS.

A restricted signing key may use either this algorithm or RSASSA_PKCS1v15, but not both. An unrestricted signing key may use either this algorithm, RSASSA_PKCS1v15, or TPM_ALG_NULL. If TPM_ALG_NULL is selected, the caller can specify the signing scheme in the signing command.

When used with a restricted signing key, the hash algorithms for messages (M) and M' are the same.

When used with an unrestricted signing key, the hash algorithm for M and M' can differ.

For both restricted and unrestricted signing keys, the random salt length will be the largest size allowed by the key size and message digest size.

NOTE If the TPM implementation is required to be compliant with FIPS 186-4, then the random salt length will be the largest size allowed by that specification.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

B.8 RSA Key Generation

B.8.1 Background

The implementation of the RSA key-generation function should meet the requirements of the intended market. The methods in FIPS 186-3 are recommended.

In the reference implementation, the primes used for the key are generated using the methods of FIPS 186-3, B.3.3 "Generation of Random Primes that are Probably Prime."

NOTE FIPS 186-3 only allows this method to be used for primes of 1024 bits or larger. For smaller primes, the methods described in B.3.5 "Generation of Probable Primes with Conditions Based on Auxiliary Provable Primes" or B.3.6 "Generation of Probable Primes with Conditions Based on Auxiliary Probable Primes" can be used if FIPS compliance is required.

B.8.2 Large Prime Generation

For generating a prime the reference implementation has two different implementations: one using testing of candidates and the other using a number sieve. The process for testing of candidates is described in this clause.

The inputs are:

- $primeSize$ – this is the number of bits in the prime to be generated. It should be half the number of bits in the public modulus to be generated
- e – the public exponent

  NOTE 1 In the reference implementation, the exponent is required to be a prime number > 2^{16}

- a random number generation function according to the type of key being generated (see 27.6.2 and 27.6.3)

  NOTE 2 Derivation of RSA keys is not supported.

The prime generation process is:

a) set prime candidate p to the next *primeSize* number of bits from the provided random number generation function

b) adjust *p* so that the high-order two bits and the low order bit are one

   NOTE 3 In the reference implementation, when a prime is generated, the upper two octets for prime candidates are verified to be B5 05_{16} or greater. This forces the prime to be greater than 0.7071075439453125 * $2^{(n/2)}$ where n is the number of bits in the public modulus. This is slightly larger than the required value of $\sqrt{2}/2$ * $2^{(n/2)}$. This value ensures that the MSb of the product of these to prime will be SET. Setting of the two most significant bits would also ensure that the magnitude of the product is large enough but reduced the range of allowed primes by small factor (about 4.3%).

c) test p to determine if it is probably prime

   1) Using a greatest common divisor ($\mathbf{GCD}$()) function, see if p shares any common factors with a composite number that is the product of the first 1024 primes and if so, go to a).

   2) do N rounds of Miller-Rabin where N is determined by the size of the prime and if the test fails on any round, go to a)

      NOTE 4 The value for N may be found in FIPS 186-3, Table C.2.

NOTE 5 The witness values used by Miller-Rabin are from the same random number function used to generate the prime candidate.

d) return p

B.8.3 RSA Key Generation Algorithm

The key generation process is:

a) initialize the values of the algorithm

1) Set $securityStrength$ according to the size of the public modulus of the key to be generated as specified in SP800-57 part 1.

2) $primeSize$ ≔ 1/2 the size of the RSA modulus (*inPublic.parameters.keyBits* of the template)

b) find a first prime (p) using the method in B.8.2

c) find a second prime (q) using the method in B.8.2:

d) If $|p - q| < 2^{100}$, go to b)

e) compute the public modulus $n \coloneqq p \bullet q$

NOTE 1 Depending on the starting values the algorithm could take many iterations to find two suitable primes.

f) compute the private exponent $d \coloneqq e^{-1}$ (mod $(p - 1)(q - 1)$)

NOTE 2 The reference implementation also provides an option to use the CRT form of the private exponent d.

g) if $d < 2^{nLen/2}$ where $nLen$ is the number of bits in the public modulus (n), then go to step b)

NOTE 3 If required, a random value is encrypted with the public exponent and decrypted with the private exponent to validate that the key can be used for signing and signature verification.

h) return n, p and d

B.9 RSA Cryptographic Primitives

B.9.1 Introduction

When RSA is implemented on a TPM, it may provide these additional commands to support cryptographic operations. The command description in TPM 2.0 Part 3 indicates the restrictions on the types of keys that may be used with each of the commands.

B.9.2 TPM2_RSA_Encrypt()

TPM2_RSA_Encrypt() may be used to perform encryption according to the methods described in IETF RFC 8017. If the scheme of the key is TPM_ALG_NULL, then the encryption scheme may be specified in the command. Otherwise, the scheme specified in the key will be used. The scheme options are:

- TPM_ALG_NULL – selects RSAEP as described in B.2
- TPM_ALG_OAEP – selects RSAES_OAEP as described in B.4
- TPM_ALG_RSAES – selects RSAES_PKCSV1_5 as described in B.5

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

B.9.3 TPM2_RSA_Decrypt()

TPM2_RSA_Decrypt() performs the decryption operations defined in IETF RFC 8017, clause 7.1.2. The handle used in this command is required to have the *decrypt* attribute SET. If the scheme of the key is TPM_ALG_NULL, then the encryption scheme may be specified in the command. Otherwise, the scheme specified in the key will be used. The scheme options are:

- **TPM_ALG_NULL** – selects RSADP as described in B.3
- **TPM_ALG_OAEP** – selects RSAES_OAEP as described in B.4
- **TPM_ALG_RSAES** – selects RSAES_PKCSV1_5 as described in B.5

B.10 Secret Sharing

B.10.1 Overview

When data is to be delivered securely to the TPM a secret sharing mechanism is required. There are three cases when RSA is used for secret sharing:

1) injecting a salt value for an authorization session,
2) exchanging protection values for object duplication, and
3) exchanging protection values for identity credentials.

For each of these uses, a secret value is OAEP encrypted as described in B.4.

The size of the secret value is limited to the size of the digest produced by the *scheme* hash algorithm (or *nameAlg* if the scheme hash algorithm is TPM_ALG_NULL) of the object that is associated with the public key used for OAEP encryption.

B.10.2 RSA Encryption of Salt

In TPM2_StartAuthSession(), when *tpmKey* is an RSA key, the secret value (*salt*) is encrypted using OAEP as described in B.4. The string “SECRET” (see 4.5) is used as the *L*. The data value in OAEP-encrypted blob (*salt*) is used to compute *sessionKey*.

B.10.3 RSA Secret Sharing for Duplication

When the new parent for a duplicated object is an RSA key, a random seed value is created and used in the KDF operations to generate a symmetric encryption key and IV according to equation (33) and an HMAC key according to equation (35). The seed value will be OAEP encrypted to the public key of the new parent as described in B.4 using “DUPLICATE” as the L parameter. The seed size will be the size of a digest produced by the OAEP hash algorithm of the new parent.

On TPM2_Import() the private key of the new parent is used to decrypt the key protector containing the seed value. If the label value in the OAEP encrypted blob is not “DUPLICATE”, then the decryption routine should generate an error. The error should cause the seed value to be set to an invalid value so that the error will not be reported until the integrity HMAC is validated.

NOTE This is to ensure consistency in behavior with ECC and to minimize the information available to an attacker.

B.10.4 RSA Secret Sharing for Credentials

When a credential is protected (such as TPM2_MakeCredential() and TPM2_ActivateCredential()), a random seed value is created and used as described in B.10.3. The only difference is that the label value used for the KDF will be “IDENTITY” instead of “DUPLICATE”

 Copyright © TCG 2006-2020

Annex C
(normative/informative)
ECC

C.1 Introduction

The ECC algorithm is used for digital signatures and for secret sharing.

NOTE 1 As implemented in a TPM, ECC is not used directly for encryption of data. Rather, ECDH secret sharing is used to establish a symmetric key, and then a symmetric algorithm is used for the actual data encryption.

A TPM should support prime modulus ECC.

If the ECC algorithm is supported, the TPM is required to support ECDSA and ECDH (SP800-56A, Clause 6.2.2.2 "One-Pass Diffie-Hellman, C(1, 1, ECC CDH)").

The TPM should support ECC key sizes of at least 256 bits. Support for other key sizes is allowed.

NOTE 2 It is anticipated that the recommended ECC key size will increase over time in revisions to this specification.

The TPM does not check the security of ECC curve parameters. It does check that the public and private keys are properly paired.

NOTE 3 Validating the pairing of the key's public and private portions need not be performed when the key is being loaded. However, this check is required to be performed before the authorization value of the key or the private portion of the asymmetric key may be used.

C.2 Split Operations

C.2.1 Introduction

Several of the EC schemes us two-phase protocols in which the TPM generates an ephemeral key pair in the first phase and uses that ephemeral key in the second phase. These protocols require that the ephemeral key only be used once. Ordinary TPM keys have context that may be saved and restored by TPM context management. This clause describes the methods used to implement the required single use ephemeral keys.

C.2.2 Commit Random Value

A split operation requires two TPM commands the first of which is TPM2_Commit(). It uses a TPM-generated, random value in the commit computation. A second command (such as, any of the signing commands) completes the split signing operation and uses the same commit value. The random commit value is required to:

- have at least the number of bits equal to the security strength of the signing key;
  - not be known outside of the TPM; and
  - only be used once.

Because the random value is not allowed to be known outside of the TPM, the TPM is required to store the random value between the two commands in split sequence. To allow more than one split sequence to be in process at a time, the TPM may have an array of values and return a count value as one of the

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

response parameters of the TPM2_Commit() indicating the array entry being used for the sequence. This count value is an input to the TPM in the command that completes the split sequence.

NOTE The number of split sequences supported by the TPM may be found using TPM2_GetCapability(*capability* = TPM_CAP_TPM_PROPERTIES, *property* = TPM_PT_SPLIT_MAX).

To minimize the size of the array used for storing these values, a TPM may generate pseudo-random values instead.

If using pseudo-random values, the TPM creates the value using **KDFa**(), a counter (*commitCount*), and a random value (*commitRandom*). On each TPM Reset, the TPM will select a new random value for *commitRandom* and reset *commitCount* to zero. On TPM2_Commit(), the TPM would use the current value of the *commitCounter* to generate the pseudo-random value (*r*) by

$$r \coloneqq \mathbf{KDFa}\ (vendorAlg, commitRandom, \text{“ECDAA Commit”}, name, commitCount, bits) \tag{62}$$

where

$nameAlg$	the *nameAlg* of the signing key (*signHandle*)
$commitRandom$	the current value of *commitRandom*
“ECDAA Commit”	value used to differentiate uses of **KDFa**()
$name$	the Name of *signHandle*
$commitCount$	the current value of *commitCount*
$bits$	the number of bits in the order of the curve of the signing key (*signHandle*) NOTE: When the number of bits is not a multiple of 8, it is rounded up to be a multiple of 8.

To track the usage of the *commitCount*, the TPM maintains a bit array (*A*[]) that has a power of 2 number of bits (*N*) (that is, the bits indexes of *A*[] are from 0 to 2^N-1). After computing the value of *r*, the low-order *N* bits of *commitCount* are used to index *A*[] and the corresponding bit is SET. The low-order 16 bits of *commitCount* are returned as the *counter* parameter.

C.2.3 TPM2_Commit()

TPM2_Commit() performs the first part of a split operation. The TPM will perform the point multiplications on the provided points and return intermediate signing values. Alternatively, the TPM will simply return a public ephemeral key based on a commit private value. The *signHandle* parameter refers to an ECC key. TPM2_Commit() has the following parameters, all of which are optional.

$P1$	point on the curve used by *signHandle* (a TPM2B_ECC_POINT)
$s2$	octet array used to derive x-coordinate of a base point (a TPM2B_ECC_PARAMETER)
$y2$	y-coordinate of the point associated with *s2* (a TPM2B_ECC_PARAMETER)

NOTE 1 *P1* is a TPM2B_ECC_POINT, a sized buffer containing a TPMS_ECC_POINT. It is not a sized buffer containing an array of bytes. A size of zero for the TPM2B_ECC_POINT will create an unmarshaling error because the minimum size for *P1* is 4 (two ECC parameters, both of which are Empty Buffers). If *P1* is an Empty Buffer, the TPM returns TPM_RC_INSUFFICIENT regardless of *s2* and *y2*. If *P1* is an Empty Point and *s2* and *y2* are Empty Buffers, then the TPM will set E := [r]G where r is the commit random value and G is the generator point for the curve.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

In the algorithm below, the following additional values are used in addition to the command parameters:

$\mathbf{H}_{nameAlg}$	hash function using the *nameAlg* of the key associated with *signHandle*
p	field modulus of the curve associated with *signHandle*
n	order of the curve associated with *signHandle*
d_s	private key associated with *signHandle*
G	generator of the curve associated with *signHandle*
c	counter that increments each time TPM2_Commit() is executed
$A[i]$	array of bits used to indicate when a value of *c* has been used in a signing operation; the values of *i* are 0 to 2*N*-1.
N	log2 of the number of values in *A*
k	nonce that is set to a random value each on each TPM Reset; the nonce size is twice the security strength of any ECDAA key supported by the TPM

The commit algorithm is:

a) Validate that *s2* and *y2* are either both Empty Buffers or both not Empty Buffers (TPM_RC_SIZE)

b) If s2 is an Empty Buffer, skip to step e)

c) compute $x2 \coloneqq \mathbf{H}_{nameAlg}(s2) \bmod p$

d) if $(x2, y2)$ is not a point on the curve of *signHandle,* return TPM_RC_ECC_POINT

e) if $p1$ is not an Empty Point and *p1* is not a point on the curve of *signHandle,* return TPM_RC_ECC_POINT

f) set K, L, and E to be Empty Buffers

g) generate or derive *r* (see C.2.2)

h) set $r \coloneqq r \bmod n$

i) if *s2* is not an Empty Buffer, set $K \coloneqq [d_s]\,(x2, y2)$ and $L \coloneqq [r]\,(x2, y2)$

j) if *p1* is not an Empty Point, set $E \coloneqq [r]\,(p1)$

k) if $p1$ is an Empty Point and *s2* is an Empty Buffer, set $E \coloneqq [r]\,G$

l) if K, L, or E is the point at infinity, return TPM_RC_NO_RESULT

m) set $counter \coloneqq commitCount$

n) set $commitCount \coloneqq commitCount + 1$

NOTE 2 Depending on the method of generating *r*, it may be necessary to update the tracking array here.

o) output K, L, E and $counter$

NOTE 3 Depending on the input parameters, K and L or E may be Empty Points

C.2.4 TPM2_EC_Ephemeral()

TPM2_EC_Ephemeral() is similar to TPM2_Commit() in that it uses the commit random value to generate an ephemeral key for use in a two-phase operation. However, TPM2_EC_Ephemeral() only used the

 Copyright © TCG 2006-2020

random value r to generate a corresponding public key $Q \coloneqq [r]G$ where G is the generator point for a specified curve.

As with TPM2_Commit(), a counter value is returned. This value needs to be used in a subsequent command in order to complete the two-phase operation.

C.2.5 Recovering the Private Ephemeral Key

To complete a split or two-phase operation, the TPM uses the same random or pseudo-random value generated in TPM2_Commit(). The random or pseudo-random value is determined by the *counter* field provided as an input parameter for the command that is the second phase of the split operation.

If the values are stored in an array, *counter* is used to index the array and, after the value is used in the signing operation, the value is erased. If using the pseudo-random method, the following algorithm is used to reconstruct the random value.

a) set $t \coloneqq$ low-order 16 bits of $commitCount$

b) verify that $t - 2^N < counter < t$; else return TPM_RC_RANGE

c) set $i \coloneqq$ low-order N bits of $counter$

d) if $A[i]$ is CLEAR, return TPM_RC_VALUE

e) set $c \coloneqq commitCount - t$

f) if $counter \geq t$; $c \coloneqq c - 2^{16}$

g) $c \coloneqq c + counter$

h) compute r as in equation (62) using c in place of $commitCount$

i) if the command completes successfully set $A[i] \coloneqq 0$

C.3 ECC-Based Secret Sharing

An ECC key protects a secret in two cases: object duplication and seeding of a session. In both cases, the method for generating the required key uses **KDFe**(), as described in 11.4.10.3.

C.4 EC Signing

C.4.1 ECDSA

For a TPM compliant with this specification, the default ECC signing scheme (DSA) is as defined ISO/IEC 14888-3.

C.4.2 ECDAA

C.4.2.1 Introduction

If a TPM supports ECC, it is recommended that it also support the ECDAA scheme described in this clause C.4.2.

Direct Anonymous Attestation based on ECC (ECDAA) is a TPM signature scheme that provides anonymous signatures (meaning that different signatures from the same signer cannot be correlated), or pseudonymous signatures (meaning that different signatures from the same signer can be correlated but

 Copyright © TCG 2006-2020

the identity of the signer is still unknown). Multiple ECDAA schemes are supported in this TPM implementation.

The TPM signs data with an ECDAA key in an unconventional way. A Verifier verifies signature values using data equivalent to a public key, and verifies the public key using data equivalent to a certificate (which is also called a credential) supplied by an Issuer. However, the public key and the credential are randomized by the TPM and the TPM's Host platform before they are sent to the Verifier. This prevents both the Verifier and the Issuer from identifying the TPM that created the signature value.

It is anticipated that the most common use of ECDAA will be to certify (TPM2_Certify()) a TPM object (usually a key). A credential issuer will provide a certificate for an ECDAA key. This certificate will validate that the ECDAA key belongs to a valid TPM without disclosing the ECDAA key. That ECDAA key may then be used to certify other TPM objects. These certificates prove that the certified object belongs to a valid TPM without disclosing the identity of that TPM. If the certified key is a signing key, it may then be used to attest to various TPM states, without disclosing the identity of the TPM to which it belongs.

This scheme is substantially different from the AIK scheme in 1.2 in that the ECDAA key may be used to provide the anonymity for keys rather than having to send each new attestation key to a privacy certificate authority (PCA) in order to have an anonymizing certificate produced. After a certificate has been obtained for an ECDAA key, it may be used to produce anonymized certificates for many TPM keys without requiring additional interaction with a privacy CA.

An ECDAA key may be used in any command that produces a signature. The TPM may not be used to verify an ECDAA signature.

C.4.2.2 ECDAA Key Generation on the TPM

While any signing key may be an ECDAA key, it is most useful as a Primary Key in the Endorsement hierarchy. This ensures that a TPM will normally produce the same ECDAA keys and receive the same credentials from a given Issuer, no matter how many times the credentialing process is performed, and no matter how many owners the TPM has had. This property is desirable because an Issuer should only give credentials to a platform after verifying that the platform has the architecture of a trusted platform. The Issuer would give replacement (different) credentials only when it is necessary to retire the old credentials. Replacement credentials erase the previous DAA history of the platform, at least as far as the credentials from that issuer are concerned. Replacement might be desirable, as when a platform changes hands, for example, in order to eliminate any association via DAA between the seller and the buyer. On the other hand, replacement might be undesirable, since it enables a rogue to rejoin a community from which it has been barred. Replacement is done by submitting a different TPMT_PUBLIC.*unique* field value to the TPM when the key is created (TPM2_Create() or TPM2_CreatePrimary()). Software may use any value of TPMT_PUBLIC.*unique* field at any time, in any order, but the Issuer can detect when a request uses a different value from the previous request and could reject the request.

The cryptographic parameters of the curve are indicated by the template in the command (TPM2_Create() or TPM2_CreatePrimary()) that creates the curve. The curve ID depends on the Issuer who is expected to provide a credential for the DAA key (different Issuers may require different curves). The TPM generates a private key (d_s) and a public key (Q_s). The non-cryptographic parameters in the template (that is, object attributes and signing scheme) are chosen by the entity that calls the command to create the DAA key. Inappropriate choice of the non-cryptographic parameters will cause the Issuer to reject an application for a DAA credential.

The security strength of an ECDAA key is the same as an ECC key of the same size. The key size is determined by the order of the curve (n) and the cofactor (h).

If the Endorsement Primary Seed is used as the DAA seed, then, like other EK, an ECDAA key will change whenever the EPS is changed.

 Copyright © TCG 2006-2020

The process for generating an ECDAA key is identical to the process used for any ECC key.

For the TCG defined ECDAA protocol, the curve described by p, n, and b is a Barreto-Naehrig (BN) elliptic curve. BN curves are of the form $y^2=x^3+b$ as defined in [ISO/IEC 15946-5 : 2008 Clause 7.3 “BN curve”], which is equivalent to [IEEE P1363.3 (Draft 2) Clause A.11.5 BN Curves].

NOTE The linear term (a) of generic ECC curves (curves with the form $y^2=x^3+ax+b$) is zero in BN curves. All BN curves are suitable but some are less efficient than others. The BN curves recommended in this version of DAA were chosen by the DAA designers.

The cryptographic value of the public key in the resultant TPM key structure is Q_s, which is used by the Issuer when computing the membership credential on the DAA private key d_s. Q_s is not used to verify the DAA signatures produced by the TPM and corresponding host platform.

C.4.2.3 ECDAA Sign Operation

The ECDAA scheme may be used in any command that uses a signing key. These are, the attestation group and TPM2_Sign().

For an attestation command using the ECDAA scheme, both the *qualifiedSigner* and *extraData* fields in the attestation block (a TPMS_ATTEST) are set to be the Empty Buffer before the data is hashed. The attestation data is then marshaled and hashed. The resulting hash data is then concatenated to the first hash to produce the value to sign (P).

$$P \coloneqq \mathbf{H}_{schemeHash}(qualifyingData \mid\mid \mathbf{H}_{schemeHash}(\text{TPMS_ATTEST})) \quad (63)$$

For TPM2_Sign(), the value to sign is the input *digest* and

$$P \coloneqq digest \quad (64)$$

To complete the ECDAA sign operation, the TPM uses the same random or pseudo-random value (r) used in TPM2_Commit(). The value is determined by the *counter* field in the scheme parameter of the signing command. This parameter is used in the process defined inC.2.5.

The signature is created using a modified Schnorr signature using the P and r values described above:

a) set k to a random value such that $0 < k < n$

b) compute $T \coloneqq \mathbf{H}(k \mid\mid P)(\text{mod } n)$

c) compute integer $s \coloneqq (r + Td_s)(\text{mod } n)$

d) if $s = 0$, output failure (negligible probability)

The signature is the tuple (k , s).

NOTE The k value is returned in the *R* parameter of the TPMT_SIGNATURE structure.

C.4.3 EC Schnorr

C.4.3.1 Introduction

If a TPM supports ECC, it should support the TPM_ALG_ECSCHNORR scheme.

The scheme description uses the following values:

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

G	generator point for the curve of the signing key
d_S	private value of the signing key
Q_S	public point of the signing key ($Q_S \coloneqq [d_S]G$)
n	order of G
$\mathbf{H}_{schemeHash}$	hash algorithm specified in the signing scheme

C.4.3.2 EC Schnorr Sign

An EC Schnorr signature is generated when the signing scheme for a key is TPM_ALG_ECSCHNORR. The scheme many be used in any signing operation

To sign a digest P

a) set k to a random value such that 0 < k < n

b) compute $E \coloneqq (x_E, y_E) \coloneqq [k]G$

c) if E is the point at infinity, go to a)

d) compute $r \coloneqq \mathbf{TRUNC}(\mathbf{H}_{schemeHash}(\mathbf{FE2BS}(x_E) \mid\mid P), n)$

NOTE 1 x_E is a field element with the same number of bits as the curve order n

NOTE 2 **TRUNC**() is a function that reduces the number of octets in the first argument until it has no more octets than the second argument. Truncation occurs from the less significant end of the number. If the digest produced by $\mathbf{H}_{schemeHash}$ has the same number of octets as the curve order n, then no truncation occurs.

NOTE 3 **FE2BS**() is a function that converts the number x_E (a field element) into a canonical value (octet or byte string) with the same number of octets as the field order n. This may result in a value with leading octets of zero. As x_E is computed (mod p) the value may be greater than n

e) compute integer $s \coloneqq (k + rd_S) \pmod{n}$

NOTE 4 This is the same computation as step c) in C.4.2.3.

f) if $s = 0$ or $s = k$ go to a)

NOTE 5 The $s = k$ check is to eliminate the possibility that $0 = r \pmod{n}$. Optionally, an implementation could check after d) that $0 \neq r \pmod{n}$.

The signature is the tuple (r, s).

C.4.3.3 EC Schnorr Signature Validate

To validate a Schnorr signature (r, s) over digest P

a) verify that 0 < s < n

b) compute $(x_E, y_E) \coloneqq [s]G + [-r]Q_S$

c) compute $r' \coloneqq \mathbf{TRUNC}(\mathbf{H}_{schemeHash}(\mathbf{FE2BS}(x_E) \mid\mid P), n)$

d) the signature is valid if $r' = r$

 Copyright © TCG 2006-2020

NOTE The comparison of r' and r is done assuming that both values are numeric and not octet strings. This reduces the chance of interoperability problems due to padding performed on r.

C.5 ECC Key Generation

For an ECC key, the method of FIPS 186-4, Annex B.4.1 *Key Pair Generation Using Extra Random* is used. The caller provides a random number generation function according to the type of key being generated (see 27.6.2, 27.6.3, and 28.4) and that function is called when random bits are required by the generation process. The key generation process is:

a) obtain a random value c of $\mathbf{length}(n)$ + 64 bits where n is the order of the curve
b) set $d \coloneqq (c \bmod (n\text{-}1)) + 1$
c) compute $Q \coloneqq (x_Q, y_Q) \coloneqq [d]G$
d) return d and Q

C.6 Secret Sharing

C.6.1 ECDH

For secret sharing with an ECC key, the One-Pass Diffie-Hellman, C(1, 1, ECC CDH) method from SP800-56A is used.

Using the notation of SP800-56A, the initiator generates an ephemeral key pair ($d_{e,U}$, $Q_{e,U}$) from the curve parameters. The public point of the ephemeral key($Q_{e,U}$) is used by the recipient to recover the shared secret.

The initiator uses the private portion of the ephemeral key ($d_{e,U}$) and the public portion ($Q_{s,V}$) of an ECC key of the recipient and computes the point $P \coloneqq h\ [d_{e,U}]Q_{s,V}$. Then it will set $Z \coloneqq x_P$ where x_P is the x-coordinate of P.

The recipient may compute $P \coloneqq h\ [d_{s,V}]Q_{e,U}$ and $Z \coloneqq x_P$.

The Z value is used in ***KDFe*** to generate a value for *seed* that is appropriate for the use of the seed. The seed will be the size of the digest produced by the *hashAlg* used in the KDF. Seed is computed by:

$$seed \coloneqq \mathbf{KDFe}(hashAlg, Z, label, PartyUInfo, PartyVInfo, bits) \quad (65)$$

where

$hashAlg$	the nameAlg of the recipient key
Z	the x coordinate (x_P) of the product (P) of a public point and a private key (P ≔ h [d] Q)
$label$	an application-dependent value
$PartyUInfo$	the x-coordinate of the secret exchange value ($Q_{e,U}$)
$PartyVInfo$	the x-coordinate of a public key ($Q_{s,V}$)
$bits$	the number of bits in the digest of hashAlg

 Copyright © TCG 2006-2020

C.6.2 ECDH Encryption of Salt

In TPM2_StartAuthSession(), when *tpmKey* is an ECC key, a seed value is produced as described in C.6.1 with the *label* parameter set to "SECRET". This seed value is then used as the session secret.

C.6.3 ECC Secret Sharing for Duplication

When the new parent for a duplicated object is an ECC key, an ephemeral key is created and used to generate a seed value as described in C.6.1. When creating the seed, the label parameter is set to "DUPLICATE". The seed value is then used to generate the encryption and integrity values for the duplication blob as described in clause 22.

C.6.4 ECC Secret Sharing for Credentials

When the decryption key for an identity blob is an ECC key, an ephemeral key is created and used to generate a seed value as described in C.6.1. When creating the seed, the label parameter is set to "IDENTITY". The seed value is then used to generate the encryption and integrity values for the identity blob as described in clause 22.

C.7 ECC Primitive Operations

C.7.1 Introduction

When ECC is implemented on a TPM, it may provide these additional commands to support cryptographic operations with unrestricted ECC keys.

C.7.2 TPM2_ECDH_KeyGen()

TPM2_ECDH_KeyGen produces an ephemeral key pair. It multiplies the private ephemeral key with the public point of a loaded TPM key to produce the Diffie-Hellman shared secret.

This function can be performed by software as the public key and parameters are known. The function would be provided by the TPM as a service.

Since the operation can be performed by software, no authorization is required to use the public portion of the key and the key attributes are not checked.

C.7.3 TPM2_ECDH_ZGen()

TPM2_ECDH_ZGen performs the ECDH primitive function with one static and one ephemeral key as defined in SP800-56A, clause 6.2.2. The input point (Q_e) is multiplied by the private coordinate (d_s) to produce the point $Z = (x_Z, y_Z) \coloneqq hd_sQ_e$.

Since this operation used the private portion of an ECC key, authorization is required. To prevent inadvertent compromise of a signing key, *sign* and restricted are required to be CLEAR in the referenced key.

 Copyright © TCG 2006-2020

C.7.4 Two-phase Key Exchange

C.7.4.1 Introduction

Various key exchange protocols use an ephemeral key from each party. For these protocols, each party generates an ephemeral key and that key is sent to the other party along with other information. The other party then uses the key material from the other party along with its own ephemeral key to generate the key-exchange values.

These protocols require two phases. In the first phase, the TPM generates an ephemeral key to be sent to the other party. In the second phase, the TPM combines data from the other party with the ephemeral key generated in the first phase. The protocols require that the ephemeral key generated by the TPM only be used once and be discarded after the key exchange is complete. This property of this key is the same as required for ECDAA.

TPM2_EC_Ephemeral() uses the commit mechanism to generate a random value (r) and a public key $P \coloneqq [r]G$. The value of P is returned to the caller along with the counter value associated with r.

TPM2_ZGen_2Phase() is used to complete the second phase of the key exchange. The counter value returned by TPM2_EC_Ephemeral() is provided from which the TPM recreates r and regenerates the associated public key. When TPM2_ZGen_2Phase() completes successfully, the TPM will "retire" the r value so that it may not be used again.

One of the parameters of TPM2_ZGen_2Phase() is a scheme selector (*inScheme*). This indicates to the TPM which of the supported schemes is to be used. This annex describes two of the allowed schemes. They are the two EC schemes from SP800-56A that require two ephemeral and two static keys. The schemes are described in SP800-56A in *6.1.1.2 Full Unified Model, C(2, 2, ECC CDH)* and *6.1.1.4 Full MQV, C(2, 2, ECC MQV).* These schemes use the following terms:

$d_{s,A}$	the private part of a TPM-resident ECC key referenced by the *keyA* parameter
$Q_{s,A}$	the public point of the key referenced by *keyA* equal to $[d_{s,A}]G$ with coordinates $(x_{s,A}, y_{s,A})$
$d_{e,A}$	a private ephemeral key generated by the TPM (the value of r associated with *counter* parameter)
$Q_{e,A}$	the public ephemeral key associated with *counter* equal to $[d_{e,A}]G$ or $[r]G$ with coordinates $(x_{e,A}, y_{e,A})$
$Q_{s,B}$	the *inQsB* parameter – a point on the curve of *keyA* assumed to be a static public key associated with the other party in the key exchange with coordinates $(x_{s,B}, y_{s,B})$
$Q_{e,B}$	the *inQeB* parameter – a point on the curve of *keyA* assumed to be an ephemeral public key associated with the other party in the key exchange with coordinates $(x_{e,B}, y_{e,B})$

C.7.4.2 Full Unified Model

When this scheme is selected for TPM2_ZGen_2Phase(), the TPM will:

a) set *outZ1* $\coloneqq [d_{s,A}]Q_{s,B}$

b) set *outZ2* $\coloneqq [d_{e,A}]Q_{e,B}$

 Copyright © TCG 2006-2020

NOTE If outZ1 or outZ2 is the point at infinity, then both coordinate values of the point will be Empty Buffers.

C.7.4.3 Full MQV

This scheme uses an associated value function (**avf**()) that is defines as:

Inputs:

$Q = (x, y)$	a public key
n	the modulus of the curve containing Q

Process:

Process:

a) Set $f \coloneqq \lceil(\lceil \log_2(n)\rceil / 2)\rceil$

b) Set $x' = 2^f + (x \bmod 2^f)$

c) return x'

The MQV computation is:

d) validate that $Q_{s,B}$ and $Q_{e,B}$ are on the curve associated with $d_{s,A}$

e) using *counter*, recover $d_{e,A} = r$ as described in C.2.5

f) set $Q_{e,A} \coloneqq [d_{e,A}]G$ where G is the generator point for the curve of $d_{s,A}$

g) set $t_A \coloneqq (d_{e,A} + d_{s,A} \cdot \mathbf{avf}(Q_{e,A}))$ (mod n)

h) set *outZ1* $\coloneqq [h \cdot t_A]\,(Q_{e,B} + [\mathbf{avf}(Q_{e,B})](Q_{s,B}))$

NOTE 1 if *outZ1* is the point at infinity both the coordinate values of outZ1 will be Empty Buffers

NOTE 2 This protocol may be susceptible to unknown key-share (UKS) attacks.

C.8 ECC Point Padding

To provide consistent behavior across all TPM implementations this clause specifies the padding requirements for ECC parameters. An ECC point returned by the TPM is composed of x and y ECC parameters, both of which are required to be the size of their associated curve (e.g., 32 bytes for NIST P-256). If necessary, leading bytes of zero will be added to these point values. When the ECC parameters are returned by the command TPM2_ECC_Parameters(), the numeric values will be as specified in the TCG Algorithm registry. However, the size of values, other than the *x* and *y* of the generator point, may not be the same as the registry because values may or may not be zero padded.

To ensure interoperability, an ECC point that is part of a TPM2B_PUBLIC should be padded with zeros to be the size of the order of the curve that defines the key.

When an ECC point is input to the TPM, the padding is required as described above for TPM output of an ECC point.

NOTE: The reason for requiring padding on the input of an ECC point is that an ECC point makes up the *unique* field of an ECC key. The Name of the key is computed by hashing the key's public area as input to the TPM. If the ECC point in the *unique* field is not properly padded, the Name would not be consistent.

 Copyright © TCG 2006-2020

An intermediate ECC point, such as the result of an ECC point multiplication or the public key of an ephemeral ECC key, is required to be padded if used as input to the KDFe() function.

NOTE This ensures that the secret derived from KDFe(), which is used e.g. as salt in TPM2_StartAuthSession(), or as protection seed of the outer wrapper in TPM2_Duplicate(), is the same on different implementations.

Revision 1.16 of this specification also required the ECC private key in *duplicate* of TPM2_Import() to be padded.

 Copyright © TCG 2006-2020

Annex D
(normative/informative)
Support for SMx Family of Algorithms

D.1 Introduction

This section provides additional information for implementation of the SM2, SM3, and SM4 algorithms published by State Encryption Management Bureau, China.

D.2 SM2

D.2.1 Introduction

SM2 is contains information relating to ECC cryptography and is in five parts.

- Part 1: General – "provides necessary basics of mathematics and related cryptographic techniques used in public key cryptographic algorithm SM2 based on elliptic curves." The methods of this part are compatible with the EC methods in other standards and no special considerations are necessary to accommodate this standard

[GM/T 0003.1-2012 *Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves, part1: General Protocol*, published by State Encryption Management Bureau, China]

- Part 2: *Digital Signature Algorithm* – defines the process for generation and verification of a digital signature using the methods described in Part 1. The signing method in this part of the standard require addition of a new signing scheme and methods. These are described in this annex. .

[GM/T 0003.2-2012 *Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves, part2: Digital Signature Algorithm*, published by State Encryption Management Bureau, China]

- Part 3: *Key Exchange Protocol* – defines a two phase key exchange protocol using the methods of Part 1. The method in this part of the SM2 standard is supported by addition of a key exchange command (TPM2_ZGen_2Phase()). The algorithm is fully described in TPM 2.0 Part 3 of this TPM specification.

[GM/T 0003.3-2012 *Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves, part3: Key Exchange Protocol*, published by State Encryption Management Bureau, China]

- Part 4: Public Key Encryption Algorithm – defines an encryption method using single pass EC Diffie-Hellman to exchange a key that is then used to generate a stream cipher. The TPM does not use this method.

[GM/T 0003.4-2012 *Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves, part4: Public Key Encryption Algorithm*, published by State Encryption Management Bureau, China.]

- Part5: Parameter definition – defines the parameters for a 256-bit ECC curve.

[GM/T 0003.5-2012 *Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves, part5: Parameter definition*, published by State Encryption Management Bureau, China]

 Copyright © TCG 2006-2020

D.2.2 SM2 Digital Signature Algorithm

D.2.2.1 SM2 Sign

The SM2 signing scheme has an algorithm ID of TPM_ALG_SM2. If the TPM implements this algorithm, then any structure that allows an ECC-based signing scheme may use this algorithm ID.

The TPM only implements a portion of the full SM2 signing scheme. That portion is the part that uses the private key to sign a digest.

The inputs to the algorithm are:

e	a digest to sign
d_s	a private ECC key
n	the modulus of the curve for d_s

The computation implemented in the TPM is:

a) set k to a random value such that $1 \leq k \leq n$-1

b) compute $P_1 \coloneqq (x_1, y_1) \coloneqq [k]G$

c) compute $r \coloneqq e + x_1$ (mod n)

d) if r equals 0 or ($r + k$) equals n, go to 0

e) compute $s \coloneqq ((1 + d_s)^{-1} \cdot (k - r \cdot d_s))$ (mod n)

f) if s equals 0, go to 0

g) the signature is the tuple (r, s)

D.2.2.2 SM2 Signature Verification

For verification (TPM2_VerifySignature() and TPM2_PolicySigned()), the inputs are:

e	the digest that was signed
(r, s)	the signature tuple
P	a public ECC key
G	the generator point for the curve of P
n	the modulus of the curve for d_s

a) The verification computation performed by the TPM is:

b) verify that r and s are in the inclusive interval 1 to (n – 1)

c) compute $t \coloneqq (r + s)$ (mod n)

d) verify that $0 < t$

e) compute $(x, y) \coloneqq [s]G + [t]P$

f) compute $r' \coloneqq (e + x)$ (mod n)

g) verify that $r' = r$

If any of the verification steps fails, then the signature is not valid.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

D.2.2.3 Implementation Issues

In the SM2 standard, the message to sign is combined with key-specific data to produce an e value that is signed using the algorithm shown above. The computation for e uses a value Z_A that, according to the SM2 standard, is computed by:

$$Z_A \coloneqq \mathbf{H}(\ ENTL_A \,||\, ID_A \,||\, a \,||\, b \,||\, x_G \,||\, y_G \,||\, x_A \,||\, y_A\ ) \tag{66}$$

where

$ENTL_A$	two octets containing the length of ID_A in octets
ID_A	octet string containing information that can identify an entity's identity unambiguously (see ISO/IEC 15946-3 3.9)
a	coefficient for the linear term of the equation for the curve of the signing key
b	coefficient for the constant term of the equation for the curve of the signing key
x_G	the x coordinate of the generator point for the curve of the signing key
y_G	the x coordinate of the generator point for the curve of the signing key
x_A	the x coordinate of the public key of the signing key
y_A	the y coordinate of the public key of the signing key

Using Z_A and a message (M) the digest to sign (e) is computed by:

$$e \coloneqq \mathbf{H}(Z_A \,||\, M) \tag{67}$$

Since the TPM does not do the operation in equation (67), the caller may need to modify the input message before using the TPM to sign the digest. If the application requires it, the caller would need to do the computation of e before giving the value to the TPM to sign.

One consequence of this is that attestation operations will not create a signature that is in all details, compliant with SM2 Part 2. Instead, the attestation signatures will be TPM specific. The reason that attestations do not sign using the full scheme are:

- There is no infrastructure for the distribution of ID_A values
- Requiring the use of an ID_A value in a signature could allow correlation of a user and void the privacy assurances of the attestation
- Ensuring that an external digest does not match a valid attestation becomes intractable.

The reason that the attestation problem becomes intractable is that, using Z_A with an attestation means that the first bytes that were used to form the digest of the signed value (e) would vary with each key used to sign. An attacker could perform a hash using the key specific values followed by message data that has all the characteristics of an attestation. The TPM will not be able to discern the transition from Z_A data to the false attestation data.

To prevent this kind of attack without adding excessive complexity to the TPM, the attestation is done without including Z_A. Since the use of Z_A does not improve the security of the SM2 signature, leaving it out does not compromise the value of the SM2 signing process for attestations. Also, since an attestation only has meaning in the context of a TPM, having TPM-specific verification of a signature over an attestation block should not create an issue.

 Copyright © TCG 2006-2020

TPM2_Sign() may be used with the TPM_ALG_SM2 scheme identifier to create a full SM2-compatible signature. To do an SM2 signature, the application would compute Z_A, and then use the resulting digest as the first data in one of the TPM hash commands (which could be a TPM2_HashSequenceStart()); with the Z_A value followed by the message data (M). The digest of $\mathbf{H}(Z_A \| M)$ would then be used as the *digest* parameter for TPM2_Sign().

NOTE Since Z_A is a constant value for a key, an application might choose to keep Z_A as part of the meta-data for the key so that it would not need to be recomputed each time the key is used for an SM2 signature.

D.2.3 SM2 Key Exchange

D.2.3.1 Introduction

The key exchange algorithm in GM/T 0003.3-2012 is a two-phase algorithm. It is similar to the scheme described in C.7.4.3.

NOTE This protocol may be susceptible to unknown key-share (UKS) attacks.

This SM2 key exchange computations use an associated value function (**avfSM2**()) that is similar to the function defined in SP800-56A with the only differencing being that the result is one bit less than the value defined in SP800-56A. The **avfSM2**() function is:

Inputs:

$Q = (x, y)$	a public key
n	the modulus of the curve containing Q

Process:

a) set $f := \lceil (\lceil \log_2(n) \rceil / 2) \rceil - 1$

b) set $x' := 2^f + (x \pmod{2^f})$

c) return x'

NOTE This function is similar to the function in SP800-56A except that, in the formulation in GM/T 0002-2012 as shown in a) above, the value of f is one less than the equivalent in SP800-56A.

D.2.3.2 SM2 Key Exchange Protocol

The key exchange protocol is between two entities, A and B. The TPM performs computations as party A. Since the protocol is symmetric, both party A and party B may be TPMs and they will both perform the same operations, using the values from the other TPM as party B values.

The caller must use TPM2_EC_Ephemeral() to have the TPM generate a single-use ephemeral key. The ephemeral public key is sent to the other party as $Q_{e,B}$.

The inputs to the key exchange computation are:

$counter$	the counter parameter from TPM2_Commit()
$Q_{s,B}$	a public EC key from party B; usually, the public part of a static key
$Q_{e,B}$	a public EC key; usually, the public part of an ephemeral key
$d_{s,A}$	a private EC key (an unrestricted decryption key)

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

The protocol:

a) validate that $Q_{s,B}$ and $Q_{e,B}$ are on the curve associated with $d_{s,A}$
b) using *counter*, recover *r* as described in C.2.5
c) set $Q_{e,A} \coloneqq [r]G$ where G is the generator point for the curve of $d_{s,A}$
d) set $t_A \coloneqq (d_{s,A} + d_{e,A} \cdot \mathbf{avfSM2}(Q_{e,A}))$ (mod n)
e) set $Z \coloneqq [h \cdot t_A]\ (Q_{s,B} + [\mathbf{avfSM2}(Q_{e,B})](Q_{e,B}))$
f) if Z is the point at infinity, return failure

D.3 SM3

[GM/T 0004-2012 *Cryptographic Hash Algorithm SM3*, published by State Encryption Management Bureau, China]

SM3 is a hash algorithm that uses a 512-bit block and produces a digest of 256 bits.

If the TPM implements this algorithm, then the algorithm ID for SM3 (TPM_ALG_SM3_256) may be used in any structure that allows a hash algorithm.

D.4 SM4

[GM/T 0002-2012 *Block Cipher Algorithm SM4*, published by State Encryption Management Bureau, China]

SM4 is a symmetric block cipher with a key and block size of 128 bits.

If the TPM implements this algorithm, then the algorithm ID for SM4 (TPM_ALG_SM4) may be used in any structure that allows a symmetric block cipher.

 Copyright © TCG 2006-2020

Annex E
(normative/informative)
TDES

E.1 TDES Key Parity Generation

A TDES key is generated by getting 24 bytes from the random number generator appropriate for the type of key generation (such as a KDF for a derived key). The 24 bytes are treated as 3, 64-bit values in canonical TPM form (big-endian bytes). The odd parity is then generated for each byte with the parity replacing the least significant bit in each byte to create 3 DES keys. The resulting three DES keys are then validated to make sure that none of them is on the list of prohibited DES key values. If any of the generated key values is prohibited, then the TPM will repeat the key generating process by generating 24 new bytes.

 Copyright © TCG 2006-2020

Annex F
(informative)
Library Profile Guide

F.1 Introduction

This annex provides guidance to TPM platform specific work groups when developing platform specific TPM specifications. The platform specific specification must specify these items. It aggregates platform specific information from other parts of this specification.

F.2 Platform Specific Constants

- Constants returned by TPM2_GetCapability with the capability prefix TPM_PT_PS. See Part 2.
- The manufacturer, vendor strings, and firmware version.

F.3 PCR

- Number of PCR.
- The minimum size of a selection structure.
- Number of banks.
- Supported hash algorithms.
- PCR authorization, authorization groups, and locality.
- Which can be reset, under what conditions, and what the reset value is.
- Whether the PCR is preserved on resume.
- Which PCR increment the PCR update counter.
- DRTM and H-CRTM behavior, and PCR values at _TPM2_Init().

F.4 Algorithms

Define algorithms. See the TCG algorithm registry.

- Hash, asymmetric, and symmetric algorithms.
- For elliptic curve, the curves.
- Key sizes.
- Padding modes.
- Endorsement key certificate provisioning, and the NV attributes for the certificates.
- Algorithms and modes for parameter encryption.

F.5 Commands

See Part 2 and note that some commands have prerequisites or are implemented as a set.

- Mandatory, optional, and forbidden commands.
- Whether firmware upgrade is required, and whether the library specification or a vendor specific method is permitted.

 Copyright © TCG 2006-2020

F.6 Buffers

- The minimum for the maximum size of an NV Index.
- The minimum size for the input and output buffers.

F.7 NV Storage

- Specify as much as possible the NV storage capacity. Since NV Indexes has varying meta-data requirements, the value may not be exact.
- The types of NV Indexes supported. This is linked to the supported TPM2_NV commands.
- Whether NV Indexes and persistent keys may or may not come from the same NV memory pool.
- A minimum number of total Indexes, and minimums for certain Index types, such as counters.
- Whether both orderly and non-orderly indexes are supported.
- The minimum number of persistent key slots.

F.8 Sessions and Objects

- The minimum number of loaded and active sessions.
- The minimum number of loaded objects.

F.9 Physical Presence

- If physical presence is implemented, specify the table of commands that require physical presence.

F.10 Dictionary Attack Lockout

- The default value for maxTries and recoveryTime.

F.11 Self Test

- Whether TPM2_SelfTest() can be blocking or non-blocking.

F.12 ACT

- The number of supported ACT instances (usually 0 or 1).
- Trigger event when the ACT times out.
- Whether TPM2_ClockRateAdjust() may affect the ACT rate, and maximum adjustment.
- The ACT behavior if TPM is in a low power state (sleep mode) (typically, ACT must advance).
- Whether the ACT state must be preserved over a power cycle (setting of the *preserveSignaled* attribute).
- Whether clearing or setting the *signaled* attribute must also clear or set the associated trigger event (e.g. when ACT signaling is turned off, or ACT signaling is preserved across TPM Resume).

 Copyright © TCG 2006-2020

- Whether the remaining ACT timeout must be retrievable in TPM Failure Mode.

Trusted Platform Module Library

Part 2: Structures

Family “2.0”

Level 00 Revision 01.59

November 8, 2019

Published

Contact: admin@trustedcomputinggroup.org

TCG Published

Copyright © TCG 2006-2020

Licenses and Notices

Copyright Licenses:

- Trusted Computing Group (TCG) grants to the user of the source code in this specification (the "Source Code") a worldwide, irrevocable, nonexclusive, royalty free, copyright license to reproduce, create derivative works, distribute, display and perform the Source Code and derivative works thereof, and to grant others the rights granted herein.
- The TCG grants to the user of the other parts of the specification (other than the Source Code) the rights to reproduce, distribute, display, and perform the specification solely for the purpose of developing products based on such documents.

Source Code Distribution Conditions:

- Redistributions of Source Code must retain the above copyright licenses, this list of conditions and the following disclaimers.
- Redistributions in binary form must reproduce the above copyright licenses, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.

Disclaimers:

- THE COPYRIGHT LICENSES SET FORTH ABOVE DO NOT REPRESENT ANY FORM OF LICENSE OR WAIVER, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, WITH RESPECT TO PATENT RIGHTS HELD BY TCG MEMBERS (OR OTHER THIRD PARTIES) THAT MAY BE NECESSARY TO IMPLEMENT THIS SPECIFICATION OR OTHERWISE. Contact TCG Administration (admin@trustedcomputinggroup.org) for information on specification licensing rights available through TCG membership agreements.
- THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO EXPRESS OR IMPLIED WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, ACCURACY, COMPLETENESS, OR NONINFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.
- Without limitation, TCG and its members and licensors disclaim all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification and to the implementation of this specification, and TCG disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this specification or any information herein.

Any marks and brands contained herein are the property of their respective owners.

CONTENTS

1 Scope ... 1
2 Terms and definitions ... 1
3 Symbols and abbreviated terms ... 1
4 Notation ... 1
4.1 Introduction ... 1
4.2 Named Constants ... 2
4.3 Data Type Aliases (typedefs) ... 3
4.4 Enumerations ... 3
4.5 Interface Type ... 4
4.6 Arrays ... 5
4.7 Structure Definitions ... 6
4.8 Conditional Types ... 7
4.9 Unions ... 9
4.9.1 Introduction ... 9
4.9.2 Union Definition ... 9
4.9.3 Union Instance ... 10
4.9.4 Union Selector Definition ... 11
4.10 Bit Field Definitions ... 12
4.11 Parameter Limits ... 13
4.12 Algorithm Macros ... 14
4.12.1 Introduction ... 14
4.12.2 Algorithm Token Semantics ... 15
4.12.3 Algorithm Tokens in Unions ... 15
4.12.4 Algorithm Tokens in Interface Types ... 16
4.12.5 Algorithm Tokens for Table Replication ... 16
4.13 Size Checking ... 18
4.14 Data Direction ... 18
4.15 Structure Validations ... 20
4.16 Name Prefix Convention ... 20
4.17 Data Alignment ... 21
4.18 Parameter Unmarshaling Errors ... 21
5 Base Types ... 23
5.1 Primitive Types ... 23
5.2 Specification Logic Value Constants ... 23
5.3 Miscellaneous Types ... 24
6 Constants ... 25
6.1 TPM_SPEC (Specification Version Values) ... 25
6.2 TPM_GENERATED ... 25
6.3 TPM_ALG_ID ... 26
6.4 TPM_ECC_CURVE ... 30
6.5 TPM_CC (Command Codes) ... 30
6.5.1 Format ... 30
6.5.2 TPM_CC Listing ... 31
6.6 TPM_RC (Response Codes) ... 35
6.6.1 Description ... 35
6.6.2 Response Code Formats ... 35
6.6.3 TPM_RC Values ... 38
6.7 TPM_CLOCK_ADJUST ... 43
6.8 TPM_EO (EA Arithmetic Operands) ... 43
6.9 TPM_ST (Structure Tags) ... 44
6.10 TPM_SU (Startup Type) ... 46

 Copyright © TCG 2006-2020

6.11 TPM_SE (Session Type) ... 47
6.12 TPM_CAP (Capabilities) ... 48
6.13 TPM_PT (Property Tag) ... 49
6.14 TPM_PT_PCR (PCR Property Tag) ... 54
6.15 TPM_PS (Platform Specific) ... 56

7 Handles ... 57
7.1 Introduction ... 57
7.2 TPM_HT (Handle Types) ... 57
7.3 Persistent Handle Sub-ranges ... 58
7.4 TPM_RH (Permanent Handles) ... 59
7.5 TPM_HC (Handle Value Constants) ... 60

8 Attribute Structures ... 63
8.1 Description ... 63
8.2 TPMA_ALGORITHM ... 63
8.3 TPMA_OBJECT (Object Attributes) ... 64
8.3.1 Introduction ... 64
8.3.2 Structure Definition ... 64
8.3.3 Attribute Descriptions ... 65
8.3.3.1 Introduction ... 65
8.3.3.2 Bit[1] – *fixedTPM* ... 66
8.3.3.3 Bit[2] – *stClear* ... 66
8.3.3.4 Bit[4] – *fixedParent* ... 66
8.3.3.5 Bit[5] – *sensitiveDataOrigin* ... 66
8.3.3.6 Bit[6] – *userWithAuth* ... 67
8.3.3.7 Bit[7] – *adminWithPolicy* ... 67
8.3.3.8 Bit[10] – *noDA* ... 68
8.3.3.9 Bit[11] – *encryptedDuplication* ... 68
8.3.3.10 Bit[16] – *restricted* ... 69
8.3.3.11 Bit[17] – *decrypt* ... 69
8.3.3.12 Bit[18] – *sign* / *encrypt* ... 70
8.3.3.13 Bit[19] – *x509sign* ... 70
8.4 TPMA_SESSION (Session Attributes) ... 71
8.5 TPMA_LOCALITY (Locality Attribute) ... 73
8.6 TPMA_PERMANENT ... 74
8.7 TPMA_STARTUP_CLEAR ... 75
8.8 TPMA_MEMORY ... 76
8.9 TPMA_CC (Command Code Attributes) ... 77
8.9.1 Introduction ... 77
8.9.2 Structure Definition ... 77
8.9.3 Field Descriptions ... 77
8.9.3.1 Bits[15:0] – *commandIndex* ... 77
8.9.3.2 Bit[22] – *nv* ... 77
8.9.3.3 Bit[23] – *extensive* ... 78
8.9.3.4 Bit[24] – *flushed* ... 78
8.9.3.5 Bits[27:25] – *cHandles* ... 78
8.9.3.6 Bit[28] – *rHandle* ... 78
8.9.3.7 Bit[29] – V ... 78
8.9.3.8 Bits[31:30] – Res ... 79
8.10 TPMA_MODES ... 79
8.11 TPMA_X509_KEY_USAGE ... 80
8.12 TPMA_ACT ... 81

9 Interface Types ... 82
9.1 Introduction ... 82
9.2 TPMI_YES_NO ... 82

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

9.3 TPMI_DH_OBJECT 82
9.4 TPMI_DH_PARENT 83
9.5 TPMI_DH_PERSISTENT 83
9.6 TPMI_DH_ENTITY 84
9.7 TPMI_DH_PCR 84
9.8 TPMI_SH_AUTH_SESSION 85
9.9 TPMI_SH_HMAC 85
9.10 TPMI_SH_POLICY 85
9.11 TPMI_DH_CONTEXT 85
9.12 TPMI_DH_SAVED 86
9.13 TPMI_RH_HIERARCHY 86
9.14 TPMI_RH_ENABLES 86
9.15 TPMI_RH_HIERARCHY_AUTH 87
9.16 TPMI_RH_HIERARCHY_POLICY 87
9.17 TPMI_RH_PLATFORM 87
9.18 TPMI_RH_OWNER 88
9.19 TPMI_RH_ENDORSEMENT 88
9.20 TPMI_RH_PROVISION 88
9.21 TPMI_RH_CLEAR 89
9.22 TPMI_RH_NV_AUTH 89
9.23 TPMI_RH_LOCKOUT 89
9.24 TPMI_RH_NV_INDEX 90
9.25 TPMI_RH_AC 90
9.26 TPMI_RH_ACT 91
9.27 TPMI_ALG_HASH 91
9.28 TPMI_ALG_ASYM (Asymmetric Algorithms) 91
9.29 TPMI_ALG_SYM (Symmetric Algorithms) 92
9.30 TPMI_ALG_SYM_OBJECT 92
9.31 TPMI_ALG_SYM_MODE 92
9.32 TPMI_ALG_KDF (Key and Mask Generation Functions) 93
9.33 TPMI_ALG_SIG_SCHEME 93
9.34 TPMI_ECC_KEY_EXCHANGE 93
9.35 TPMI_ST_COMMAND_TAG 94
9.36 TPMI_ALG_MAC_SCHEME 94
9.37 TPMI_ALG_CIPHER_MODE 94

10 Structure Definitions 95

10.1 TPMS_EMPTY 95
10.2 TPMS_ALGORITHM_DESCRIPTION 95
10.3 Hash/Digest Structures 95

10.3.1 TPMU_HA (Hash) 95
10.3.2 TPMT_HA 96

10.4 Sized Buffers 96

10.4.1 Introduction 96
10.4.2 TPM2B_DIGEST 97
10.4.3 TPM2B_DATA 97
10.4.4 TPM2B_NONCE 97
10.4.5 TPM2B_AUTH 97
10.4.6 TPM2B_OPERAND 98
10.4.7 TPM2B_EVENT 98
10.4.8 TPM2B_MAX_BUFFER 98
10.4.9 TPM2B_MAX_NV_BUFFER 98
10.4.10 TPM2B_TIMEOUT 99
10.4.11 TPM2B_IV 99

10.5 Names 99

10.5.1 Introduction 99
10.5.2 TPMU_NAME 99

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.5.3 TPM2B_NAME 100
10.6 PCR Structures 100
10.6.1 TPMS_PCR_SELECT 100
10.6.2 TPMS_PCR_SELECTION 101
10.7 Tickets 101
10.7.1 Introduction 101
10.7.2 A NULL Ticket 102
10.7.3 TPMT_TK_CREATION 102
10.7.4 TPMT_TK_VERIFIED 103
10.7.5 TPMT_TK_AUTH 104
10.7.6 TPMT_TK_HASHCHECK 105
10.8 Property Structures 105
10.8.1 TPMS_ALG_PROPERTY 105
10.8.2 TPMS_TAGGED_PROPERTY 105
10.8.3 TPMS_TAGGED_PCR_SELECT 106
10.8.4 TPMS_TAGGED_POLICY 106
10.8.5 TPMS_ACT_DATA 106
10.9 Lists 107
10.9.1 TPML_CC 107
10.9.2 TPML_CCA 107
10.9.3 TPML_ALG 107
10.9.4 TPML_HANDLE 108
10.9.5 TPML_DIGEST 108
10.9.6 TPML_DIGEST_VALUES 109
10.9.7 TPML_PCR_SELECTION 109
10.9.8 TPML_ALG_PROPERTY 109
10.9.9 TPML_TAGGED_TPM_PROPERTY 110
10.9.10 TPML_TAGGED_PCR_PROPERTY 110
10.9.11 TPML_ECC_CURVE 110
10.9.12 TPML_TAGGED_POLICY 111
10.9.13 TPML_ACT_DATA 111
10.10 Capabilities Structures 111
10.10.1 TPMU_CAPABILITIES 112
10.10.2 TPMS_CAPABILITY_DATA 112
10.11 Clock/Counter Structures 113
10.11.1 TPMS_CLOCK_INFO 113
10.11.2 *Clock* 114
10.11.3 *ResetCount* 114
10.11.4 *RestartCount* 114
10.11.5 *Safe* 114
10.11.6 TPMS_TIME_INFO 114
10.12 TPM Attestation Structures 115
10.12.1 Introduction 115
10.12.2 TPMS_TIME_ATTEST_INFO 115
10.12.3 TPMS_CERTIFY_INFO 115
10.12.4 TPMS_QUOTE_INFO 115
10.12.5 TPMS_COMMAND_AUDIT_INFO 116
10.12.6 TPMS_SESSION_AUDIT_INFO 116
10.12.7 TPMS_CREATION_INFO 116
10.12.8 TPMS_NV_CERTIFY_INFO 116
10.12.9 TPMS_NV_DIGEST_CERTIFY_INFO 117
10.12.10 TPMI_ST_ATTEST 117
10.12.11 TPMU_ATTEST 117
10.12.12 TPMS_ATTEST 118
10.12.13 TPM2B_ATTEST 119

 Copyright © TCG 2006-2020

10.13 Authorization Structures 119
10.13.1 Introduction 119
10.13.2 TPMS_AUTH_COMMAND 119
10.13.3 TPMS_AUTH_RESPONSE 119
11 Algorithm Parameters and Structures 120
11.1 Symmetric 120
11.1.1 Introduction 120
11.1.2 TPMI_!ALG.S_KEY_BITS 120
11.1.3 TPMU_SYM_KEY_BITS 120
11.1.4 TPMU_SYM_MODE 121
11.1.5 TPMU_SYM_DETAILS 121
11.1.6 TPMT_SYM_DEF 122
11.1.7 TPMT_SYM_DEF_OBJECT 122
11.1.8 TPM2B_SYM_KEY 122
11.1.9 TPMS_SYMCIPHER_PARMS 123
11.1.10 TPM2B_LABEL 123
11.1.11 TPMS_DERIVE 123
11.1.12 TPM2B_DERIVE 124
11.1.13 TPMU_SENSITIVE_CREATE 124
11.1.14 TPM2B_SENSITIVE_DATA 124
11.1.15 TPMS_SENSITIVE_CREATE 125
11.1.16 TPM2B_SENSITIVE_CREATE 125
11.1.17 TPMS_SCHEME_HASH 125
11.1.18 TPMS_SCHEME_ECDAA 126
11.1.19 TPMI_ALG_KEYEDHASH_SCHEME 126
11.1.20 HMAC_SIG_SCHEME 126
11.1.21 TPMS_SCHEME_XOR 126
11.1.22 TPMU_SCHEME_KEYEDHASH 127
11.1.23 TPMT_KEYEDHASH_SCHEME 127
11.2 Asymmetric 128
11.2.1 Signing Schemes 128
11.2.1.1 Introduction 128
11.2.1.2 RSA Signature Schemes 128
11.2.1.3 ECC Signature Schemes 128
11.2.1.4 TPMU_SIG_SCHEME 129
11.2.1.5 TPMT_SIG_SCHEME 129
11.2.2 Encryption Schemes 129
11.2.2.1 Introduction 129
11.2.2.2 RSA Encryption Schemes 129
11.2.2.3 ECC Key Exchange Schemes 130
11.2.3 Key Derivation Schemes 130
11.2.3.1 Introduction 130
11.2.3.2 TPMU_KDF_SCHEME 130
11.2.3.3 TPMT_KDF_SCHEME 130
11.2.3.4 TPMI_ALG_ASYM_SCHEME 131
11.2.3.5 TPMU_ASYM_SCHEME 131
11.2.3.6 TPMT_ASYM_SCHEME 131
11.2.4 RSA 132
11.2.4.1 TPMI_ALG_RSA_SCHEME 132
11.2.4.2 TPMT_RSA_SCHEME 132
11.2.4.3 TPMI_ALG_RSA_DECRYPT 132
11.2.4.4 TPMT_RSA_DECRYPT 132
11.2.4.5 TPM2B_PUBLIC_KEY_RSA 133
11.2.4.6 TPMI_RSA_KEY_BITS 133
11.2.4.7 TPM2B_PRIVATE_KEY_RSA 133

 Copyright © TCG 2006-2020

11.2.5 ECC ... 134
11.2.5.1 TPM2B_ECC_PARAMETER ... 134
11.2.5.2 TPMS_ECC_POINT ... 134
11.2.5.3 TPM2B_ECC_POINT ... 134
11.2.5.4 TPMI_ALG_ECC_SCHEME ... 135
11.2.5.5 TPMI_ECC_CURVE ... 135
11.2.5.6 TPMT_ECC_SCHEME ... 135
11.2.5.7 TPMS_ALGORITHM_DETAIL_ECC ... 136
11.3 Signatures ... 136
11.3.1 TPMS_SIGNATURE_RSA ... 136
11.3.2 TPMS_SIGNATURE_ECC ... 137
11.3.3 TPMU_SIGNATURE ... 137
11.3.4 TPMT_SIGNATURE ... 137
11.4 Key/Secret Exchange ... 138
11.4.1 Introduction ... 138
11.4.2 TPMU_ENCRYPTED_SECRET ... 138
11.4.3 TPM2B_ENCRYPTED_SECRET ... 138
12 Key/Object Complex ... 139
12.1 Introduction ... 139
12.2 Public Area Structures ... 139
12.2.1 Description ... 139
12.2.2 TPMI_ALG_PUBLIC ... 139
12.2.3 Type-Specific Parameters ... 140
12.2.3.1 Description ... 140
12.2.3.2 TPMU_PUBLIC_ID ... 140
12.2.3.3 TPMS_KEYEDHASH_PARMS ... 141
12.2.3.4 TPMS_ASYM_PARMS ... 141
12.2.3.5 TPMS_RSA_PARMS ... 142
12.2.3.6 TPMS_ECC_PARMS ... 143
12.2.3.7 TPMU_PUBLIC_PARMS ... 143
12.2.3.8 TPMT_PUBLIC_PARMS ... 144
12.2.4 TPMT_PUBLIC ... 144
12.2.5 TPM2B_PUBLIC ... 144
12.2.6 TPM2B_TEMPLATE ... 145
12.3 Private Area Structures ... 145
12.3.1 Introduction ... 145
12.3.2 Sensitive Data Structures ... 145
12.3.2.1 Introduction ... 145
12.3.2.2 TPM2B_PRIVATE_VENDOR_SPECIFIC ... 145
12.3.2.3 TPMU_SENSITIVE_COMPOSITE ... 146
12.3.2.4 TPMT_SENSITIVE ... 146
12.3.3 TPM2B_SENSITIVE ... 146
12.3.4 Encryption ... 147
12.3.5 Integrity ... 147
12.3.6 _PRIVATE ... 147
12.3.7 TPM2B_PRIVATE ... 147
12.4 Identity Object ... 148
12.4.1 Description ... 148
12.4.2 TPMS_ID_OBJECT ... 148
12.4.3 TPM2B_ID_OBJECT ... 148
13 NV Storage Structures ... 149
13.1 TPM_NV_INDEX ... 149
13.2 TPM_NT ... 150

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

13.3 TPMS_NV_PIN_COUNTER_PARAMETERS 150
13.4 TPMA_NV (NV Index Attributes) 150
13.5 TPMS_NV_PUBLIC 154
13.6 TPM2B_NV_PUBLIC 154

14 Context Data 155

14.1 Introduction 155
14.2 TPM2B_CONTEXT_SENSITIVE 155
14.3 TPMS_CONTEXT_DATA 155
14.4 TPM2B_CONTEXT_DATA 155
14.5 TPMS_CONTEXT 156
14.6 Parameters of TPMS_CONTEXT 157

14.6.1 *sequence* 157
14.6.2 *savedHandle* 157
14.6.3 *hierarchy* 158

14.7 Context Protection 158

14.7.1 Context Integrity 158
14.7.2 Context Confidentiality 158

15 Creation Data 159

15.1 TPMS_CREATION_DATA 159
15.2 TPM2B_CREATION_DATA 159

16 Attached Component Structures 160

16.1 TPM_AT 160
16.2 TPM_AE 160
16.3 TPMS_AC_OUTPUT 160
16.4 TPML_AC_CAPABILITIES 161

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

Tables

Table 1 — Name Prefix Convention 20
Table 2 — Unmarshaling Errors 22
Table 3 — Definition of Base Types 23
Table 4 — Defines for Logic Values 23
Table 5 — Definition of Types for Documentation Clarity 24
Table 6 — Definition of (UINT32) TPM_SPEC Constants <> 25
Table 7 — Definition of (UINT32) TPM_GENERATED Constants <O> 25
Table 8 — Legend for TPM_ALG_ID Table 26
Table 9 — Definition of (UINT16) TPM_ALG_ID Constants <IN/OUT, S> 27
Table 10 — Definition of (UINT16) {ECC} TPM_ECC_CURVE Constants <IN/OUT> 30
Table 11 — TPM Command Format Fields Description 30
Table 12 — Definition of (UINT32) TPM_CC Constants (Numeric Order) <IN/OUT, S> 31
Table 13 — Format-Zero Response Codes 36
Table 14 — Format-One Response Codes 37
Table 15 — Response Code Groupings 37
Table 16 — Definition of (UINT32) TPM_RC Constants (Actions) <OUT> 38
Table 17 — Definition of (INT8) TPM_CLOCK_ADJUST Constants <IN> 43
Table 18 — Definition of (UINT16) TPM_EO Constants <IN/OUT> 43
Table 19 — Definition of (UINT16) TPM_ST Constants <IN/OUT, S> 45
Table 20 — Definition of (UINT16) TPM_SU Constants <IN> 47
Table 21 — Definition of (UINT8) TPM_SE Constants <IN> 47
Table 22 — Definition of (UINT32) TPM_CAP Constants 48
Table 23 — Definition of (UINT32) TPM_PT Constants <IN/OUT, S> 49
Table 24 — Definition of (UINT32) TPM_PT_PCR Constants <IN/OUT, S> 54
Table 25 — Definition of (UINT32) TPM_PS Constants <OUT> 56
Table 26 — Definition of Types for Handles 57
Table 27 — Definition of (UINT8) TPM_HT Constants <S> 57
Table 28 — Definition of (TPM_HANDLE) TPM_RH Constants <S> 59
Table 29 — Definition of (TPM_HANDLE) TPM_HC Constants <S> 61
Table 30 — Definition of (UINT32) TPMA_ALGORITHM Bits 63
Table 31 — Definition of (UINT32) TPMA_OBJECT Bits 64
Table 32 — Definition of (UINT8) TPMA_SESSION Bits <IN/OUT> 71
Table 33 — Definition of (UINT8) TPMA_LOCALITY Bits <IN/OUT> 73
Table 34 — Definition of (UINT32) TPMA_PERMANENT Bits <OUT> 74
Table 35 — Definition of (UINT32) TPMA_STARTUP_CLEAR Bits <OUT> 75
Table 36 — Definition of (UINT32) TPMA_MEMORY Bits <Out> 76
Table 37 — Definition of (TPM_CC) TPMA_CC Bits <OUT> 77

 Copyright © TCG 2006-2020

Table 38 — Definition of (UINT32) TPMA_MODES Bits <Out> 79
Table 39 — Definition of (UINT32) TPMA_ACT Bits 81
Table 40 — Definition of (BYTE) TPMI_YES_NO Type 82
Table 41 — Definition of (TPM_HANDLE) TPMI_DH_OBJECT Type 82
Table 42 — Definition of (TPM_HANDLE) TPMI_DH_PARENT Type 83
Table 43 — Definition of (TPM_HANDLE) TPMI_DH_PERSISTENT Type 83
Table 44 — Definition of (TPM_HANDLE) TPMI_DH_ENTITY Type <IN> 84
Table 45 — Definition of (TPM_HANDLE) TPMI_DH_PCR Type <IN> 84
Table 46 — Definition of (TPM_HANDLE) TPMI_SH_AUTH_SESSION Type <IN/OUT> 85
Table 47 — Definition of (TPM_HANDLE) TPMI_SH_HMAC Type <IN/OUT> 85
Table 48 — Definition of (TPM_HANDLE) TPMI_SH_POLICY Type <IN/OUT> 85
Table 49 — Definition of (TPM_HANDLE) TPMI_DH_CONTEXT Type 85
Table 50 — Definition of (TPM_HANDLE) TPMI_DH_SAVED Type 86
Table 51 — Definition of (TPM_HANDLE) TPMI_RH_HIERARCHY Type 86
Table 52 — Definition of (TPM_HANDLE) TPMI_RH_ENABLES Type 86
Table 53 — Definition of (TPM_HANDLE) TPMI_RH_HIERARCHY_AUTH Type <IN> 87
Table 54 — Definition of (TPM_HANDLE) TPMI_RH_HIERARCHY_POLICY Type <IN> 87
Table 55 — Definition of (TPM_HANDLE) TPMI_RH_PLATFORM Type <IN> 87
Table 56 — Definition of (TPM_HANDLE) TPMI_RH_OWNER Type <IN> 88
Table 57 — Definition of (TPM_HANDLE) TPMI_RH_ENDORSEMENT Type <IN> 88
Table 58 — Definition of (TPM_HANDLE) TPMI_RH_PROVISION Type <IN> 88
Table 59 — Definition of (TPM_HANDLE) TPMI_RH_CLEAR Type <IN> 89
Table 60 — Definition of (TPM_HANDLE) TPMI_RH_NV_AUTH Type <IN> 89
Table 61 — Definition of (TPM_HANDLE) TPMI_RH_LOCKOUT Type <IN> 89
Table 62 — Definition of (TPM_HANDLE) TPMI_RH_NV_INDEX Type <IN/OUT> 90
Table 63 — Definition of (TPM_HANDLE) TPMI_RH_AC Type <IN> 90
Table 64 — Definition of (TPM_HANDLE) TPMI_RH_ACT Type 91
Table 65 — Definition of (TPM_ALG_ID) TPMI_ALG_HASH Type 91
Table 66 — Definition of (TPM_ALG_ID) TPMI_ALG_ASYM Type 91
Table 67 — Definition of (TPM_ALG_ID) TPMI_ALG_SYM Type 92
Table 68 — Definition of (TPM_ALG_ID) TPMI_ALG_SYM_OBJECT Type 92
Table 69 — Definition of (TPM_ALG_ID) TPMI_ALG_SYM_MODE Type 92
Table 70 — Definition of (TPM_ALG_ID) TPMI_ALG_KDF Type 93
Table 71 — Definition of (TPM_ALG_ID) TPMI_ALG_SIG_SCHEME Type 93
Table 72 — Definition of (TPM_ALG_ID){ECC} TPMI_ECC_KEY_EXCHANGE Type 93
Table 73 — Definition of (TPM_ST) TPMI_ST_COMMAND_TAG Type 94
Table 74 — Definition of (TPM_ALG_ID) TPMI_ALG_MAC_SCHEME Type 94
Table 75 — Definition of (TPM_ALG_ID) TPMI_ALG_CIPHER_MODE Type 94
Table 76 — Definition of TPMS_EMPTY Structure <IN/OUT> 95

 Copyright © TCG 2006-2020

Table 77 — Definition of TPMS_ALGORITHM_DESCRIPTION Structure <OUT> 95
Table 78 — Definition of TPMU_HA Union <IN/OUT > 95
Table 79 — Definition of TPMT_HA Structure <IN/OUT> 96
Table 80 — Definition of TPM2B_DIGEST Structure 97
Table 81 — Definition of TPM2B_DATA Structure 97
Table 82 — Definition of Types for TPM2B_NONCE 97
Table 83 — Definition of Types for TPM2B_AUTH 97
Table 84 — Definition of Types for TPM2B_OPERAND 98
Table 85 — Definition of TPM2B_EVENT Structure 98
Table 86 — Definition of TPM2B_MAX_BUFFER Structure 98
Table 87 — Definition of TPM2B_MAX_NV_BUFFER Structure 98
Table 88 — Definition of TPM2B_TIMEOUT Structure 99
Table 89 — Definition of TPM2B_IV Structure <IN/OUT> 99
Table 90 — Definition of TPMU_NAME Union <> 99
Table 91 — Definition of TPM2B_NAME Structure 100
Table 92 — Definition of TPMS_PCR_SELECT Structure 101
Table 93 — Definition of TPMS_PCR_SELECTION Structure 101
Table 94 — Values for *proof* Used in Tickets 102
Table 95 — General Format of a Ticket 102
Table 96 — Definition of TPMT_TK_CREATION Structure 103
Table 97 — Definition of TPMT_TK_VERIFIED Structure 103
Table 98 — Definition of TPMT_TK_AUTH Structure 104
Table 99 — Definition of TPMT_TK_HASHCHECK Structure 105
Table 100 — Definition of TPMS_ALG_PROPERTY Structure <OUT> 105
Table 101 — Definition of TPMS_TAGGED_PROPERTY Structure <OUT> 105
Table 102 — Definition of TPMS_TAGGED_PCR_SELECT Structure <OUT> 106
Table 103 — Definition of TPMS_TAGGED_POLICY Structure <OUT> 106
Table 104 — Definition of TPMS_ACT_DATA Structure <OUT> 106
Table 105 — Definition of TPML_CC Structure 107
Table 106 — Definition of TPML_CCA Structure <OUT> 107
Table 107 — Definition of TPML_ALG Structure 107
Table 108 — Definition of TPML_HANDLE Structure <OUT> 108
Table 109 — Definition of TPML_DIGEST Structure 108
Table 110 — Definition of TPML_DIGEST_VALUES Structure 109
Table 111 — Definition of TPML_PCR_SELECTION Structure 109
Table 112 — Definition of TPML_ALG_PROPERTY Structure <OUT> 110
Table 113 — Definition of TPML_TAGGED_TPM_PROPERTY Structure <OUT> 110
Table 114 — Definition of TPML_TAGGED_PCR_PROPERTY Structure <OUT> 110
Table 115 — Definition of {ECC} TPML_ECC_CURVE Structure <OUT> 110

 Copyright © TCG 2006-2020

Table 116 — Definition of TPML_TAGGED_POLICY Structure <OUT> ... 111
Table 117 — Definition of TPML_ACT_DATA Structure <OUT> ... 111
Table 118 — Definition of TPMU_CAPABILITIES Union <OUT> ... 112
Table 119 — Definition of TPMS_CAPABILITY_DATA Structure <OUT> ... 112
Table 120 — Definition of TPMS_CLOCK_INFO Structure ... 113
Table 121 — Definition of TPMS_TIME_INFO Structure ... 114
Table 122 — Definition of TPMS_TIME_ATTEST_INFO Structure <OUT> ... 115
Table 123 — Definition of TPMS_CERTIFY_INFO Structure <OUT> ... 115
Table 124 — Definition of TPMS_QUOTE_INFO Structure <OUT> ... 115
Table 125 — Definition of TPMS_COMMAND_AUDIT_INFO Structure <OUT> ... 116
Table 126 — Definition of TPMS_SESSION_AUDIT_INFO Structure <OUT> ... 116
Table 127 — Definition of TPMS_CREATION_INFO Structure <OUT> ... 116
Table 128 — Definition of TPMS_NV_CERTIFY_INFO Structure <OUT> ... 116
Table 129 — Definition of TPMS_NV_DIGEST_CERTIFY_INFO Structure <OUT> ... 117
Table 130 — Definition of (TPM_ST) TPMI_ST_ATTEST Type <OUT> ... 117
Table 131 — Definition of TPMU_ATTEST Union <OUT> ... 117
Table 132 — Definition of TPMS_ATTEST Structure <OUT> ... 118
Table 133 — Definition of TPM2B_ATTEST Structure <OUT> ... 119
Table 134 — Definition of TPMS_AUTH_COMMAND Structure <IN> ... 119
Table 135 — Definition of TPMS_AUTH_RESPONSE Structure <OUT> ... 119
Table 136 — Definition of {!ALG.S} (TPM_KEY_BITS) TPMI_!ALG.S_KEY_BITS Type ... 120
Table 137 — Definition of TPMU_SYM_KEY_BITS Union ... 120
Table 138 — Definition of TPMU_SYM_MODE Union ... 121
Table 139 —xDefinition of TPMU_SYM_DETAILS Union ... 121
Table 140 — Definition of TPMT_SYM_DEF Structure ... 122
Table 141 — Definition of TPMT_SYM_DEF_OBJECT Structure ... 122
Table 142 — Definition of TPM2B_SYM_KEY Structure ... 123
Table 143 — Definition of TPMS_SYMCIPHER_PARMS Structure ... 123
Table 144 — Definition of TPM2B_LABEL Structure ... 123
Table 145 — Definition of TPMS_DERIVE Structure ... 123
Table 146 — Definition of TPM2B_DERIVE Structure ... 124
Table 147 — Definition of TPMU_SENSITIVE_CREATE Union <> ... 124
Table 148 — Definition of TPM2B_SENSITIVE_DATA Structure ... 124
Table 149 — Definition of TPMS_SENSITIVE_CREATE Structure <IN> ... 125
Table 150 — Definition of TPM2B_SENSITIVE_CREATE Structure <IN, S> ... 125
Table 151 — Definition of TPMS_SCHEME_HASH Structure ... 125
Table 152 — Definition of {ECC} TPMS_SCHEME_ECDAA Structure ... 126
Table 153 — Definition of (TPM_ALG_ID) TPMI_ALG_KEYEDHASH_SCHEME Type ... 126
Table 154 — Definition of Types for HMAC_SIG_SCHEME ... 126

 Copyright © TCG 2006-2020

Table 155 — Definition of TPMS_SCHEME_XOR Structure 126
Table 156 — Definition of TPMU_SCHEME_KEYEDHASH Union <IN/OUT > 127
Table 157 — Definition of TPMT_KEYEDHASH_SCHEME Structure 127
Table 158 — Definition of {RSA} Types for RSA Signature Schemes 128
Table 159 — Definition of {ECC} Types for ECC Signature Schemes 128
Table 160 — Definition of TPMU_SIG_SCHEME Union <IN/OUT > 129
Table 161 — Definition of TPMT_SIG_SCHEME Structure 129
Table 162 — Definition of Types for {RSA} Encryption Schemes 129
Table 163 — Definition of Types for {ECC} ECC Key Exchange 130
Table 164 — Definition of Types for KDF Schemes 130
Table 165 — Definition of TPMU_KDF_SCHEME Union <IN/OUT> 130
Table 166 — Definition of TPMT_KDF_SCHEME Structure 130
Table 167 — Definition of (TPM_ALG_ID) TPMI_ALG_ASYM_SCHEME Type <IO> 131
Table 168 — Definition of TPMU_ASYM_SCHEME Union 131
Table 169 — Definition of TPMT_ASYM_SCHEME Structure <> 132
Table 170 — Definition of (TPM_ALG_ID) {RSA} TPMI_ALG_RSA_SCHEME Type 132
Table 171 — Definition of {RSA} TPMT_RSA_SCHEME Structure 132
Table 172 — Definition of (TPM_ALG_ID) {RSA} TPMI_ALG_RSA_DECRYPT Type 132
Table 173 — Definition of {RSA} TPMT_RSA_DECRYPT Structure 132
Table 174 — Definition of {RSA} TPM2B_PUBLIC_KEY_RSA Structure 133
Table 175 — Definition of {RSA} (TPM_KEY_BITS) TPMI_RSA_KEY_BITS Type 133
Table 176 — Definition of {RSA} TPM2B_PRIVATE_KEY_RSA Structure 133
Table 177 — Definition of TPM2B_ECC_PARAMETER Structure 134
Table 178 — Definition of {ECC} TPMS_ECC_POINT Structure 134
Table 179 — Definition of {ECC} TPM2B_ECC_POINT Structure 134
Table 180 — Definition of (TPM_ALG_ID) {ECC} TPMI_ALG_ECC_SCHEME Type 135
Table 181 — Definition of {ECC} (TPM_ECC_CURVE) TPMI_ECC_CURVE Type 135
Table 182 — Definition of (TPMT_SIG_SCHEME) {ECC} TPMT_ECC_SCHEME Structure 135
Table 183 — Definition of {ECC} TPMS_ALGORITHM_DETAIL_ECC Structure <OUT> 136
Table 184 — Definition of {RSA} TPMS_SIGNATURE_RSA Structure 136
Table 185 — Definition of Types for {RSA} Signature 136
Table 186 — Definition of {ECC} TPMS_SIGNATURE_ECC Structure 137
Table 187 — Definition of Types for {ECC} TPMS_SIGNATURE_ECC 137
Table 188 — Definition of TPMU_SIGNATURE Union <IN/OUT> 137
Table 189 — Definition of TPMT_SIGNATURE Structure 137
Table 190 — Definition of TPMU_ENCRYPTED_SECRET Union 138
Table 191 — Definition of TPM2B_ENCRYPTED_SECRET Structure 138
Table 192 — Definition of (TPM_ALG_ID) TPMI_ALG_PUBLIC Type 139
Table 193 — Definition of TPMU_PUBLIC_ID Union <IN/OUT> 140

 Copyright © TCG 2006-2020

Table 194 — Definition of TPMS_KEYEDHASH_PARMS Structure 141
Table 195 — Definition of TPMS_ASYM_PARMS Structure <> 141
Table 196 — Definition of {RSA} TPMS_RSA_PARMS Structure 142
Table 197 — Definition of {ECC} TPMS_ECC_PARMS Structure 143
Table 198 — Definition of TPMU_PUBLIC_PARMS Union <IN/OUT> 143
Table 199 — Definition of TPMT_PUBLIC_PARMS Structure 144
Table 200 — Definition of TPMT_PUBLIC Structure 144
Table 201 — Definition of TPM2B_PUBLIC Structure 144
Table 202 — Definition of TPM2B_TEMPLATE Structure 145
Table 203 — Definition of TPM2B_PRIVATE_VENDOR_SPECIFIC Structure 145
Table 204 — Definition of TPMU_SENSITIVE_COMPOSITE Union <IN/OUT> 146
Table 205 — Definition of TPMT_SENSITIVE Structure 146
Table 206 — Definition of TPM2B_SENSITIVE Structure <IN/OUT> 146
Table 207 — Definition of _PRIVATE Structure <> 147
Table 208 — Definition of TPM2B_PRIVATE Structure <IN/OUT> 147
Table 209 — Definition of TPMS_ID_OBJECT Structure <> 148
Table 210 — Definition of TPM2B_ID_OBJECT Structure <IN/OUT> 148
Table 211 — Definition of (UINT32) TPM_NV_INDEX Bits <> 149
Table 212 — Definition of TPM_NT Constants 150
Table 213 — Definition of TPMS_NV_PIN_COUNTER_PARAMETERS Structure 150
Table 214 — Definition of (UINT32) TPMA_NV Bits 152
Table 215 — Definition of TPMS_NV_PUBLIC Structure 154
Table 216 — Definition of TPM2B_NV_PUBLIC Structure 154
Table 217 — Definition of TPM2B_CONTEXT_SENSITIVE Structure <IN/OUT> 155
Table 218 — Definition of TPMS_CONTEXT_DATA Structure <IN/OUT> 155
Table 219 — Definition of TPM2B_CONTEXT_DATA Structure <IN/OUT> 155
Table 220 — Definition of TPMS_CONTEXT Structure 156
Table 221 — Context Handle Values 157
Table 222 — Definition of TPMS_CREATION_DATA Structure <OUT> 159
Table 223 — Definition of TPM2B_CREATION_DATA Structure <OUT> 159
Table 224 — Definition of (UINT32) TPM_AT Constants 160
Table 225 — Definition of (UINT32) TPM_AE Constants <OUT> 160
Table 226 — Definition of TPMS_AC_OUTPUT Structure <OUT> 160
Table 227 — Definition of TPML_AC_CAPABILITIES Structure <OUT> 161

 Copyright © TCG 2006-2020

Figures

Figure 1 — Command Format 30
Figure 2 — Format-Zero Response Codes 36
Figure 3 — Format-One Response Codes 36
Figure 4 — TPM 1.2 TPM_NV_INDEX 149
Figure 5 — TPM 2.0 TPM_NV_INDEX 149

 Copyright © TCG 2006-2020

Trusted Platform Module Library

Part 2: Structures

1 Scope

This part of the *Trusted Platform Module Library* specification contains the definitions of the constants, flags, structure, and union definitions used to communicate with the TPM. Values defined in this document are used by the TPM commands defined in TPM 2.0 Part 3: *Commands* and by the functions in TPM 2.0 Part 4: *Supporting Routines*.

NOTE The structures in this document are the canonical form of the structures on the interface. All structures are "packed" with no octets of padding between structure elements. The TPM-internal form of the structures is dependent on the processor and compiler for the TPM implementation.

2 Terms and definitions

For the purposes of this document, the terms and definitions given in TPM 2.0 Part 1 apply.

3 Symbols and abbreviated terms

For the purposes of this document, the symbols and abbreviated terms given in TPM 2.0 Part 1 apply.

4 Notation

4.1 Introduction

The information in this document is formatted so that it may be converted to standard computer-language formats by an automated process. The purpose of this automated process is to minimize the transcription errors that often occur during the conversion process.

For the purposes of this document, the conventions given in TPM 2.0 Part 1 apply.

In addition, the conventions and notations in clause 4 describe the representation of various data so that it is both human readable and amenable to automated processing.

When a table row contains the keyword “reserved” (all lower case) in columns 1 or 2, the tools will not produce any values for the row in the table.

NOTE The unmarshaling code examples are the actual code that would be produced by the automatic code generator used in the construction of the reference code. The actual code contains additional parameter checking that is omitted for clarity of the principle being illustrated. Actual examples of the code are found in TPM 2.0 Part 4.

 Copyright © TCG 2006-2020

4.2 Named Constants

A named constant is a numeric value to which a name has been assigned. In the C language, this is done with a `#define` statement. In this specification, a named constant is defined in a table that has a title that starts with “Definition” and ends with “Constants.”

The table title will indicate the name of the class of constants that are being defined in the table. When applicable, the title will include the data type of the constants in parentheses.

The table in Example 1 names a collection of 16-bit constants and Example 2 shows the C code that might be produced from that table by an automated process.

NOTE A named constant (**#define**) has no data type in C and an enumeration would be a better choice for many of the defined constants. However, the C language does not allow an enumerated type to have a storage type other than **int** so the method of using a combination of **typedef** and **#define** is used.

EXAMPLE 1

Table xx — Definition of (UINT16) COUNTING Constants

Parameter	Value	Description
first	1	decimal value is implicitly the size of the
second	0x0002	hex value will match the number of bits in the constant
third	3	
fourth	0x0004	

EXAMPLE 2

```
/* The C language equivalent of the constants from the table above */
typedef     UINT16      COUNTING;
#define     first       1
#define     second      0x0002
#define     third       3
#define     fourth      0x0004
```


 Copyright © TCG 2006-2020

4.3 Data Type Aliases (typedefs)

When a group of named items is assigned a type, it is placed in a table that has a title starting with "Definition of Types." In this specification, defined types have names that use all upper-case characters.

The table in Example 1 shows how typedefs would be defined in this specification and Example 2 shows the C-compatible code that might be produced from that table by an automated process.

EXAMPLE 1

Table xx — Definition of Types for Some Purpose

Type	Name	Description
unsigned short	UINT16	
UINT16	SOME_TYPE	
unsigned long	UINT32	
UINT32	LAST_TYPE	

EXAMPLE 2

```
/* C language equivalent of the typedefs from the table above */
typedef unsigned short    UINT16;
typedef UINT16            SOME_TYPE;
typedef unsigned long     UINT32;
typedef UINT32            LAST_TYPE;
```

4.4 Enumerations

A table that defines an enumerated data type will start with the word "Definition" and end with "Values."

A value in parenthesis will denote the intrinsic data size of the value and may have the values "INT8", "UINT8", "INT16", "UINT16", "INT32", and "UINT32." If this value is not present, "UINT16" is assumed.

Most C compilers set the type of an enumerated value to be an integer on the machine – often 16 bits – but this is not always consistent. To ensure interoperability, the enumeration values may not exceed 32,384.

The table in Example 1 shows how an enumeration would be defined in this specification. Example 2 shows the C code that might be produced from that table by an automated process.

EXAMPLE 1

Table xx — Definition of (UINT16) CARD_SUIT Values

Suit Names	Value	Description
CLUBS	0x0000	
DIAMONDS	0x000D	
HEARTS	0x001A	
SPADES	0x0027	

EXAMPLE 2

```
/* C language equivalent of the structure defined in the table above */
typedef enum {
    CLUBS       =   0x0000,
    DIAMONDS    =   0x000D,
    HEARTS      =   0x001A,
    SPADES      =   0x0027
} CARD_SUIT;
```


 Copyright © TCG 2006-2020

4.5 Interface Type

An interface type is used for an enumeration that is checked by the unmarshaling code. This type is defined for purposes of automatic generation of the code that will validate the type. The title will start with the keyword “Definition” and end with the keyword “Type.” A value in parenthesis indicates the base type of the interface. The table may contain an entry that is prefixed with the “#” character to indicate the response code if the validation code determines that the input parameter is the wrong type.

EXAMPLE 1

Table xx — Definition of (CARD_SUIT) RED_SUIT Type

Values	Comments
HEARTS	
DIAMONDS	
#TPM_RC_SUIT	response code returned when the unmarshaling of this type fails NOTE TPM_RC_SUIT is an example and no such response code is actually defined in this specification.

EXAMPLE 2

```
/* Validation code that might be automatically generated from table above */
if((*target != HEARTS) && (*target != DIAMONDS))
    return TPM_RC_SUIT;
```

In some cases, the allowed values are numeric values with no associated mnemonic. In such a case, the list of numeric values may be given a name. Then, when used in an interface definition, the name would have a "$" prefix to indicate that a named list of values should be substituted.

To illustrate, assume that the implementation only supports two sizes (1024 and 2048 bits) for keys associated with some algorithm (MY algorithm).

EXAMPLE 3

Table xx — Defines for MY Algorithm Constants

Name	Value	Comments
MY_KEY_SIZES_BITS	{1024, 2048}	braces because this is a list value

 Copyright © TCG 2006-2020

Then, whenever an input value would need to be a valid MY key size for the implementation, the value $MY_KEY_SIZES_BITS could be used. Given the definition for MY_KEY_SIZES_BITS in example 3 above, the tables in example 4 and 5 below, are equivalent.

EXAMPLE 4

Table xx — Definition of (UINT16) MY_KEY_BITS Type

Parameter	Description
{1024, 2048}	the number of bits in the supported key

EXAMPLE 5

Table xx — Definition of (UINT16) MY_KEY_BITS Type

Parameter	Description
$MY_KEY_SIZES_BITS	the number of bits in the supported key

4.6 Arrays

Arrays are denoted by a value in square brackets ("[ ]") following a parameter name. The value in the brackets may be either an integer value such as "[20]" or the name of a component of the same structure that contains the array.

The table in Example 1 shows how a structure containing fixed and variable-length arrays would be defined in this specification. Example 2 shows the C code that might be produced from that table by an automated process.

EXAMPLE 1

Table xx — Definition of A_STRUCT Structure

Parameter	Type	Description
array1[20]	UINT16	an array of 20 UINT16s
a_size	UINT16	
array2[a_size]	UINT32	an array of UINT32 values that has a number of elements determined by a_size above

EXAMPLE 2

```
/* C language equivalent of the typedefs from the table above */
typedef struct {
    UINT16      array1[20];
    UINT16      a_size;
    UINT32      array2[];
} A_STRUCT;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

4.7 Structure Definitions

The tables used to define structures have a title that starts with the word "Definition" and ends with "Structure." The first column of the table will denote the reference names for the structure members; the second column the data type of the member; and the third column a synopsis of the use of the element.

The table in Example 1 shows an example of how a structure would be defined in this specification and Example 2 shows the C code that might be produced from the table by an automated process. Example 3 illustrates the type of unmarshaling code that could be generated using the information available in the table.

EXAMPLE 1

Table xx — Definition of SIMPLE_STRUCTURE Structure

Parameter	Type	Description
tag	TPM_ST	
value1	INT32	
value2	INT32	

EXAMPLE 2

```
/* C language equivalent of the structure defined in the table above */
typedef struct {
    TPM_ST      tag;
    INT32       value1
    INT32       value2;
} SIMPLE_STRUCTURE;
```

EXAMPLE 3

```
TPM_RC SIMPLE_STRUCTURE_Unmarshal(SIMPLE_STRUCTURE *target, BYTE **buffer, INT32 *size)
{
    TPM_RC      rc;
    // If unmarshal of tag succeeds
    rc = TPM_ST_Unmarshal((TPM_ST *)&(target->tag), buffer, size
    If(rc  == TPM_RC_SUCCESS)
    {
         // then unmarshal value1,
        rc = INT32_Unmarshal((INT32 *)&(target->value1, buffer, size);
        // and if that succeeds...
        if(rc  == TPM_RC_SUCCESS)
        {
            // then unmarshal the value 2
            rc = INT32_Unmarshal((INT32 *)&(target->value2, buffer, size);
        }
    }
    return rc;
}
```

 Copyright © TCG 2006-2020

A table may have a term in {}. This indicates that the table is conditionally compiled. It is commonly used when a table's inclusion is based on the implementation of a cryptographic algorithm. See, for example, Table 172 — Definition of (TPM_ALG_ID) {RSA} TPMI_ALG_RSA_DECRYPT Type, which is dependent on the RSA algorithm.

4.8 Conditional Types

An interface type may have a conditional value. This value is indicated by a “+” prepended to the name of the value. When this type is referenced in a structure, a “+” appended to the reference indicates that the instance allows the conditional value to be returned. If the reference does not has an appended “+”, then the conditional type is not allowed.

EXAMPLE 1 Table 65 defining TPMI_ALG_HASH indicates that TPM_ALG_NULL is a conditional type. TPMI_ALG_HASH is a member of the TPMS_SCHEME_XOR structure and that reference is TPMI_ALG_HASH+, indicating that TPM_ALG_NULL is an allowed value for hashAlg. TPMI_ALG_HASH is also referenced in TPMS_PCR_SELECTION. In that structure the TPMI_ALG_HASH does not have an appended “+”, so TPM_ALG_NULL would not be an allowed value for hash.

NOTE In many cases, the input values are algorithm IDs. When two collections of algorithm IDs differ only because one collection allows TPM_ALG_NULL and the other does not, it is preferred that there not be two completely different enumerations because this leads to many casts. To avoid this, the “+” can be added to a TPM_ALG_NULL value in the table defining the type. When the use of that type allows TPM_ALG_NULL to be in the set, the use would append a “+” to the instance.

When a type with a conditional value is referenced within a structure or union and the type reference has a “+” prepended to the type, it allows the references to that structure to treat it as if it had a conditional type. That means that a reference to that structure may have a “+” appended to the type. When the “+” is present in the structure/union reference, then the conditional value of the conditional type within the structure/union is allowed.

EXAMPLE 2 Table 141 — Definition of TPMT_SYM_DEF_OBJECT Structure defines the TPMT_SYM_DEF_OBJECT. The algorithm element of that structure is a TPMI_ALG_SYM_OBJECT with a “+” prepended. This means that when a TPMT_SYM_DEF_OBJECT is referenced, the reference may have an appended “+” as it does in the definition of the symmetric parameter of TPMS_ASYM_PARMS. The “+” in TPMA_ASYM_PARMS means that the algorithm parameter in the TPMT_SYM_DEF_OBJECT may have the conditional value (TPM_ALG_NULL).

EXAMPLE 3

Table xx — Definition of (CARD_SUIT) TPMI_CARD_SUIT Type

Values	Comments
SPADES	
HEARTS	
DIAMONDS	
CLUBS	
+JOKER	an optional value that may be allowed
#TPM_RC_SUIT	response code returned when the input value is not one of the values above

Level 00 Revision 01.59 **Copyright © TCG 2006-2020** November 8, 2019

EXAMPLE 4

Table xx — Definition of POKER_CARD Structure

Parameter	Type	Description
suit	TPMI_CARD_SUIT+	allows joker
number	UINT8	the card value

EXAMPLE 5

Table xx — Definition of BRIDGE_CARD Structure

Parameter	Type	Description
suit	TPMI_CARD_SUIT	does not allow joker
number	UINT8	the card value

 Copyright © TCG 2006-2020

4.9 Unions

4.9.1 Introduction

A union allows a structure to contain a variety of structures or types. The union has members, only one of which is present at a time. Three different tables are required to fully characterize a union so that it may be communicated on the TPM interface and used by the TPM:

- union definition;
- union instance; and
- union selector definition.

4.9.2 Union Definition

The table in Example 1 illustrates a union definition. The title of a union definition table starts with "Definition" and ends with "Union." The "Parameter" column of a union definition lists the different names that are used when referring to a specific type. The "Type" column identifies the data type of the member. The "Selector" column identifies the value that is used by the marshaling and unmarshaling code to determine which case of the union is present.

If a parameter is the keyword "null" or the type is empty, then this denotes a selector with no contents. The table in Example 1 illustrates a union in which a conditional null selector is allowed to indicate an empty union member.

Example 2 shows how the table would be converted into C-compatible code.

The expectation is that the unmarshaling code for the union will validate that the selector for the union is one of values in the selector list.

EXAMPLE 1

Table xx — Definition of NUMBER_UNION Union

Parameter	Type	Selector	Description
a_byte	BYTE	BYTE_SELECT	
an_int	int	INT_SELECT	
a_float	float	FLOAT_SELECT	
+null		NULL_SELECT	the empty branch

EXAMPLE 2

```
// C-compatible version of the union defined in the table above
typedef union {
    BYTE            a_byte;
    int             an_int;
    float           a_float;
} NUMBER_UNION;
```


 Copyright © TCG 2006-2020

EXAMPLE 3

```
// Possible auto-generated code to unmarshal a union in Example 2 based on the
// input value of selector
TPM_RC NUMBER_UNION_Unmarshal(NUMBER_UNION *target, BYTE **buffer,
                              INT32 *size, UINT32 selector)
{
    switch (selector) {
    case BYTE_SELECT:
        return BYTE_Unmarshal((BYTE *)&(target->a_byte), buffer, size);
    case INT_SELECT:
        return INT_Unmarshal((int *)&(target->an_int), buffer, size);
    case FLOAT_SELECT:
        return FLOAT_Unmarshal((float *)&(target->a_float), buffer, size);
    case NULL_SELECT:
        return TPM_RC_SUCCESS;
}
```

A table may have a type with no selector. This is used when the first part of the structure for all union members is identical. This type is a programming convenience, allowing code to reference the common members without requiring a case statement to determine the specific structure. In object oriented programming terms, this type is a superclass and the types with selectors are subclasses. Since there is no selector, this union member cannot be marshaled or unmarshaled.

EXAMPLE 4 Table 188 has an 'any' parameter with no selector. Any of the other union members may be cast to TPMS_SCHEME_HASH, since all begin with TPMI_ALG_HASH.

4.9.3 Union Instance

When a union is used in a structure that is sent on the interface, the structure will minimally contain a selector and a union. The selector value indicates which of the possible union members is present so that the unmarshaling code can unmarshal the correct type. The selector may be any of the parameters that occur in the structure before the union instance. To denote the structure parameter that is used as the selector, its name is in brackets ("[ ]") placed before the parameter name associated with the union.

The table in Example 1 shows the definition of a structure that contains a union and a selector. Example 2 shows how the table would be converted into C-compatible code and Example 3 shows how the unmarshaling code would handle the selector.

EXAMPLE 1

Table xx — Definition of STRUCTURE_WITH_UNION Structure

Parameter	Type	Description
select	NUMBER_SELECT	a value indicating the type in *number*
[select] number	NUMBER_UNION	a union as shown in 4.9.2

EXAMPLE 2

```
// C-compatible version of the union structure in the table above
typedef struct {
    NUMBER_SELECT   select;
    NUMBER_UNION    number;
} STRUCT_WITH_UNION;
```

 Copyright © TCG 2006-2020

EXAMPLE 3

```
// Possible unmarshaling code for the structure above
TPM_RC STRUCT_WITH_UNION_Unmarshal(STRUCT_WITH_UNION *target, BYTE **buffer, INT32 *size)
{
    TPM_RC          rc;
    // Unmarshal the selector value
    rc = NUMBER_SELECT_Unmarshal((NUMBER_SELECT *)&target->select, buffer, size)
    if(rc != TPM_RC_SUCCESS)
        return rc;
    // Use the unmarshaled selector value to indicate to the union unmarshal
    // function which unmarshaling branch to follow.
    return(NUMBER_UNION_Unmarshal((NUMBER_UNION *)&(target->number),
                                  buffer, size, (UINT32)target->select);
}
```

4.9.4 Union Selector Definition

The selector definition limits the values that are used in unmarshaling a union. Two different selector sets applied to the same union define different types.

For the union in 4.9.2, a selector definition should be limited to no more than four values, one for each of the union members. The selector definition could have fewer than four values.

In Example 1, the table defines a value for each of the union members.

EXAMPLE 1

Table xx — Definition of (INT8) NUMBER_SELECT Values <IN>

Name	Value	Comments
BYTE_SELECT	3	
INT_SELECT	2	
FLOAT_SELECT	1	
NULL_SELECT	0	

The unmarshaling code would limit the input values to the defined values. When the NUMBER_SELECT is used in the union instance of 4.9.3, any of the allowed union members of NUMBER_UNION could be present.

A different selection could be used to limit the values in a specific instance. To get the different selection, a new structure is defined with a different selector. The table in example 2 illustrates a way to subset the union. The base type of the selection is NUMBER_SELECT so a NUMBER_SELECT will be unmarshaled before the checks are made to see if the value is in the correct range for JUST_INTEGERS types. If the base type had been UINT8, then no checking would occur prior to checking that the value is in the allowed list. In this particular case, the effect is the same in either case since the only values that will be accepted by the unmarshaling code for JUST_INTEGER are BYTE_SELECT and INT_SELECT.

EXAMPLE 2

Table xx — Definition of (NUMBER_SELECT) AN_INTEGER Type <IN>

Values	Comments
{BYTE_SELECT, INT_SELECT}	list of allowed values

NOTE Since NULL_SELECT is not in the list of values accepted as a JUST_INTEGER, the "+" modifier will have no effect if used for a JUST_INTEGERS type shown in Example 3.

 Copyright © TCG 2006-2020

The selector in Example 2 can then be used in a subset union as shown in Example 3.

EXAMPLE 3

Table xx — Definition of JUST_INTEGERS Structure

Parameter	Type	Description
select	AN_INTEGER	a value indicating the type in *number*
[select] number	NUMBER_UNION	a union as shown in 4.9.2

4.10 Bit Field Definitions

A table that defines a structure containing bit fields has a title that starts with “Definition” and ends with “Bits.” A type identifier in parentheses in the title indicates the size of the datum that contains the bit fields.

When the bit fields do not occupy consecutive locations, a spacer field is defined with a name of “Reserved.” Bits in these spaces are reserved and shall be zero.

The table in Example 1 shows how a structure containing bit fields would be defined in this specification. Example 2 shows the C code that might be produced from that table by an automated process.

When a field has more than one bit, the range is indicated by a pair of numbers separated by a colon (“:”). The numbers will be in high:low order.

EXAMPLE1

Table xx — Definition of (UINT32) SOME_ATTRIBUTE Bits

Bit	Name	Action
0	zeroth_bit	**SET (1)**: what to do if bit is 1 **CLEAR (0):** what to do if bit is 0
1	first_bit	**SET (1)**: what to do if bit is 1 **CLEAR (0):** what to do if bit is 0
6:2	Reserved	A placeholder that spans 5 bits
7	third_bit	**SET (1)**: what to do if bit is 1 **CLEAR (0):** what to do if bit is 0
31:8	Reserved	Placeholder to fill 32 bits

EXAMPLE 2

```
/* C language equivalent of the attributes structure defined in the table above */
typedef struct {
    int zeroth_bit : 1;
    int first_bit  : 1;
    int Reserved3  : 5;
    int third_bit  : 1;
    int Reserved7  : 24;
} SOME_ATTRIBUTE;
```

NOTE The packing of bit fields into an integer is compiler and tool chain dependent. This C language equivalent is valid for a compiler that packs bit fields from the least significant bit to the most significant bit. It is likely to be correct for a little endian processor and likely to be incorrect for a big endian processor.

 Copyright © TCG 2006-2020

4.11 Parameter Limits

A parameter used in a structure may be given a set of values that can be checked by the unmarshaling code. The allowed values for a parameter may be included in the definition of the parameter by appending the values and delimiting them with braces ("{ }"). The values are comma-separated expressions. A range of numbers may be indicated by separating two expressions with a colon (":"). The first number is an expression that represents the minimum allowed value and the second number indicates the maximum. If the minimum or maximum value expression is omitted, then the range is open-ended.

Lower limits expressed using braces apply only to inputs to the TPM. The lower limit for a value returned by the TPM is determined by input parameters and the TPM implementation. Upper limits expressed using braces apply to both inputs to and outputs from the TPM.

NOTE In many cases, the upper limits are dependent on the TPM implementation. The values for these limits can be determined by accessing the TPM's capabilities.

Level 00 Revision 01.59 **Copyright © TCG 2006-2020** November 8, 2019

The maximum size of an array may be indicated by putting a "{}" delimited expression following the square brackets ("[ ]") that indicate that the value is an array.

EXAMPLE

Table xx — Definition of B_STRUCT Structure

Parameter	Type	Description
value1 {20:25}	UINT16	a parameter that must have a value between 20 and 25, inclusive
value2 {20}	UINT16	a parameter that must have a value of 20
value3 {:25}	INT16	a parameter that may be no larger than 25 Since the parameter is signed, the minimum value is the largest negative integer that may be expressed in 16 bits.
value4 {20:}		a parameter that must be at least 20
value5 {1,2,3,5}	UINT16	a parameter that may only have one of the four listed values
value6 {1, 2, 10:(10+10)}	UINT32	a parameter that may have a value of 1, 2, or be between 10 and 20
array1[value1]	BYTE	Because the index refers to *value1*, which is a value limited to be between 20 and 25 inclusive, array1 is an array that may have between 20 and 25 octets. This is not the preferred way to indicate the upper limit for an array as it does not indicate the upper bound of the size. NOTE This is a limitation of the current parser. A different parser could associate the range of value1 with this value and compute the maximum size of the array.
array2[value4]{:25}	BYTE	an array that may have between 20 and 25 octets This arrangement is used to allow the automatic code generation to allocate 25 octets to store the largest array2 that can be unmarshaled. The code generation can determine from this expression that *value4* shall have a value of 25 or less. From the definition of *value4* above, it can determine that *value4* must have a value of at least 20.

4.12 Algorithm Macros

4.12.1 Introduction

This specification is intended to be algorithm agile in two different ways. In the first, agility is provided by allowing different subsets of the algorithms listed in the TCG registry. In the second, agility is provided by allowing the list of algorithms in the TCG registry to change without requiring changes to this specification.

This second form of algorithm agility is accomplished by using placeholder tokens that represent all of the algorithms of a particular type. The type of the algorithm is indicated by the letters in the Type column of the TPM_ALG_ID table in the TCG registry.

 Copyright © TCG 2006-2020

The use of these tokens is described in the remainder of this clause 4.12.

4.12.2 Algorithm Token Semantics

The string “!ALG” or “!alg” indicates the algorithm token. This token may be followed by an algorithm type selection. The presence of the type selection is indicated by a period (“.”) following the token. The selection is all alphanumeric characters following the period.

NOTE In this selection context, the underscore character (“_”) is not considered an alphanumeric character.

The selection is either an exclusive selection or an inclusive selection. An exclusive selection is one for which the Type entry for the algorithm is required to exactly match the type selection of the token. An inclusive selection is one where the Type entry for the algorithm is required to contain all of the characters of the selection but may contain additional attributes.

EXAMPLE 1 The “!ALG.AX” token would select those algorithms that only have the ‘A’ and ‘X’ types (that is, an asymmetric signing algorithm). The “!ALG.ax” token would select those algorithms that at least have ‘A’ and ‘X’ types but would include algorithms with other types such as ‘ANX’ (asymmetric signing and anonymous asymmetric signing).

When a replacement is made, the token will be replaced by an algorithm root identifier using either upper or lower case. If the algorithm token is part of another word, then the replacement uses upper case characters, otherwise, lower case is used.

NOTE The root identifier of an algorithm is the name in the TPM_ALG_ID table with “TPM_ALG_” removed. For example TPM_ALG_SHA1 has “SHA1” as its root.

The typical use of these tokens follows.

4.12.3 Algorithm Tokens in Unions

A common place for algorithm tokens is in a union of values that are dependent on the type of the algorithm

EXAMPLE 1 An algorithm token indicating all hashes would be “!ALG.H” and could be used in a table to indicate that a union contains all defined hashes.

Table A — Definition of TPMU_HA Union

Parameter	Type	Selector	Description
!ALG.H [!ALG_DIGEST_SIZE]	BYTE	TPM_ALG_!ALG	all hashes
null		TPM_ALG_NULL	

If the TCG registry only contained SHA1, SHA256, and the SM3_256 hash algorithm identifiers, then the table above would be semantically equivalent to:

 Copyright © TCG 2006-2020

Table xx — Definition of TPMU_HA Union

Parameter	Type	Selector	Description
sha1 [SHA1_DIGEST_SIZE]	BYTE	TPM_ALG_SHA1	
sha256 [SHA256_DIGEST_SIZE]	BYTE	TPM_ALG_SHA256	
sm3_256 [SM3_256_DIGEST_SIZE]	BYTE	TPM_ALG_SM3_256	
null		TPM_ALG_NULL	

As shown in table A, the case of the replacement is determined by context. When !ALG is not an element of a longer name, then lower case characters are used. When !ALG is part of a longer name (indicated by leading or trailing underscore ("_"), then upper case is used for the replacement.

Only one occurrence of the algorithm type (such as !ALG.H) is required for a line. If a line contains multiple list selections they are required to be identical.

If a table contains multiple lines with algorithm tokens, then each line is expanded separately.

4.12.4 Algorithm Tokens in Interface Types

An interface type is often used with a union to create a tagged structure – the structure contains a union and a tag to indicate which of the union elements is actually present. The interface type for a tagged structure will usually contain the same elements as the union.

EXAMPLE If SHA1, SHA256, and SM3_256 are the only defined hash algorithms, then an interface type to select a hash would be:

Table xx — Definition of (TPM_ALG_ID) TPMI_ALG_HASH Type

Values	Comments
TPM_ALG_SHA1	example
TPM_ALG_SHA256	example
TPM_ALG_SM3_256	example
+TPM_ALG_NULL	
#TPM_RC_HASH	

An equivalent table may be represented using an algorithm macro.

Table xx — Definition of (TPM_ALG_ID) TPMI_ALG_HASH Type

Values	Comments
TPM_ALG_!ALG.H	all hash algorithms defined by the TCG
+TPM_ALG_NULL	
#TPM_RC_HASH	

4.12.5 Algorithm Tokens for Table Replication

When a table is used to define an algorithm-specific value, that table may be replicated using the algorithm replacement token to create a table with values specific to the algorithm type. This type of replication is indicated by using an algorithm token in the name of the table.

EXAMPLE If AES and SM4 are the only defined symmetric block ciphers, then:

 Copyright © TCG 2006-2020

Table xx — Definition of {!ALG.S} (TPM_KEY_BITS) TPMI_!ALG_KEY_BITS Type

Parameter	Description
$!ALG_KEY_SIZES_BITS	number of bits in the key
#TPM_RC_VALUE	error when key size is not supported

has the same meaning as:

Table xx — Definition of {AES} (TPM_KEY_BITS) TPMI_AES_KEY_BITS Type

Parameter	Description
$AES_KEY_SIZES_BITS	number of bits in the key
#TPM_RC_VALUE	error when key size is not supported

Table xx — Definition of {SM4} (TPM_KEY_BITS) TPMI_SM4_KEY_BITS Type

Parameter	Description
$SM4_KEY_SIZES_BITS	number of bits in the key
#TPM_RC_VALUE	error when key size is not supported

 Copyright © TCG 2006-2020

4.13 Size Checking

In some structures, a size field is present to indicate the number of octets in some subsequent part of the structure. In the B_STRUCT table in 4.11, *value4* indicates how many octets to unmarshal for *array2*. This semantic applies when the size field determines the number of octets to unmarshal. However, in some cases, the subsequent structure is self-defining. If the size precedes a parameter that is not an octet array, then the unmarshaled size of that parameter is determined by its data type. The table in Example 1 shows a structure where the size parameter would nominally indicate the number of octets in the remainder of the structure.

EXAMPLE 1

Table xx — Definition of C_STRUCT Structure

Parameter	Type	Comments
size	UINT16	the expected size of the remainder of the structure
anInteger	UINT32	a 4-octet value

In this particular case, the value of size would be incorrect if it had any value other than 4. So that the table parser is able to know that the purpose of the size parameter is to define the number of octets expected in the remainder of the structure, an equal sign (“=”) is appended to the parameter name.

In the example below, the *size*= causes the parser to generate validation code that will check that the unmarshaled size of *someStructure* and *someData* adds to the value unmarshaled for *size*. When the “=” decoration is present, a value of zero is not allowed for the size.

EXAMPLE 2

Table xx — Definition of D_STRUCT Structure

Parameter	Type	Comments
size=	UINT16	the size of a structure The “=” indicates that the TPM is required to validate that the remainder of the D_STRUCT structure is exactly the value in *size*. That is, the number of bytes in the input buffer used to successfully unmarshal *someStructure* must be the same as *size*.
someStructure	A_STRUCT	a structure to be unmarshaled The size of the structure is computed when it is unmarshaled. Because an “=” is present on the definition of *size,* the TPM is required to validate that the unmarshaled size exactly matches *size*.
someData	UINT32	a value

4.14 Data Direction

A structure or union may be input (IN), output (OUT), or internal. An input structure is sent to the TPM and is unmarshaled by the TPM. An output structure is sent from the TPM and is marshaled by the TPM. An internal structure is not used outside of the TPM except that it may be included in a saved context.

By default, structures are assumed to be both IN and OUT and the code generation tool will generate both marshaling and unmarshaling code for the structure. This default may be changed by using values enclosed in angle brackets (“<>”) as part of the table title. If the angle brackets are empty, then the

 Copyright © TCG 2006-2020

structure is internal and neither marshaling nor unmarshaling code is generated. If the angle brackets contain the letter "I" (such as in "IN" or "in" or "i"), then the structure is input and unmarshaling code will be generated. If the angle brackets contain the letter "O" (such as in "OUT" or "out" or "o"), then the structure is output and marshaling code will be generated.

EXAMPLE 1 Both of the following table titles would indicate a structure that is used in both input and output

Table xx — Definition of TPMS_A Structure

Table xx — Definition of TPMS_A Structure <IN/OUT>

EXAMPLE 2 The following table title would indicate a structure that is used only for input

Table xx — Definition of TPMS_A Structure <IN>

EXAMPLE 3 The following table title would indicate a structure that is used only for output

Table xx — Definition of TPMS_A Structure <OUT>

 Copyright © TCG 2006-2020

4.15 Structure Validations

By default, when a structure is used for input to the TPM, the code generation tool will generate the unmarshaling code for that structure. Auto-generation may be suppressed by adding an "S" within the angle brackets.

EXAMPLE The following table titles indicate a structure for which the auto-generation of the validation code is to be suppressed.

Table xx — Definition of TPMT_A Structure <S>

Table xx — Definition of TPMT_A Structure <IN, S>

Table xx — Definition of TPMT_A Structure <IN/OUT, S>

4.16 Name Prefix Convention

Parameters are constants, variables, structures, unions, and structure members. Structure members are given a name that is indicative of its use, with no special prefix. The other parameter types are named according to their type with their name starting with "TPMx_", where "x" is an optional character to indicate the data type.

In some cases, additional qualifying characters will follow the underscore. These are generally used when dealing with an enumerated data type.

Table 1 — Name Prefix Convention

Prefix	Description
TPM	an indication/signal from the TPM's system interface
TPM_	a constant or an enumerated type
TPM2_	a command defined by this specification
TPM2B_	a structure that is a sized buffer where the size of the buffer is contained in a 16-bit, unsigned value The first parameter is the size in octets of the second parameter. The second parameter may be any type.
TPMA_	a structure where each of the fields defines an attribute and each field is usually a single bit All the attributes in an attribute structure are packed with the overall size of the structure indicated in the heading of the attribute description (UINT8, UINT16, or UINT32).
TPM_ALG_	an enumerated type that indicates an algorithm A TPM_ALG_ is often used as a selector for a union.
TPMI_	an interface type The value is specified for purposes of dynamic type checking when unmarshaled.
TPML_	a list length followed by the indicated number of entries of the indicated type This is an array with a length field.
TPMS_	a structure that is not a size buffer or a tagged buffer or a list
TPMT_	a structure with the first parameter being a structure tag, indicating the type of the structure that follows A structure tag may be either a TPM_ST_ or TPM_ALG_ depending on context.
TPMU_	a union of structures, lists, or unions If a union exists, there will normally be a companion TPMT_ that is the expression of the union in a tagged structure, where the tag is the selector indicating which member of the union is present.

 Copyright © TCG 2006-2020

Prefix	Description
TPM_xx_	an enumeration value of a particular type The value of “xx” will be indicative of the use of the enumerated type. A table of “TPM_xx” constant definitions will exist to define each of the TPM_xx_ values. EXAMPLE 1 TPM_CC_ indicates that the type is used for a *commandCode.* The allowed enumeration values will be found in the table defining the TPM_CC constants (Table 12) EXAMPLE 2 TPM_RC_ indicates that the type is used for a *responseCode*. The allowed enumeration values are in Table 16.

4.17 Data Alignment

The data structures in this TPM 2.0 Part 2 use octet alignment for all structures. When used in a table to indicate a maximum size, the `sizeof()` function returns the octet-aligned size of the structure, with no padding.

4.18 Parameter Unmarshaling Errors

The TPM commands are defined in TPM 2.0 Part 3. The command definition includes C code that details the actions performed by that command. The code is written assuming that the parameters of the command have been unmarshaled.

NOTE 1 An implementation is not required to process parameters in this manner or to separate the parameter parsing from the command actions. This method was chosen for the specification so that the normative behavior described by the detailed actions would be clear and unencumbered.

Unmarshaling is the process of processing the parameters in the input buffer and preparing the parameters for use by the command-specific action code. No data movement need take place but it is required that the TPM validate that the parameters meet the requirements of the expected data type as defined in this TPM 2.0 Part 2.

When an error is encountered while unmarshaling a command parameter, an error response code is returned and no command processing occurs. A table defining a data type may have response codes embedded in the table to indicate the error returned when the input value does not match the parameters of the table.

EXAMPLE 1

Table 12 has a listing of TPM command code values. The last row in the table contains "#TPM_RC_COMMAND_CODE" indicating the response code that is returned if the TPM is unmarshaling a value that it expects to be a TPM_CC and the input value is not in the table.

NOTE 2 In the reference implementation, a parameter number is added to the response code so that the offending parameter can be isolated.

 Copyright © TCG 2006-2020

In many cases, the table contains no specific response code value and the return code will be determined as defined in Table 2.

Table 2 — Unmarshaling Errors

Response code	Usage
TPM_RC_INSUFFICIENT	the input buffer did not contain enough octets to allow unmarshaling of the expected data type;
TPM_RC_RESERVED_BITS	a non-zero value was found in a reserved field of an attribute structure (TPMA_)
TPM_RC_SIZE	the value of a size parameter is larger or smaller than allowed
TPM_RC_VALUE	A parameter does not have one of its allowed values
TPM_RC_TAG	A parameter that should be a structure tag has a value that is not supported by the TPM

In some commands, a parameter may not be used because of various options of that command. However, the unmarshaling code is required to validate that all parameters have values that are allowed by the TPM 2.0 Part 2 definition of the parameter type even if that parameter is not used in the command actions.

 Copyright © TCG 2006-2020

5 Base Types

5.1 Primitive Types

The types listed in Table 3 are the primitive types on which all of the other types and structures are based. The values in the "Type" column should be edited for the compiler and computer on which the TPM is implemented. The values in the "Name" column should remain the same because these values are used in the remainder of the specification.

NOTE The types are compatible with the C99 standard and should be defined in stdint.h that is provided with a C99-compliant compiler;

The parameters in the Name column should remain in the order shown.

Table 3 — Definition of Base Types

Type	Name	Description
uint8_t	UINT8	unsigned, 8-bit integer
uint8_t	BYTE	unsigned 8-bit integer
int8_t	INT8	signed, 8-bit integer
int	BOOL	a bit in an `int` This is not used across the interface but is used in many places in the code. If the type were sent on the interface, it would have to have a type with a specific number of bytes.
uint16_t	UINT16	unsigned, 16-bit integer
int16_t	INT16	signed, 16-bit integer
uint32_t	UINT32	unsigned, 32-bit integer
int32_t	INT32	signed, 32-bit integer
uint64_t	UINT64	unsigned, 64-bit integer
int64_t	INT64	signed, 64-bit integer

5.2 Specification Logic Value Constants

Table 4 — Defines for Logic Values

Name	Value	Description
TRUE	1	
FALSE	0	
YES	1	
NO	0	
SET	1	
CLEAR	0	

 Copyright © TCG 2006-2020

5.3 Miscellaneous Types

These types are defined either for compatibility with previous versions of this specification or for clarity of this specification.

Table 5 — Definition of Types for Documentation Clarity

Type	Name	Description
UINT32	TPM_ALGORITHM_ID	this is the 1.2 compatible form of the TPM_ALG_ID
UINT32	TPM_MODIFIER_INDICATOR	
UINT32	TPM_AUTHORIZATION_SIZE	the *authorizationSize* parameter in a command
UINT32	TPM_PARAMETER_SIZE	the *parameterSize* parameter in a command
UINT16	TPM_KEY_SIZE	a key size in octets
UINT16	TPM_KEY_BITS	a key size in bits

 Copyright © TCG 2006-2020

6 Constants

6.1 TPM_SPEC (Specification Version Values)

These values are readable with TPM2_GetCapability() (see 6.13 for the format).

NOTE 1 This table will require editing when the specification is updated.

NOTE 2 The year and day of year are those of this specification if the TPM does not implement errata. If the TPM implements errata, the values indicate the release date of the errata document. There is no provision for indicating that not all errata are implemented.

Table 6 — Definition of (UINT32) TPM_SPEC Constants <>

Name	Value	Comments
TPM_SPEC_FAMILY	0x322E3000	ASCII “2.0” with null terminator
TPM_SPEC_LEVEL	00	the level number for the specification
TPM_SPEC_VERSION	159	the version number of the spec (001.59 * 100)
TPM_SPEC_YEAR	2019	the year of the version
TPM_SPEC_DAY_OF_YEAR	312	the day of the year (November 8)

6.2 TPM_GENERATED

This constant value differentiates TPM-generated structures from non-TPM structures.

Table 7 — Definition of (UINT32) TPM_GENERATED Constants <O>

Name	Value	Comments
TPM_GENERATED_VALUE	0xff544347	0xFF ‘TCG’ (FF 54 43 47_{16})

 Copyright © TCG 2006-2020

6.3 TPM_ALG_ID

The TCG maintains a registry of all algorithms that have an assigned algorithm ID. That registry is the definitive list of algorithms that may be supported by a TPM.

NOTE Inclusion of an algorithm does NOT indicate that the necessary claims of the algorithm are available under reasonable and non-discriminatory (RAND) terms from a TCG member.

Table 9 is an informative example of a TPM_ALG_ID constants table in the TCG Algorithm registry. Table 9 is provided for illustrative purposes only.

An algorithm ID is often used like a tag to determine the type of a structure in a context-sensitive way. The values for TPM_ALG_ID shall be in the range of 00 00_{16} – 7F FF_{16}. Other structure tags will be in the range 80 00_{16} – FF FF_{16}.

NOTE In TPM 1.2, these were defined as 32-bit constants. This specification limits the future size of the algorithm ID to 16 bits. The TPM_ALGORITHM_ID data type will continue to be a 32-bit number.

An algorithm shall not be assigned a value in the range 00 $C1_{16}$ – 00 $C6_{16}$ in order to prevent any overlap with the command structure tags used in TPM 1.2.

The implementation of some algorithms is dependent on the presence of other algorithms. When there is a dependency, the algorithm that is required is listed in column labeled "D" (dependent) in Table 9.

EXAMPLE Implementation of TPM_ALG_RSASSA requires that the RSA algorithm be implemented.

TPM_ALG_KEYEDHASH and TPM_ALG_NULL are required of all TPM implementations.

Table 8 — Legend for TPM_ALG_ID Table

Column Title	Comments
Algorithm Name	the mnemonic name assigned to the algorithm
Value	the numeric value assigned to the algorithm
Type	The allowed values are: **A** – asymmetric algorithm with a public and private key **S** – symmetric algorithm with only a private key **H** – hash algorithm that compresses input data to a digest value or indicates a method that uses a hash **X** – signing algorithm **N** – an anonymous signing algorithm **E** – an encryption algorithm **M** – a method such as a mask generation function **O** – an object type
C	(**C**lassification) The allowed values are: **A** – Assigned **S** – TCG Standard **L** – TCG Legacy
Dep	(**D**ependent) Indicates which other algorithm is required to be implemented if this algorithm is implemented
Reference	the reference document that defines the algorithm
Comments	clarifying information

 Copyright © TCG 2006-2020

Table 9 — Definition of (UINT16) TPM_ALG_ID Constants <IN/OUT, S>

Algorithm Name	Value	Type	Dep	C	Reference	Comments
TPM_ALG_ERROR	0x0000					should not occur
TPM_ALG_RSA	0x0001	A O		S	IETF RFC 8017	the RSA algorithm
TPM_ALG_TDES	0x0003	S		A	ISO/IEC 18033-3	block cipher with various key sizes (Triple Data Encryption Algorithm, commonly called Triple Data Encryption Standard)
TPM_ALG_SHA	0x0004	H		S	ISO/IEC 10118-3	the SHA1 algorithm
TPM_ALG_SHA1	0x0004	H		S	ISO/IEC 10118-3	redefinition for documentation consistency
TPM_ALG_HMAC	0x0005	H X		S	ISO/IEC 9797-2	Hash Message Authentication Code (HMAC) algorithm
TPM_ALG_AES	0x0006	S		S	ISO/IEC 18033-3	the AES algorithm with various key sizes
TPM_ALG_MGF1	0x0007	H M		S	IEEE Std 1363™-2000 IEEE Std 1363a™-2004	hash-based mask-generation function
TPM_ALG_KEYEDHASH	0x0008	H O		S	TCG TPM 2.0 library specification	an object type that may use XOR for encryption or an HMAC for signing and may also refer to a data object that is neither signing nor encrypting
TPM_ALG_XOR	0x000A	H S		S	TCG TPM 2.0 library specification	the XOR encryption algorithm
TPM_ALG_SHA256	0x000B	H		S	ISO/IEC 10118-3	the SHA 256 algorithm
TPM_ALG_SHA384	0x000C	H		A	ISO/IEC 10118-3	the SHA 384 algorithm
TPM_ALG_SHA512	0x000D	H		A	ISO/IEC 10118-3	the SHA 512 algorithm
TPM_ALG_NULL	0x0010			S	TCG TPM 2.0 library specification	Null algorithm
TPM_ALG_SM3_256	0x0012	H		A	GM/T 0004-2012	SM3 hash algorithm
TPM_ALG_SM4	0x0013	S		A	GM/T 0002-2012	SM4 symmetric block cipher
TPM_ALG_RSASSA	0x0014	A X	RSA	S	IETF RFC 8017	a signature algorithm defined in section 8.2 (RSASSA-PKCS1-v1_5)
TPM_ALG_RSAES	0x0015	A E	RSA	S	IETF RFC 8017	a padding algorithm defined in section 7.2 (RSAES-PKCS1-v1_5)
TPM_ALG_RSAPSS	0x0016	A X	RSA	S	IETF RFC 8017	a signature algorithm defined in section 8.1 (RSASSA-PSS)
TPM_ALG_OAEP	0x0017	A E H	RSA	S	IETF RFC 8017	a padding algorithm defined in section 7.1 (RSAES_OAEP)
TPM_ALG_ECDSA	0x0018	A X	ECC	S	ISO/IEC 14888-3	signature algorithm using elliptic curve cryptography (ECC)

 Copyright © TCG 2006-2020

Algorithm Name	Value	Type	Dep	C	Reference	Comments
TPM_ALG_ECDH	0x0019	A M	ECC	S	NIST SP800-56A	secret sharing using ECC Based on context, this can be either One-Pass Diffie-Hellman, C(1, 1, ECC CDH) defined in 6.2.2.2 or Full Unified Model C(2, 2, ECC CDH) defined in 6.1.1.2
TPM_ALG_ECDAA	0x001A	A X N	ECC	S	TCG TPM 2.0 library specification	elliptic-curve based, anonymous signing scheme
TPM_ALG_SM2	0x001B	A X	ECC	A	GM/T 0003.1–2012 GM/T 0003.2–2012 GM/T 0003.3–2012 GM/T 0003.5–2012	SM2 – depending on context, either an elliptic-curve based, signature algorithm or a key exchange protocol NOTE 1 Type listed as signing but, other uses are allowed according to context.
TPM_ALG_ECSCHNORR	0x001C	A X	ECC	S	TCG TPM 2.0 library specification	elliptic-curve based Schnorr signature
TPM_ALG_ECMQV	0x001D	A M	ECC	A	NIST SP800-56A	two-phase elliptic-curve key exchange – C(2, 2, ECC MQV) section 6.1.1.4
TPM_ALG_KDF1_SP800_56A	0x0020	H M	ECC	S	NIST SP800-56A	concatenation key derivation function (approved alternative 1) section 5.8.1
TPM_ALG_KDF2	0x0021	H M		A	IEEE Std 1363a-2004	key derivation function KDF2 section 13.2
TPM_ALG_KDF1_SP800_108	0x0022	H M		S	NIST SP800-108	a key derivation method Section 5.1 KDF in Counter Mode
TPM_ALG_ECC	0x0023	A O		S	ISO/IEC 15946-1	prime field ECC
TPM_ALG_SYMCIPHER	0x0025	O S		S	TCG TPM 2.0 library specification	the object type for a symmetric block cipher
TPM_ALG_CAMELLIA	0x0026	S		A	ISO/IEC 18033-3	Camellia is symmetric block cipher. The Camellia algorithm with various key sizes
TPM_ALG_SHA3_256	0x0027	H		A	NIST PUB FIPS 202	Hash algorithm producing a 256-bit digest
TPM_ALG_SHA3_384	0x0028	H		A	NIST PUB FIPS 202	Hash algorithm producing a 384-bit digest
TPM_ALG_SHA3_512	0x0029	H		A	NIST PUB FIPS 202	Hash algorithm producing a 512-bit digest
TPM_ALG_CTR	0x0040	S E		A	ISO/IEC 10116	Counter mode – if implemented, all symmetric block ciphers (S type) implemented shall be capable of using this mode.

 Copyright © TCG 2006-2020

Algorithm Name	Value	Type	Dep	C	Reference	Comments
TPM_ALG_OFB	0x0041	S E		A	ISO/IEC 10116	Output Feedback mode – if implemented, all symmetric block ciphers (S type) implemented shall be capable of using this mode.
TPM_ALG_CBC	0x0042	S E		A	ISO/IEC 10116	Cipher Block Chaining mode – if implemented, all symmetric block ciphers (S type) implemented shall be capable of using this mode.
TPM_ALG_CFB	0x0043	S E		S	ISO/IEC 10116	Cipher Feedback mode – if implemented, all symmetric block ciphers (S type) implemented shall be capable of using this mode.
TPM_ALG_ECB	0x0044	S E		A	ISO/IEC 10116	Electronic Codebook mode – if implemented, all symmetric block ciphers (S type) implemented shall be capable of using this mode. NOTE 2 This mode is not recommended for uses unless the key is frequently rotated such as in video codecs
reserved	0x00C1 through 0x00C6					0x00C1 – 0x00C6 are reserved to prevent any overlap with the command structure tags used in TPM 1.2
reserved	0x8000 through 0xFFFF					reserved for other structure tags

Copyright © TCG 2006-2020

6.4 TPM_ECC_CURVE

The TCG maintains a registry of all curves that have an assigned curve identifier. That registry is the definitive list of curves that may be supported by a TPM.

Table 10 is a copy of the TPM_ECC_CURVE constants table in the TCG registry as of the date of publication of this specification. Table 10 is provided for illustrative purposes only.

Table 10 — Definition of (UINT16) {ECC} TPM_ECC_CURVE Constants <IN/OUT>

Name	Value	Comments
+TPM_ECC_NONE	0x0000	
TPM_ECC_NIST_P192	0x0001	
TPM_ECC_NIST_P224	0x0002	
TPM_ECC_NIST_P256	0x0003	
TPM_ECC_NIST_P384	0x0004	
TPM_ECC_NIST_P521	0x0005	
TPM_ECC_BN_P256	0x0010	curve to support ECDAA
TPM_ECC_BN_P638	0x0011	curve to support ECDAA
TPM_ECC_SM2_P256	0x0020	
#TPM_RC_CURVE		

6.5 TPM_CC (Command Codes)

6.5.1 Format

A command is a 32-bit structure with fields assigned as shown in Figure 1. If V is SET, the command is vendor specific. If V is CLEAR, the command is not vendor specific.

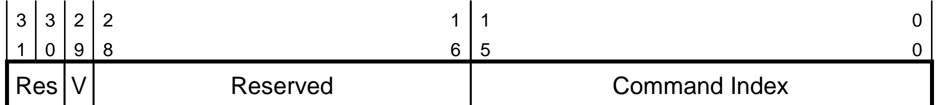


Figure 1 — Command Format

Table 11 — TPM Command Format Fields Description

Bit	Name	Definition
15:0	Command Index	the index of the command
28:16	Reserved	shall be zero
29	V	vendor specific
31:30	Res	shall be zero

 Copyright © TCG 2006-2020

6.5.2 TPM_CC Listing

Table 12 lists the command codes assigned to each command name. The Dep column indicates whether the command has a dependency on the implementation of a specific algorithm.

Table 12 — Definition of (UINT32) TPM_CC Constants (Numeric Order) <IN/OUT, S>

Name	Command Code	Dep	Comments
TPM_CC_FIRST	0x0000011F		Compile variable. May decrease based on implementation.
TPM_CC_NV_UndefineSpaceSpecial	0x0000011F		
TPM_CC_EvictControl	0x00000120		
TPM_CC_HierarchyControl	0x00000121		
TPM_CC_NV_UndefineSpace	0x00000122		
TPM_CC_ChangeEPS	0x00000124		
TPM_CC_ChangePPS	0x00000125		
TPM_CC_Clear	0x00000126		
TPM_CC_ClearControl	0x00000127		
TPM_CC_ClockSet	0x00000128		
TPM_CC_HierarchyChangeAuth	0x00000129		
TPM_CC_NV_DefineSpace	0x0000012A		
TPM_CC_PCR_Allocate	0x0000012B		
TPM_CC_PCR_SetAuthPolicy	0x0000012C		
TPM_CC_PP_Commands	0x0000012D		
TPM_CC_SetPrimaryPolicy	0x0000012E		
TPM_CC_FieldUpgradeStart	0x0000012F		
TPM_CC_ClockRateAdjust	0x00000130		
TPM_CC_CreatePrimary	0x00000131		
TPM_CC_NV_GlobalWriteLock	0x00000132		
TPM_CC_GetCommandAuditDigest	0x00000133		
TPM_CC_NV_Increment	0x00000134		
TPM_CC_NV_SetBits	0x00000135		
TPM_CC_NV_Extend	0x00000136		
TPM_CC_NV_Write	0x00000137		
TPM_CC_NV_WriteLock	0x00000138		
TPM_CC_DictionaryAttackLockReset	0x00000139		
TPM_CC_DictionaryAttackParameters	0x0000013A		
TPM_CC_NV_ChangeAuth	0x0000013B		

 Copyright © TCG 2006-2020

Name	Command Code	Dep	Comments
TPM_CC_PCR_Event	0x0000013C		PCR
TPM_CC_PCR_Reset	0x0000013D		PCR
TPM_CC_SequenceComplete	0x0000013E		
TPM_CC_SetAlgorithmSet	0x0000013F		
TPM_CC_SetCommandCodeAuditStatus	0x00000140		
TPM_CC_FieldUpgradeData	0x00000141		
TPM_CC_IncrementalSelfTest	0x00000142		
TPM_CC_SelfTest	0x00000143		
TPM_CC_Startup	0x00000144		
TPM_CC_Shutdown	0x00000145		
TPM_CC_StirRandom	0x00000146		
TPM_CC_ActivateCredential	0x00000147		
TPM_CC_Certify	0x00000148		
TPM_CC_PolicyNV	0x00000149		Policy
TPM_CC_CertifyCreation	0x0000014A		
TPM_CC_Duplicate	0x0000014B		
TPM_CC_GetTime	0x0000014C		
TPM_CC_GetSessionAuditDigest	0x0000014D		
TPM_CC_NV_Read	0x0000014E		
TPM_CC_NV_ReadLock	0x0000014F		
TPM_CC_ObjectChangeAuth	0x00000150		
TPM_CC_PolicySecret	0x00000151		Policy
TPM_CC_Rewrap	0x00000152		
TPM_CC_Create	0x00000153		
TPM_CC_ECDH_ZGen	0x00000154	ECC	
TPM_CC_HMAC	0x00000155	!CMAC	See NOTE 1
TPM_CC_MAC	0x00000155	CMAC	See NOTE 1
TPM_CC_Import	0x00000156		
TPM_CC_Load	0x00000157		
TPM_CC_Quote	0x00000158		
TPM_CC_RSA_Decrypt	0x00000159	RSA	
TPM_CC_HMAC_Start	0x0000015B	!CMAC	See NOTE 1
TPM_CC_MAC_Start	0x0000015B	CMAC	See NOTE 1
TPM_CC_SequenceUpdate	0x0000015C		
TPM_CC_Sign	0x0000015D		

 Copyright © TCG 2006-2020

Name	Command Code	Dep	Comments
TPM_CC_Unseal	0x0000015E		
TPM_CC_PolicySigned	0x00000160		Policy
TPM_CC_ContextLoad	0x00000161		Context
TPM_CC_ContextSave	0x00000162		Context
TPM_CC_ECDH_KeyGen	0x00000163	ECC	
TPM_CC_EncryptDecrypt	0x00000164		
TPM_CC_FlushContext	0x00000165		Context
TPM_CC_LoadExternal	0x00000167		
TPM_CC_MakeCredential	0x00000168		
TPM_CC_NV_ReadPublic	0x00000169		NV
TPM_CC_PolicyAuthorize	0x0000016A		Policy
TPM_CC_PolicyAuthValue	0x0000016B		Policy
TPM_CC_PolicyCommandCode	0x0000016C		Policy
TPM_CC_PolicyCounterTimer	0x0000016D		Policy
TPM_CC_PolicyCpHash	0x0000016E		Policy
TPM_CC_PolicyLocality	0x0000016F		Policy
TPM_CC_PolicyNameHash	0x00000170		Policy
TPM_CC_PolicyOR	0x00000171		Policy
TPM_CC_PolicyTicket	0x00000172		Policy
TPM_CC_ReadPublic	0x00000173		
TPM_CC_RSA_Encrypt	0x00000174	RSA	
TPM_CC_StartAuthSession	0x00000176		
TPM_CC_VerifySignature	0x00000177		
TPM_CC_ECC_Parameters	0x00000178	ECC	
TPM_CC_FirmwareRead	0x00000179		
TPM_CC_GetCapability	0x0000017A		
TPM_CC_GetRandom	0x0000017B		
TPM_CC_GetTestResult	0x0000017C		
TPM_CC_Hash	0x0000017D		
TPM_CC_PCR_Read	0x0000017E		PCR
TPM_CC_PolicyPCR	0x0000017F		Policy
TPM_CC_PolicyRestart	0x00000180		
TPM_CC_ReadClock	0x00000181		
TPM_CC_PCR_Extend	0x00000182		
TPM_CC_PCR_SetAuthValue	0x00000183		

 Copyright © TCG 2006-2020

Name	Command Code	Dep	Comments
TPM_CC_NV_Certify	0x00000184		
TPM_CC_EventSequenceComplete	0x00000185		
TPM_CC_HashSequenceStart	0x00000186		
TPM_CC_PolicyPhysicalPresence	0x00000187		Policy
TPM_CC_PolicyDuplicationSelect	0x00000188		Policy
TPM_CC_PolicyGetDigest	0x00000189		Policy
TPM_CC_TestParms	0x0000018A		
TPM_CC_Commit	0x0000018B	ECC	
TPM_CC_PolicyPassword	0x0000018C		Policy
TPM_CC_ZGen_2Phase	0x0000018D	ECC	
TPM_CC_EC_Ephemeral	0x0000018E	ECC	
TPM_CC_PolicyNvWritten	0x0000018F		Policy
TPM_CC_PolicyTemplate	0x00000190		Policy
TPM_CC_CreateLoaded	0x00000191		
TPM_CC_PolicyAuthorizeNV	0x00000192		Policy
TPM_CC_EncryptDecrypt2	0x00000193		
TPM_CC_AC_GetCapability	0x00000194		
TPM_CC_AC_Send	0x00000195		
TPM_CC_Policy_AC_SendSelect	0x00000196		Policy
TPM_CC_CertifyX509	0x00000197		
TPM_CC_ACT_SetTimeout	0x00000198		
TPM_CC_LAST	0x00000198		Compile variable. May increase based on implementation.
CC_VEND	0x20000000		
TPM_CC_Vendor_TCG_Test	CC_VEND+0x0000		Used for testing of command dispatch
#TPM_RC_COMMAND_CODE			

NOTE 1 A TPM may implement either TPM2_HMAC()/TPM2_HMAC_Start() or TPM2_MAC()/TPM2_MAC_Start() but not both, as they have the same command code and there is no way to distinguish them. A TPM that supports TPM2_MAC()/TPM2_MAC_Start() will support any code that was written to use TPM2_HMAC()/TPM2_HMAC_Start(), but a TPM that supports TPM2_HMAC()/TPM2_HMAC_Start() will not support a MAC based on symmetric block ciphers.

 Copyright © TCG 2006-2020

6.6 TPM_RC (Response Codes)

6.6.1 Description

Each return from the TPM has a 32-bit response code. The TPM will always set the upper 20 bits (31:12) of the response code to 0 00 00_{16} and the low-order 12 bits (11:00) will contain the response code.

When a command succeeds, the TPM shall return TPM_RC_SUCCESS (0 00_{16}) and will update any authorization-session nonce associated with the command.

When a command fails to complete for any reason, the TPM shall return

- a TPM_ST (UINT16) with a value of TPM_TAG_RSP_COMMAND or TPM_ST_NO_SESSIONS, followed by
- a UINT32 (*responseSize*) with a value of 10, followed by
- a UINT32 containing a response code with a value other than TPM_RC_SUCCESS.

Commands defined in this specification will use a tag of either TPM_ST_NO_SESSIONS or TPM_ST_SESSIONS. Error responses will use a tag value of TPM_ST_NO_SESSIONS and the response code will be as defined in this specification. Commands that use tags defined in the TPM 1.2 specification will use TPM_TAG_RSP_COMMAND in an error and a response code defined in TPM 1.2.

If the tag of the command is not a recognized command tag, the TPM error response will differ depending on TPM 1.2 compatibility. If the TPM supports 1.2 compatibility, the TPM shall return a tag of TPM_TAG_RSP_COMMAND and an appropriate TPM 1.2 response code (TPM_BADTAG = 00 00 00 $1E_{16}$). If the TPM does not have compatibility with TPM 1.2, the TPM shall return TPM_ST_NO_SESSION and a response code of TPM_RC_TAG.

When a command fails, the TPM shall not update the authorization-session nonces associated with the command and will not close the authorization sessions used by the command. Audit digests will not be updated on an error. Unless noted in the command actions, a command that returns an error shall leave the state of the TPM as if the command had not been attempted. The exception to this principle is that a failure due to an authorization failure may update the dictionary-attack protection values.

6.6.2 Response Code Formats

The response codes for this specification are defined such that there is no overlap between the response codes used for this specification and those assigned in previous TPM specifications.

The formats defined in this clause only apply when the tag for the response is TPM_ST_NO_SESSIONS.

The response codes use two different format groups. One group contains the TPM 1.2 compatible response codes and the response codes for this specification that are not related to command parameters. The second group contains the errors that may be associated with a command parameter, handle, or session.

 Copyright © TCG 2006-2020

Figure 2 shows the format for the response codes when bit 7 is zero.

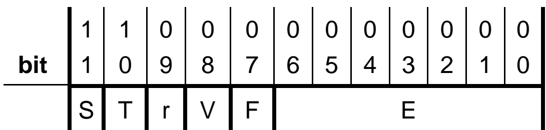


Figure 2 — Format-Zero Response Codes

The field definitions are:

Table 13 — Format-Zero Response Codes

Bit	Name	Definition
06:00	E	the error number The interpretation of this field is dependent on the setting of the F and S fields.
07	F	format selector **CLEAR** when the format is as defined in this Table 13 or when the response code is TPM_RC_BAD_TAG.
08	V	version **SET (1)**: The error number is defined in this specification and is returned when the response tag is TPM_ST_NO_SESSIONS. **CLEAR (0)**: The error number is defined by a previous TPM specification. The error number is returned when the response tag is TPM_TAG_RSP_COMMAND. NOTE In any error number returned by a TPM, the F (bit 7) and V (bit 8) attributes shall be CLEAR when the response tag is TPM_TAG_RSP_COMMAND value used in TPM 1.2.
09	Reserved	shall be zero.
10	T	TCG/Vendor indicator **SET (1):** The response code is defined by the TPM vendor. **CLEAR (0):** The response code is defined by the TCG (a value in this specification). NOTE This attribute does not indicate a vendor-specific code unless the *F* attribute (bit[07]) is CLEAR.
11	S	severity **SET (1)**: The response code is a warning and the command was not necessarily in error. This command indicates that the TPM is busy or that the resources of the TPM have to be adjusted in order to allow the command to execute. **CLEAR (0)**: The response code indicates that the command had an error that would prevent it from running.

When the format bit (bit 7) is SET, then the error occurred during the unmarshaling or validation of an input parameter to the TPM. Figure 3 shows the format for the response codes when bit 7 is one.

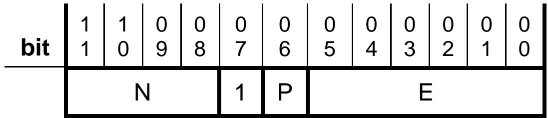


Figure 3 — Format-One Response Codes

There are 64 errors with this format. The errors can be associated with a parameter, handle, or session. The error number for this format is in bits[05:00]. When an error is associated with a parameter, TPM_RC_P (0 40_{16}) is added and N is set to the parameter number.

NOTE 1 In the reference implementation, for a RC_FMT1 response code, a constant of the form RC_Command_parameterName is the one based parameter number (TPM_RC_n) plus TPM_RC_P.

 Copyright © TCG 2006-2020

Example RC_Startup_startupType is the first parameter, TPM_RC_1 (0x100) plus TPM_RC_P (0x40) or 0x140. TPM_RC_VALUE (RC_FMT1 (0x080) + 0x004) + RC_Startup_startupType is thus 0x080 + 0x004 + 0x140 = 0x1c4.

For an error associated with a handle, a parameter number (1 to 7) is added to the N field. For an error associated with a session, a value of 8 plus the session number (1 to 7) is added to the N field. In other words, if P is clear, then a value of 0 to 7 in the N field will indicate a handle error, and a value of 8 – 15 will indicate a session error.

NOTE 2 If an implementation is not able to designate the handle, session, or parameter in error, then P and N will be zero.

The field definitions are:

Table 14 — Format-One Response Codes

Bit	Name	Definition
05:00	E	the error number The error number is independent of the other settings.
06	P	**SET (1):** The error is associated with a parameter. **CLEAR (0):** The error is associated with a handle or a session.
07	F	the response code format selector This field shall be SET for the format in this table.
11:08	N	the number of the handle, session, or parameter in error. The number is one based. See TPM_RC_1 through TPM_RC_F. If P is SET, then this field is the parameter in error. If P is CLEAR, then this field indicates the handle or session in error. Handles use values of N between 0000_2 and 0111_2. Sessions use values between 1000_2 and 1111_2. NOTE Bit 11 distinguishes between handles and sessions. Bits 10:8 000_2 indicate that the number is unspecified.

The groupings of response codes are determined by bits 08, 07, and 06 of the response code as summarized in Table 15.

Table 15 — Response Code Groupings

Bit			
08	07	06	Definition
0	0	x	a response code defined by TPM 1.2 NOTE An “x” in a column indicates that this may be either 0 or 1 and not affect the grouping of the response code.
1	0	x	a response code defined by this specification with no handle, session, or parameter number modifier
x	1	0	a response code defined by this specification with either a handle or session number modifier
x	1	1	a response code defined by this specification with a parameter number modifier

 Copyright © TCG 2006-2020

6.6.3 TPM_RC Values

In general, response codes defined in TPM 2.0 Part 2 will be unmarshaling errors and will have the F (format) bit SET. Codes that are unique to TPM 2.0 Part 3 will have the F bit CLEAR but the V (version) attribute will be SET to indicate that it is a TPM 2.0 response code. See *Response Code Details* in TPM 2.0 Part 1.

NOTE The constant RC_VER1 is used to indicate that the V attribute is SET and the constant RC_FMT1 is used to indicate that the F attribute is SET and that the return code is variable based on handle, session, and parameter modifiers.

Table 16 — Definition of (UINT32) TPM_RC Constants (Actions) <OUT>

Name	Value	Description
TPM_RC_SUCCESS	0x000	
TPM_RC_BAD_TAG	0x01E	defined for compatibility with TPM 1.2
RC_VER1	0x100	set for all format 0 response codes
TPM_RC_INITIALIZE	RC_VER1 + 0x000	TPM not initialized by TPM2_Startup or already initialized
TPM_RC_FAILURE	RC_VER1 + 0x001	commands not being accepted because of a TPM failure NOTE This may be returned by TPM2_GetTestResult() as the *testResult* parameter.
TPM_RC_SEQUENCE	RC_VER1 + 0x003	improper use of a sequence handle
TPM_RC_PRIVATE	RC_VER1 + 0x00B	not currently used
TPM_RC_HMAC	RC_VER1 + 0x019	not currently used
TPM_RC_DISABLED	RC_VER1 + 0x020	the command is disabled
TPM_RC_EXCLUSIVE	RC_VER1 + 0x021	command failed because audit sequence required exclusivity
TPM_RC_AUTH_TYPE	RC_VER1 + 0x024	authorization handle is not correct for command
TPM_RC_AUTH_MISSING	RC_VER1 + 0x025	command requires an authorization session for handle and it is not present.
TPM_RC_POLICY	RC_VER1 + 0x026	policy failure in math operation or an invalid authPolicy value
TPM_RC_PCR	RC_VER1 + 0x027	PCR check fail
TPM_RC_PCR_CHANGED	RC_VER1 + 0x028	PCR have changed since checked.
TPM_RC_UPGRADE	RC_VER1 + 0x02D	for all commands other than TPM2_FieldUpgradeData(), this code indicates that the TPM is in field upgrade mode; for TPM2_FieldUpgradeData(), this code indicates that the TPM is not in field upgrade mode
TPM_RC_TOO_MANY_CONTEXTS	RC_VER1 + 0x02E	context ID counter is at maximum.
TPM_RC_AUTH_UNAVAILABLE	RC_VER1 + 0x02F	authValue or authPolicy is not available for selected entity.
TPM_RC_REBOOT	RC_VER1 + 0x030	a _TPM_Init and Startup(CLEAR) is required before the TPM can resume operation.
TPM_RC_UNBALANCED	RC_VER1 + 0x031	the protection algorithms (hash and symmetric) are not reasonably balanced. The digest size of the hash must be larger than the key size of the symmetric algorithm.

 Copyright © TCG 2006-2020

Name	Value	Description
TPM_RC_COMMAND_SIZE	RC_VER1 + 0x042	command *commandSize* value is inconsistent with contents of the command buffer; either the size is not the same as the octets loaded by the hardware interface layer or the value is not large enough to hold a command header
TPM_RC_COMMAND_CODE	RC_VER1 + 0x043	command code not supported
TPM_RC_AUTHSIZE	RC_VER1 + 0x044	the value of *authorizationSize* is out of range or the number of octets in the Authorization Area is greater than required
TPM_RC_AUTH_CONTEXT	RC_VER1 + 0x045	use of an authorization session with a context command or another command that cannot have an authorization session.
TPM_RC_NV_RANGE	RC_VER1 + 0x046	NV offset+size is out of range.
TPM_RC_NV_SIZE	RC_VER1 + 0x047	Requested allocation size is larger than allowed.
TPM_RC_NV_LOCKED	RC_VER1 + 0x048	NV access locked.
TPM_RC_NV_AUTHORIZATION	RC_VER1 + 0x049	NV access authorization fails in command actions (this failure does not affect lockout.action)
TPM_RC_NV_UNINITIALIZED	RC_VER1 + 0x04A	an NV Index is used before being initialized or the state saved by TPM2_Shutdown(STATE) could not be restored
TPM_RC_NV_SPACE	RC_VER1 + 0x04B	insufficient space for NV allocation
TPM_RC_NV_DEFINED	RC_VER1 + 0x04C	NV Index or persistent object already defined
TPM_RC_BAD_CONTEXT	RC_VER1 + 0x050	context in TPM2_ContextLoad() is not valid
TPM_RC_CPHASH	RC_VER1 + 0x051	cpHash value already set or not correct for use
TPM_RC_PARENT	RC_VER1 + 0x052	handle for parent is not a valid parent
TPM_RC_NEEDS_TEST	RC_VER1 + 0x053	some function needs testing.
TPM_RC_NO_RESULT	RC_VER1 + 0x054	returned when an internal function cannot process a request due to an unspecified problem. This code is usually related to invalid parameters that are not properly filtered by the input unmarshaling code.
TPM_RC_SENSITIVE	RC_VER1 + 0x055	the sensitive area did not unmarshal correctly after decryption – this code is used in lieu of the other unmarshaling errors so that an attacker cannot determine where the unmarshaling error occurred
RC_MAX_FM0	RC_VER1 + 0x07F	largest version 1 code that is not a warning
		New Subsection
RC_FMT1	0x080	This bit is SET in all format 1 response codes The codes in this group may have a value added to them to indicate the handle, session, or parameter to which they apply.
TPM_RC_ASYMMETRIC	RC_FMT1 + 0x001	asymmetric algorithm not supported or not correct
TPM_RC_ATTRIBUTES	RC_FMT1 + 0x002	inconsistent attributes
TPM_RC_HASH	RC_FMT1 + 0x003	hash algorithm not supported or not appropriate
TPM_RC_VALUE	RC_FMT1 + 0x004	value is out of range or is not correct for the context

 Copyright © TCG 2006-2020

Name	Value	Description
TPM_RC_HIERARCHY	RC_FMT1 + 0x005	hierarchy is not enabled or is not correct for the use
TPM_RC_KEY_SIZE	RC_FMT1 + 0x007	key size is not supported
TPM_RC_MGF	RC_FMT1 + 0x008	mask generation function not supported
TPM_RC_MODE	RC_FMT1 + 0x009	mode of operation not supported
TPM_RC_TYPE	RC_FMT1 + 0x00A	the type of the value is not appropriate for the use
TPM_RC_HANDLE	RC_FMT1 + 0x00B	the handle is not correct for the use
TPM_RC_KDF	RC_FMT1 + 0x00C	unsupported key derivation function or function not appropriate for use
TPM_RC_RANGE	RC_FMT1 + 0x00D	value was out of allowed range.
TPM_RC_AUTH_FAIL	RC_FMT1 + 0x00E	the authorization HMAC check failed and DA counter incremented
TPM_RC_NONCE	RC_FMT1 + 0x00F	invalid nonce size or nonce value mismatch
TPM_RC_PP	RC_FMT1 + 0x010	authorization requires assertion of PP
TPM_RC_SCHEME	RC_FMT1 + 0x012	unsupported or incompatible scheme
TPM_RC_SIZE	RC_FMT1 + 0x015	structure is the wrong size
TPM_RC_SYMMETRIC	RC_FMT1 + 0x016	unsupported symmetric algorithm or key size, or not appropriate for instance
TPM_RC_TAG	RC_FMT1 + 0x017	incorrect structure tag
TPM_RC_SELECTOR	RC_FMT1 + 0x018	union selector is incorrect
TPM_RC_INSUFFICIENT	RC_FMT1 + 0x01A	the TPM was unable to unmarshal a value because there were not enough octets in the input buffer
TPM_RC_SIGNATURE	RC_FMT1 + 0x01B	the signature is not valid
TPM_RC_KEY	RC_FMT1 + 0x01C	key fields are not compatible with the selected use
TPM_RC_POLICY_FAIL	RC_FMT1 + 0x01D	a policy check failed
TPM_RC_INTEGRITY	RC_FMT1 + 0x01F	integrity check failed
TPM_RC_TICKET	RC_FMT1 + 0x020	invalid ticket
TPM_RC_RESERVED_BITS	RC_FMT1 + 0x021	reserved bits not set to zero as required
TPM_RC_BAD_AUTH	RC_FMT1 + 0x022	authorization failure without DA implications
TPM_RC_EXPIRED	RC_FMT1 + 0x023	the policy has expired
TPM_RC_POLICY_CC	RC_FMT1 + 0x024	the *commandCode* in the policy is not the *commandCode* of the command or the command code in a policy command references a command that is not implemented
TPM_RC_BINDING	RC_FMT1 + 0x025	public and sensitive portions of an object are not cryptographically bound
TPM_RC_CURVE	RC_FMT1 + 0x026	curve not supported
TPM_RC_ECC_POINT	RC_FMT1 + 0x027	point is not on the required curve.
		New Subsection
RC_WARN	0x900	set for warning response codes

 Copyright © TCG 2006-2020

Name	Value	Description
TPM_RC_CONTEXT_GAP	RC_WARN + 0x001	gap for context ID is too large
TPM_RC_OBJECT_MEMORY	RC_WARN + 0x002	out of memory for object contexts
TPM_RC_SESSION_MEMORY	RC_WARN + 0x003	out of memory for session contexts
TPM_RC_MEMORY	RC_WARN + 0x004	out of shared object/session memory or need space for internal operations
TPM_RC_SESSION_HANDLES	RC_WARN + 0x005	out of session handles – a session must be flushed before a new session may be created
TPM_RC_OBJECT_HANDLES	RC_WARN + 0x006	out of object handles – the handle space for objects is depleted and a reboot is required NOTE 1 This cannot occur on the reference implementation. NOTE 2 There is no reason why an implementation would implement a design that would deplete handle space. Platform specifications are encouraged to forbid it.
TPM_RC_LOCALITY	RC_WARN + 0x007	bad locality
TPM_RC_YIELDED	RC_WARN + 0x008	the TPM has suspended operation on the command; forward progress was made and the command may be retried See TPM 2.0 Part 1, “Multi-tasking.” NOTE This cannot occur on the reference implementation.
TPM_RC_CANCELED	RC_WARN + 0x009	the command was canceled
TPM_RC_TESTING	RC_WARN + 0x00A	TPM is performing self-tests
TPM_RC_REFERENCE_H0	RC_WARN + 0x010	the 1st handle in the handle area references a transient object or session that is not loaded
TPM_RC_REFERENCE_H1	RC_WARN + 0x011	the 2nd handle in the handle area references a transient object or session that is not loaded
TPM_RC_REFERENCE_H2	RC_WARN + 0x012	the 3rd handle in the handle area references a transient object or session that is not loaded
TPM_RC_REFERENCE_H3	RC_WARN + 0x013	the 4th handle in the handle area references a transient object or session that is not loaded
TPM_RC_REFERENCE_H4	RC_WARN + 0x014	the 5th handle in the handle area references a transient object or session that is not loaded
TPM_RC_REFERENCE_H5	RC_WARN + 0x015	the 6th handle in the handle area references a transient object or session that is not loaded
TPM_RC_REFERENCE_H6	RC_WARN + 0x016	the 7th handle in the handle area references a transient object or session that is not loaded
TPM_RC_REFERENCE_S0	RC_WARN + 0x018	the 1st authorization session handle references a session that is not loaded
TPM_RC_REFERENCE_S1	RC_WARN + 0x019	the 2nd authorization session handle references a session that is not loaded
TPM_RC_REFERENCE_S2	RC_WARN + 0x01A	the 3rd authorization session handle references a session that is not loaded
TPM_RC_REFERENCE_S3	RC_WARN + 0x01B	the 4th authorization session handle references a session that is not loaded
TPM_RC_REFERENCE_S4	RC_WARN + 0x01C	the 5th session handle references a session that is not loaded

 Copyright © TCG 2006-2020

Name	Value	Description
TPM_RC_REFERENCE_S5	RC_WARN + 0x01D	the 6th session handle references a session that is not loaded
TPM_RC_REFERENCE_S6	RC_WARN + 0x01E	the 7th authorization session handle references a session that is not loaded
TPM_RC_NV_RATE	RC_WARN + 0x020	the TPM is rate-limiting accesses to prevent wearout of NV
TPM_RC_LOCKOUT	RC_WARN + 0x021	authorizations for objects subject to DA protection are not allowed at this time because the TPM is in DA lockout mode
TPM_RC_RETRY	RC_WARN + 0x022	the TPM was not able to start the command
TPM_RC_NV_UNAVAILABLE	RC_WARN + 0x023	the command may require writing of NV and NV is not current accessible
TPM_RC_NOT_USED	RC_WARN + 0x7F	this value is reserved and shall not be returned by the TPM
		Additional Defines
TPM_RC_H	0x000	add to a handle-related error
TPM_RC_P	0x040	add to a parameter-related error
TPM_RC_S	0x800	add to a session-related error
TPM_RC_1	0x100	add to a parameter-, handle-, or session-related error
TPM_RC_2	0x200	add to a parameter-, handle-, or session-related error
TPM_RC_3	0x300	add to a parameter-, handle-, or session-related error
TPM_RC_4	0x400	add to a parameter-, handle-, or session-related error
TPM_RC_5	0x500	add to a parameter-, handle-, or session-related error
TPM_RC_6	0x600	add to a parameter-, handle-, or session-related error
TPM_RC_7	0x700	add to a parameter-, handle-, or session-related error
TPM_RC_8	0x800	add to a parameter-related error
TPM_RC_9	0x900	add to a parameter-related error
TPM_RC_A	0xA00	add to a parameter-related error
TPM_RC_B	0xB00	add to a parameter-related error
TPM_RC_C	0xC00	add to a parameter-related error
TPM_RC_D	0xD00	add to a parameter-related error
TPM_RC_E	0xE00	add to a parameter-related error
TPM_RC_F	0xF00	add to a parameter-related error
TPM_RC_N_MASK	0xF00	number mask

Copyright © TCG 2006-2020

6.7 TPM_CLOCK_ADJUST

A TPM_CLOCK_ADJUST value is used to change the rate at which the TPM internal oscillator is divided. A change to the divider will change the rate at which *Clock* and *Time* change.

NOTE The recommended adjustments are approximately 1% for a course adjustment, 0.1% for a medium adjustment, and the minimum possible on the implementation for the fine adjustment (e.g., one count of the pre-scalar if possible).

Table 17 — Definition of (INT8) TPM_CLOCK_ADJUST Constants <IN>

Name	Value	Comments
TPM_CLOCK_COARSE_SLOWER	-3	Slow the *Clock* update rate by one coarse adjustment step.
TPM_CLOCK_MEDIUM_SLOWER	-2	Slow the *Clock* update rate by one medium adjustment step.
TPM_CLOCK_FINE_SLOWER	-1	Slow the *Clock* update rate by one fine adjustment step.
TPM_CLOCK_NO_CHANGE	0	No change to the *Clock* update rate.
TPM_CLOCK_FINE_FASTER	1	Speed the *Clock* update rate by one fine adjustment step.
TPM_CLOCK_MEDIUM_FASTER	2	Speed the *Clock* update rate by one medium adjustment step.
TPM_CLOCK_COARSE_FASTER	3	Speed the *Clock* update rate by one coarse adjustment step.
#TPM_RC_VALUE		

6.8 TPM_EO (EA Arithmetic Operands)

Table 18 — Definition of (UINT16) TPM_EO Constants <IN/OUT>

Operation Name	Value	Comments
TPM_EO_EQ	0x0000	A = B
TPM_EO_NEQ	0x0001	A ≠ B
TPM_EO_SIGNED_GT	0x0002	A > B signed
TPM_EO_UNSIGNED_GT	0x0003	A > B unsigned
TPM_EO_SIGNED_LT	0x0004	A < B signed
TPM_EO_UNSIGNED_LT	0x0005	A < B unsigned
TPM_EO_SIGNED_GE	0x0006	A ≥ B signed
TPM_EO_UNSIGNED_GE	0x0007	A ≥ B unsigned
TPM_EO_SIGNED_LE	0x0008	A ≤ B signed
TPM_EO_UNSIGNED_LE	0x0009	A ≤ B unsigned
TPM_EO_BITSET	0x000A	All bits SET in B are SET in A. ((A&B)=B)
TPM_EO_BITCLEAR	0x000B	All bits SET in B are CLEAR in A. ((A&B)=0)
#TPM_RC_VALUE		Response code returned when unmarshaling of this type fails

 Copyright © TCG 2006-2020

6.9 TPM_ST (Structure Tags)

Structure tags are used to disambiguate structures. They are 16-bit values with the most significant bit SET so that they do not overlap TPM_ALG_ID values. A single exception is made for the value associated with TPM_ST_RSP_COMMAND (0x00C4), which has the same value as the TPM_TAG_RSP_COMMAND tag from earlier versions of this specification. This value is used when the TPM is compatible with a previous TPM specification and the TPM cannot determine which family of response code to return because the command tag is not valid.

Many of the structures defined in this document have parameters that are unions of other structures. That is, a parameter may be one of several structures. The parameter will have a selector value that indicates which of the options is actually present.

In order to allow the marshaling and unmarshaling code to determine which of the possible structures is allowed, each selector will have a unique interface type and will constrain the number of possible tag values.

Table 19 defines the structure tags values. The definition of many structures is context-sensitive using an algorithm ID. In cases where an algorithm ID is not a meaningful way to designate the structure, the values in this table are used.

 Copyright © TCG 2006-2020

Table 19 — Definition of (UINT16) TPM_ST Constants <IN/OUT, S>

Name	Value	Comments
TPM_ST_RSP_COMMAND	0x00C4	*tag* value for a response; used when there is an error in the tag. This is also the value returned from a TPM 1.2 when an error occurs. This value is used in this specification because an error in the command tag may prevent determination of the family. When this tag is used in the response, the response code will be TPM_RC_BAD_TAG (0 $1E_{16}$), which has the same numeric value as the TPM 1.2 response code for TPM_BADTAG. NOTE In a previously published version of this specification, TPM_RC_BAD_TAG was incorrectly assigned a value of 0x030 instead of 30 (0x01e). Some implementations my return the old value instead of the new value.
TPM_ST_NULL	0X8000	no structure type specified
TPM_ST_NO_SESSIONS	0x8001	*tag* value for a command/response for a command defined in this specification; indicating that the command/response has no attached sessions and no *authorizationSize*/*parameterSize* value is present If the *responseCode* from the TPM is not TPM_RC_SUCCESS, then the response tag shall have this value.
TPM_ST_SESSIONS	0x8002	*tag* value for a command/response for a command defined in this specification; indicating that the command/response has one or more attached sessions and the *authorizationSize*/*parameterSize* field is present
reserved	0x8003	When used between application software and the TPM resource manager, this tag indicates that the command has no sessions and the handles are using the Name format rather than the 32-bit handle format. NOTE 1 The response to application software will have a *tag* of TPM_ST_NO_SESSIONS. Between the TRM and TPM, this tag would occur in a response from a TPM that overlaps the *tag* parameter of a request with the *tag* parameter of a response, when the response has no associated sessions. NOTE 2 This tag is not used by all TPM or TRM implementations.
reserved	0x8004	When used between application software and the TPM resource manager, this tag indicates that the command has sessions and the handles are using the Name format rather than the 32-bit handle format. NOTE 1 If the command completes successfully, the response to application software will have a *tag* of TPM_ST_SESSIONS. Between the TRM and TPM, would occur in a response from a TPM that overlaps the *tag* parameter of a request with the *tag* parameter of a response, when the response has authorization sessions. NOTE 2 This tag is not used by all TPM or TRM implementations.
TPM_ST_ATTEST_NV	0x8014	tag for an attestation structure
TPM_ST_ATTEST_COMMAND_AUDIT	0x8015	tag for an attestation structure
TPM_ST_ATTEST_SESSION_AUDIT	0x8016	tag for an attestation structure

Level 00 Revision 01.59 **Copyright © TCG 2006-2020** November 8, 2019

Name	Value	Comments
TPM_ST_ATTEST_CERTIFY	0x8017	tag for an attestation structure
TPM_ST_ATTEST_QUOTE	0x8018	tag for an attestation structure
TPM_ST_ATTEST_TIME	0x8019	tag for an attestation structure
TPM_ST_ATTEST_CREATION	0x801A	tag for an attestation structure
reserved	0x801B	do not use NOTE This was previously assigned to TPM_ST_ATTEST_NV. The tag is changed because the structure has changed
TPM_ST_ATTEST_NV_DIGEST	0x801C	tag for an attestation structure
TPM_ST_CREATION	0x8021	tag for a ticket type
TPM_ST_VERIFIED	0x8022	tag for a ticket type
TPM_ST_AUTH_SECRET	0x8023	tag for a ticket type
TPM_ST_HASHCHECK	0x8024	tag for a ticket type
TPM_ST_AUTH_SIGNED	0x8025	tag for a ticket type
TPM_ST_FU_MANIFEST	0x8029	tag for a structure describing a Field Upgrade Policy

6.10 TPM_SU (Startup Type)

These values are used in TPM2_Startup() to indicate the shutdown and startup mode. The defined startup sequences are:

a) TPM Reset – Two cases:
    1) Shutdown(CLEAR) followed by Startup(CLEAR)
    2) Startup(CLEAR) with no Shutdown()
b) TPM Restart – Shutdown(STATE) followed by Startup(CLEAR)
c) TPM Resume – Shutdown(STATE) followed by Startup(STATE)

TPM_SU values of 80 00_{16} and above are reserved for internal use of the TPM and may not be assigned values.

NOTE In the reference code, a value of FF FF_{16} indicates that the startup state has not been set. If this was defined in this table to be, say, TPM_SU_NONE, then TPM_SU_NONE would be a valid input value but the caller is not allowed to indicate the that the startup type is TPM_SU_NONE so the reserved value is defined in the implementation as required for internal TPM uses.

 Copyright © TCG 2006-2020

Table 20 — Definition of (UINT16) TPM_SU Constants <IN>

Name	Value	Description
TPM_SU_CLEAR	0x0000	on TPM2_Shutdown(), indicates that the TPM should prepare for loss of power and save state required for an orderly startup (TPM Reset). on TPM2_Startup(), indicates that the TPM should perform TPM Reset or TPM Restart
TPM_SU_STATE	0x0001	on TPM2_Shutdown(), indicates that the TPM should prepare for loss of power and save state required for an orderly startup (TPM Restart or TPM Resume) on TPM2_Startup(), indicates that the TPM should restore the state saved by TPM2_Shutdown(TPM_SU_STATE)
#TPM_RC_VALUE		response code when incorrect value is used

6.11 TPM_SE (Session Type)

This type is used in TPM2_StartAuthSession() to indicate the type of the session to be created.

Table 21 — Definition of (UINT8) TPM_SE Constants <IN>

Name	Value	Description
TPM_SE_HMAC	0x00	
TPM_SE_POLICY	0x01	
TPM_SE_TRIAL	0x03	The policy session is being used to compute the *policyHash* and not for command authorization. This setting modifies some policy commands and prevents session from being used to authorize a command.
#TPM_RC_VALUE		response code when incorrect value is used

 Copyright © TCG 2006-2020

6.12 TPM_CAP (Capabilities)

The TPM_CAP values are used in TPM2_GetCapability() to select the type of the value to be returned. The format of the response varies according to the type of the value.

Table 22 — Definition of (UINT32) TPM_CAP Constants

Capability Name	Value	Property Type	Return Type
TPM_CAP_FIRST	0x00000000		
TPM_CAP_ALGS	0x00000000	TPM_ALG_ID(1)	TPML_ALG_PROPERTY
TPM_CAP_HANDLES	0x00000001	TPM_HANDLE	TPML_HANDLE
TPM_CAP_COMMANDS	0x00000002	TPM_CC	TPML_CCA
TPM_CAP_PP_COMMANDS	0x00000003	TPM_CC	TPML_CC
TPM_CAP_AUDIT_COMMANDS	0x00000004	TPM_CC	TPML_CC
TPM_CAP_PCRS	0x00000005	reserved	TPML_PCR_SELECTION
TPM_CAP_TPM_PROPERTIES	0x00000006	TPM_PT	TPML_TAGGED_TPM_PROPERTY
TPM_CAP_PCR_PROPERTIES	0x00000007	TPM_PT_PCR	TPML_TAGGED_PCR_PROPERTY
TPM_CAP_ECC_CURVES	0x00000008	TPM_ECC_CURVE(1)	TPML_ECC_CURVE
TPM_CAP_AUTH_POLICIES	0x00000009	TPM_HANDLE(2)(3)	TPML_TAGGED_POLICY
TPM_CAP_ACT	0x0000000A	TPM_HANDLE(2(4))	TPML_ACT_DATA
TPM_CAP_LAST	0x0000000A		
TPM_CAP_VENDOR_PROPERTY	0x00000100	manufacturer specific	manufacturer-specific values
#TPM_RC_VALUE			

NOTES:
(1) The TPM_ALG_ID or TPM_ECC_CURVE is cast to a UINT32
(2) The TPM will return TPM_RC_VALUE if the handle does not reference the range for permanent handles.
(3) TPM_CAP_AUTH_POLICIES was added in revision 01.32.
(4) TPM_CAP_ACT was added in revision 01.57.

 Copyright © TCG 2006-2020

6.13 TPM_PT (Property Tag)

The TPM_PT constants are used in TPM2_GetCapability(capability = TPM_CAP_TPM_PROPERTIES) to indicate the property being selected or returned.

The values in the fixed group (PT_FIXED) are not changeable through programmatic means other than a firmware update. The values in the variable group (PT_VAR) may be changed with TPM commands but should be persistent over power cycles and only changed when indicated by the detailed actions code.

Table 23 — Definition of (UINT32) TPM_PT Constants <IN/OUT, S>

Capability Name	Value	Comments
TPM_PT_NONE	0x00000000	indicates no property type
PT_GROUP	0x00000100	The number of properties in each group. NOTE The first group with any properties is group 1 (PT_GROUP * 1). Group 0 is reserved.
PT_FIXED	PT_GROUP * 1	the group of fixed properties returned as TPMS_TAGGED_PROPERTY The values in this group are only changed due to a firmware change in the TPM.
TPM_PT_FAMILY_INDICATOR	PT_FIXED + 0	a 4-octet character string containing the TPM Family value (TPM_SPEC_FAMILY)
TPM_PT_LEVEL	PT_FIXED + 1	the level of the specification NOTE 1 For this specification, the level is zero. NOTE 2 The level is on the title page of the specification.
TPM_PT_REVISION	PT_FIXED + 2	the specification Revision times 100 EXAMPLE Revision 01.01 would have a value of 101. NOTE The Revision value is on the title page of the specification.
TPM_PT_DAY_OF_YEAR	PT_FIXED + 3	the specification day of year using TCG calendar EXAMPLE November 15, 2010, has a day of year value of 319 (00 00 01 $3F_{16}$). NOTE The specification date is on the title page of the specification or errata (see 6.1).
TPM_PT_YEAR	PT_FIXED + 4	the specification year using the CE EXAMPLE The year 2010 has a value of 00 00 07 DA_{16}. NOTE The specification date is on the title page of the specification or errata (see 6.1).
TPM_PT_MANUFACTURER	PT_FIXED + 5	the vendor ID unique to each TPM manufacturer
TPM_PT_VENDOR_STRING_1	PT_FIXED + 6	the first four characters of the vendor ID string NOTE When the vendor string is fewer than 16 octets, the additional property values do not have to be present. A vendor string of 4 octets can be represented in one 32-bit value and no null terminating character is required.
TPM_PT_VENDOR_STRING_2	PT_FIXED + 7	the second four characters of the vendor ID string
TPM_PT_VENDOR_STRING_3	PT_FIXED + 8	the third four characters of the vendor ID string
TPM_PT_VENDOR_STRING_4	PT_FIXED + 9	the fourth four characters of the vendor ID sting
TPM_PT_VENDOR_TPM_TYPE	PT_FIXED + 10	vendor-defined value indicating the TPM model
TPM_PT_FIRMWARE_VERSION_1	PT_FIXED + 11	the most-significant 32 bits of a TPM vendor-specific value indicating the version number of the firmware. See 10.12.2 and 10.12.12.

 Copyright © TCG 2006-2020

Capability Name	Value	Comments
TPM_PT_FIRMWARE_VERSION_2	PT_FIXED + 12	the least-significant 32 bits of a TPM vendor-specific value indicating the version number of the firmware. See 10.12.2 and 10.12.12.
TPM_PT_INPUT_BUFFER	PT_FIXED + 13	the maximum size of a parameter (typically, a TPM2B_MAX_BUFFER)
TPM_PT_HR_TRANSIENT_MIN	PT_FIXED + 14	the minimum number of transient objects that can be held in TPM RAM NOTE This minimum shall be no less than the minimum value required by the platform-specific specification to which the TPM is built.
TPM_PT_HR_PERSISTENT_MIN	PT_FIXED + 15	the minimum number of persistent objects that can be held in TPM NV memory NOTE This minimum shall be no less than the minimum value required by the platform-specific specification to which the TPM is built.
TPM_PT_HR_LOADED_MIN	PT_FIXED + 16	the minimum number of authorization sessions that can be held in TPM RAM NOTE This minimum shall be no less than the minimum value required by the platform-specific specification to which the TPM is built.
TPM_PT_ACTIVE_SESSIONS_MAX	PT_FIXED + 17	the number of authorization sessions that may be active at a time A session is active when it has a context associated with its handle. The context may either be in TPM RAM or be context saved. NOTE This value shall be no less than the minimum value required by the platform-specific specification to which the TPM is built.
TPM_PT_PCR_COUNT	PT_FIXED + 18	the number of PCR implemented NOTE This number is determined by the defined attributes, not the number of PCR that are populated.
TPM_PT_PCR_SELECT_MIN	PT_FIXED + 19	the minimum number of octets in a TPMS_PCR_SELECT.*sizeOfSelect* NOTE This value is not determined by the number of PCR implemented but by the number of PCR required by the platform-specific specification with which the TPM is compliant or by the implementer if not adhering to a platform-specific specification.
TPM_PT_CONTEXT_GAP_MAX	PT_FIXED + 20	the maximum allowed difference (unsigned) between the *contextID* values of two saved session contexts This value shall be 2^n-1, where n is at least 16.
	PT_FIXED + 21	skipped
TPM_PT_NV_COUNTERS_MAX	PT_FIXED + 22	the maximum number of NV Indexes that are allowed to have the TPM_NT_COUNTER attribute NOTE 1 It is allowed for this value to be larger than the number of NV Indexes that can be defined. This would be indicative of a TPM implementation that did not use different implementation technology for different NV Index types. NOTE 2 The value zero indicates that there is no fixed maximum. The number of counter indexes is determined by the available NV memory pool.
TPM_PT_NV_INDEX_MAX	PT_FIXED + 23	the maximum size of an NV Index data area

 Copyright © TCG 2006-2020

Capability Name	Value	Comments
TPM_PT_MEMORY	PT_FIXED + 24	a TPMA_MEMORY indicating the memory management method for the TPM
TPM_PT_CLOCK_UPDATE	PT_FIXED + 25	interval, in milliseconds, between updates to the copy of TPMS_CLOCK_INFO.*clock* in NV
TPM_PT_CONTEXT_HASH	PT_FIXED + 26	the algorithm used for the integrity HMAC on saved contexts and for hashing the *fuData* of TPM2_FirmwareRead()
TPM_PT_CONTEXT_SYM	PT_FIXED + 27	TPM_ALG_ID, the algorithm used for encryption of saved contexts
TPM_PT_CONTEXT_SYM_SIZE	PT_FIXED + 28	TPM_KEY_BITS, the size of the key used for encryption of saved contexts
TPM_PT_ORDERLY_COUNT	PT_FIXED + 29	the modulus - 1 of the count for NV update of an orderly counter The returned value is MAX_ORDERLY_COUNT. This will have a value of $2^N – 1$ where $1 \le N \le 32$ NOTE 1 An “orderly counter” is an NV Index with an TPM_NT of TPM_NV_COUNTER and TPMA_NV_ORDERLY SET. NOTE 2 When the low-order bits of a counter equal this value, an NV write occurs on the next increment.
TPM_PT_MAX_COMMAND_SIZE	PT_FIXED + 30	the maximum value for *commandSize* in a command
TPM_PT_MAX_RESPONSE_SIZE	PT_FIXED + 31	the maximum value for *responseSize* in a response
TPM_PT_MAX_DIGEST	PT_FIXED + 32	the maximum size of a digest that can be produced by the TPM
TPM_PT_MAX_OBJECT_CONTEXT	PT_FIXED + 33	the maximum size of an object context that will be returned by TPM2_ContextSave
TPM_PT_MAX_SESSION_CONTEXT	PT_FIXED + 34	the maximum size of a session context that will be returned by TPM2_ContextSave
TPM_PT_PS_FAMILY_INDICATOR	PT_FIXED + 35	platform-specific family (a TPM_PS value)(see Table 25) NOTE The platform-specific values for the TPM_PT_PS parameters are in the relevant platform-specific specification. In the reference implementation, all of these values are 0.
TPM_PT_PS_LEVEL	PT_FIXED + 36	the level of the platform-specific specification
TPM_PT_PS_REVISION	PT_FIXED + 37	a platform specific value
TPM_PT_PS_DAY_OF_YEAR	PT_FIXED + 38	the platform-specific TPM specification day of year using TCG calendar EXAMPLE November 15, 2010, has a day of year value of 319 (00 00 01 $3F_{16}$).
TPM_PT_PS_YEAR	PT_FIXED + 39	the platform-specific TPM specification year using the CE EXAMPLE The year 2010 has a value of 00 00 07 DA_{16}.
TPM_PT_SPLIT_MAX	PT_FIXED + 40	the number of split signing operations supported by the TPM
TPM_PT_TOTAL_COMMANDS	PT_FIXED + 41	total number of commands implemented in the TPM
TPM_PT_LIBRARY_COMMANDS	PT_FIXED + 42	number of commands from the TPM library that are implemented
TPM_PT_VENDOR_COMMANDS	PT_FIXED + 43	number of vendor commands that are implemented
TPM_PT_NV_BUFFER_MAX	PT_FIXED + 44	the maximum data size in one NV write, NV read, NV extend, or NV certify command

 Copyright © TCG 2006-2020

Capability Name	Value	Comments
TPM_PT_MODES	PT_FIXED + 45	a TPMA_MODES value, indicating that the TPM is designed for these modes.
TPM_PT_MAX_CAP_BUFFER	PT_FIXED + 46	the maximum size of a TPMS_CAPABILITY_DATA structure returned in TPM2_GetCapability().
		Intentionally left empty
PT_VAR	PT_GROUP * 2	the group of variable properties returned as TPMS_TAGGED_PROPERTY The properties in this group change because of a Protected Capability other than a firmware update. The values are not necessarily persistent across all power transitions.
TPM_PT_PERMANENT	PT_VAR + 0	TPMA_PERMANENT
TPM_PT_STARTUP_CLEAR	PT_VAR + 1	TPMA_STARTUP_CLEAR
TPM_PT_HR_NV_INDEX	PT_VAR + 2	the number of NV Indexes currently defined
TPM_PT_HR_LOADED	PT_VAR + 3	the number of authorization sessions currently loaded into TPM RAM
TPM_PT_HR_LOADED_AVAIL	PT_VAR + 4	the number of additional authorization sessions, of any type, that could be loaded into TPM RAM This value is an estimate. If this value is at least 1, then at least one authorization session of any type may be loaded. Any command that changes the RAM memory allocation can make this estimate invalid. NOTE A valid implementation may return 1 even if more than one authorization session would fit into RAM.
TPM_PT_HR_ACTIVE	PT_VAR + 5	the number of active authorization sessions currently being tracked by the TPM This is the sum of the loaded and saved sessions.
TPM_PT_HR_ACTIVE_AVAIL	PT_VAR + 6	the number of additional authorization sessions, of any type, that could be created This value is an estimate. If this value is at least 1, then at least one authorization session of any type may be created. Any command that changes the RAM memory allocation can make this estimate invalid. NOTE A valid implementation may return 1 even if more than one authorization session could be created.
TPM_PT_HR_TRANSIENT_AVAIL	PT_VAR + 7	estimate of the number of additional transient objects that could be loaded into TPM RAM This value is an estimate. If this value is at least 1, then at least one object of any type may be loaded. Any command that changes the memory allocation can make this estimate invalid. NOTE A valid implementation may return 1 even if more than one transient object would fit into RAM.
TPM_PT_HR_PERSISTENT	PT_VAR + 8	the number of persistent objects currently loaded into TPM NV memory

 Copyright © TCG 2006-2020

Capability Name	Value	Comments
TPM_PT_HR_PERSISTENT_AVAIL	PT_VAR + 9	the number of additional persistent objects that could be loaded into NV memory This value is an estimate. If this value is at least 1, then at least one object of any type may be made persistent. Any command that changes the NV memory allocation can make this estimate invalid. NOTE A valid implementation may return 1 even if more than one persistent object would fit into NV memory.
TPM_PT_NV_COUNTERS	PT_VAR + 10	the number of defined NV Indexes that have NV the TPM_NT_COUNTER attribute
TPM_PT_NV_COUNTERS_AVAIL	PT_VAR + 11	the number of additional NV Indexes that can be defined with their TPM_NT of TPM_NV_COUNTER and the TPMA_NV_ORDERLY attribute SET This value is an estimate. If this value is at least 1, then at least one NV Index may be created with a TPM_NT of TPM_NV_COUNTER and the TPMA_NV_ORDERLY attributes. Any command that changes the NV memory allocation can make this estimate invalid. NOTE A valid implementation may return 1 even if more than one NV counter could be defined.
TPM_PT_ALGORITHM_SET	PT_VAR + 12	code that limits the algorithms that may be used with the TPM
TPM_PT_LOADED_CURVES	PT_VAR + 13	the number of loaded ECC curves
TPM_PT_LOCKOUT_COUNTER	PT_VAR + 14	the current value of the lockout counter (*failedTries*)
TPM_PT_MAX_AUTH_FAIL	PT_VAR + 15	the number of authorization failures before DA lockout is invoked
TPM_PT_LOCKOUT_INTERVAL	PT_VAR + 16	the number of seconds before the value reported by TPM_PT_LOCKOUT_COUNTER is decremented
TPM_PT_LOCKOUT_RECOVERY	PT_VAR + 17	the number of seconds after a lockoutAuth failure before use of lockoutAuth may be attempted again
TPM_PT_NV_WRITE_RECOVERY	PT_VAR + 18	number of milliseconds before the TPM will accept another command that will modify NV This value is an approximation and may go up or down over time.
TPM_PT_AUDIT_COUNTER_0	PT_VAR + 19	the high-order 32 bits of the command audit counter
TPM_PT_AUDIT_COUNTER_1	PT_VAR + 20	the low-order 32 bits of the command audit counter

Level 00 Revision 01.59 **Copyright © TCG 2006-2020** November 8, 2019

6.14 TPM_PT_PCR (PCR Property Tag)

The TPM_PT_PCR constants are used in TPM2_GetCapability() to indicate the property being selected or returned. The PCR properties can be read when *capability* == TPM_CAP_PCR_PROPERTIES. If there is no property that corresponds to the value of *property*, the next higher value is returned, if it exists.

Table 24 — Definition of (UINT32) TPM_PT_PCR Constants <IN/OUT, S>

Capability Name	Value	Comments
TPM_PT_PCR_FIRST	0x00000000	bottom of the range of TPM_PT_PCR properties
TPM_PT_PCR_SAVE	0x00000000	a SET bit in the TPMS_PCR_SELECT indicates that the PCR is saved and restored by TPM_SU_STATE
TPM_PT_PCR_EXTEND_L0	0x00000001	a SET bit in the TPMS_PCR_SELECT indicates that the PCR may be extended from locality 0 This property is only present if a locality other than 0 is implemented.
TPM_PT_PCR_RESET_L0	0x00000002	a SET bit in the TPMS_PCR_SELECT indicates that the PCR may be reset by TPM2_PCR_Reset() from locality 0
TPM_PT_PCR_EXTEND_L1	0x00000003	a SET bit in the TPMS_PCR_SELECT indicates that the PCR may be extended from locality 1 This property is only present if locality 1 is implemented.
TPM_PT_PCR_RESET_L1	0x00000004	a SET bit in the TPMS_PCR_SELECT indicates that the PCR may be reset by TPM2_PCR_Reset() from locality 1 This property is only present if locality 1 is implemented.
TPM_PT_PCR_EXTEND_L2	0x00000005	a SET bit in the TPMS_PCR_SELECT indicates that the PCR may be extended from locality 2 This property is only present if localities 1 and 2 are implemented.
TPM_PT_PCR_RESET_L2	0x00000006	a SET bit in the TPMS_PCR_SELECT indicates that the PCR may be reset by TPM2_PCR_Reset() from locality 2 This property is only present if localities 1 and 2 are implemented.
TPM_PT_PCR_EXTEND_L3	0x00000007	a SET bit in the TPMS_PCR_SELECT indicates that the PCR may be extended from locality 3 This property is only present if localities 1, 2, and 3 are implemented.
TPM_PT_PCR_RESET_L3	0x00000008	a SET bit in the TPMS_PCR_SELECT indicates that the PCR may be reset by TPM2_PCR_Reset() from locality 3 This property is only present if localities 1, 2, and 3 are implemented.
TPM_PT_PCR_EXTEND_L4	0x00000009	a SET bit in the TPMS_PCR_SELECT indicates that the PCR may be extended from locality 4 This property is only present if localities 1, 2, 3, and 4 are implemented.
TPM_PT_PCR_RESET_L4	0x0000000A	a SET bit in the TPMS_PCR_SELECT indicates that the PCR may be reset by TPM2_PCR_Reset() from locality 4 This property is only present if localities 1, 2, 3, and 4 are implemented.

 Copyright © TCG 2006-2020

Capability Name	Value	Comments
reserved	0x0000000B – 0x00000010	the values in this range are reserved They correspond to values that may be used to describe attributes associated with the extended localities (32-255).synthesize additional software localities. The meaning of these properties need not be the same as the meaning for the Extend and Reset properties above.
TPM_PT_PCR_NO_INCREMENT	0x00000011	a SET bit in the TPMS_PCR_SELECT indicates that modifications to this PCR (reset or Extend) will not increment the *pcrUpdateCounter*
TPM_PT_PCR_DRTM_RESET	0x00000012	a SET bit in the TPMS_PCR_SELECT indicates that the PCR is reset by a D-RTM event These PCR are reset to -1 on TPM2_Startup() and reset to 0 on a _TPM_Hash_End event following a _TPM_Hash_Start event.
TPM_PT_PCR_POLICY	0x00000013	a SET bit in the TPMS_PCR_SELECT indicates that the PCR is controlled by policy This property is only present if the TPM supports policy control of a PCR.
TPM_PT_PCR_AUTH	0x00000014	a SET bit in the TPMS_PCR_SELECT indicates that the PCR is controlled by an authorization value This property is only present if the TPM supports authorization control of a PCR.
reserved	0x00000015	reserved for the next (2nd) TPM_PT_PCR_POLICY set
reserved	0x00000016	reserved for the next (2nd) TPM_PT_PCR_AUTH set
reserved	0x00000017 – 0x00000210	reserved for the 2nd through 255th TPM_PT_PCR_POLICY and TPM_PT_PCR_AUTH values
reserved	0x00000211	reserved to the 256th, and highest allowed, TPM_PT_PCR_POLICY set
reserved	0x00000212	reserved to the 256th, and highest allowed, TPM_PT_PCR_AUTH set
reserved	0x00000213	new PCR property values may be assigned starting with this value
TPM_PT_PCR_LAST	0x00000014	top of the range of TPM_PT_PCR properties of the implementation If the TPM receives a request for a PCR property with a value larger than this, the TPM will return a zero length list and set the *moreData* parameter to NO. NOTE This is an implementation-specific value. The value shown reflects the reference code implementation.

 Copyright © TCG 2006-2020

6.15 TPM_PS (Platform Specific)

The platform values in Table 25 are used for the TPM_PT_PS_FAMILY_INDICATOR.

Table 25 is an informative example of a TPM_PS constants table in the TCG Registry of Reserved TPM 2.0 Handles and Localities. It is provided for illustrative purposes only.

NOTE Values below six (6) have the same values as the purview assignments in TPM 1.2.

Table 25 — Definition of (UINT32) TPM_PS Constants <OUT>

Capability Name	Value	Comments
TPM_PS_MAIN	0x00000000	not platform specific
TPM_PS_PC	0x00000001	PC Client
TPM_PS_PDA	0x00000002	PDA (includes all mobile devices that are not specifically cell phones)
TPM_PS_CELL_PHONE	0x00000003	Cell Phone
TPM_PS_SERVER	0x00000004	Server WG
TPM_PS_PERIPHERAL	0x00000005	Peripheral WG
TPM_PS_TSS	0x00000006	TSS WG (deprecated)
TPM_PS_STORAGE	0x00000007	Storage WG
TPM_PS_AUTHENTICATION	0x00000008	Authentication WG
TPM_PS_EMBEDDED	0x00000009	Embedded WG
TPM_PS_HARDCOPY	0x0000000A	Hardcopy WG
TPM_PS_INFRASTRUCTURE	0x0000000B	Infrastructure WG (deprecated)
TPM_PS_VIRTUALIZATION	0x0000000C	Virtualization WG
TPM_PS_TNC	0x0000000D	Trusted Network Connect WG (deprecated)
TPM_PS_MULTI_TENANT	0x0000000E	Multi-tenant WG (deprecated)
TPM_PS_TC	0x0000000F	Technical Committee (deprecated)

 Copyright © TCG 2006-2020

7 Handles

7.1 Introduction

Handles are 32-bit values used to reference shielded locations of various types within the TPM.

Table 26 — Definition of Types for Handles

Type	Name	Description
UINT32	TPM_HANDLE	

Handles may refer to objects (keys or data blobs), authorization sessions (HMAC and policy), NV Indexes, permanent TPM locations, and PCR.

7.2 TPM_HT (Handle Types)

The 32-bit handle space is divided into 256 regions of equal size with 2^{24} values in each. Each of these ranges represents a handle type.

The type of the entity is indicated by the MSO of its handle. The values for the MSO and the entity referenced are shown in Table 27.

Table 27 — Definition of (UINT8) TPM_HT Constants <S>

Name	Value	Comments
TPM_HT_PCR	0x00	**PCR** – consecutive numbers, starting at 0, that reference the PCR registers A platform-specific specification will set the minimum number of PCR and an implementation may have more.
TPM_HT_NV_INDEX	0x01	**NV Index** – assigned by the caller
TPM_HT_HMAC_SESSION	0x02	**HMAC Authorization Session** – assigned by the TPM when the session is created
TPM_HT_LOADED_SESSION	0x02	**Loaded Authorization Session** – used only in the context of TPM2_GetCapability This type references both loaded HMAC and loaded policy authorization sessions.
TPM_HT_POLICY_SESSION	0x03	**Policy Authorization Session** – assigned by the TPM when the session is created
TPM_HT_SAVED_SESSION	0x03	**Saved Authorization Session** – used only in the context of TPM2_GetCapability This type references saved authorization session contexts for which the TPM is maintaining tracking information.
TPM_HT_PERMANENT	0x40	**Permanent Values** – assigned by this specification in Table 28
TPM_HT_TRANSIENT	0x80	**Transient Objects** – assigned by the TPM when an object is loaded into transient-object memory or when a persistent object is converted to a transient object
TPM_HT_PERSISTENT	0x81	**Persistent Objects** – assigned by the TPM when a loaded transient object is made persistent
TPM_HT_AC	0x90	**Attached Component** – handle for an Attached Component.

 Copyright © TCG 2006-2020

When a transient object is loaded, the TPM shall assign a handle with an MSO of TPM_HT_TRANSIENT. The object may be assigned a different handle each time it is loaded. The TPM shall ensure that handles assigned to transient objects are unique and assigned to only one transient object at a time.

EXAMPLE 1 If a TPM is only able to hold 4 transient objects in internal memory, it might choose to assign handles to those objects with the values 80 00 00 00_{16} – 80 00 00 03_{16}.

When a transient object is converted to a persistent object (TPM2_EvictControl()), the TPM shall validate that the handle provided by the caller has an MSO of TPM_HT_PERSISTENT and that the handle is not already assigned to a persistent object.

A handle is assigned to a session when the session is started. The handle shall have an MSO equal to TPM_HT_SESSION and remain associated with that session until the session is closed or flushed. The TPM shall ensure that a session handle is only associated with one session at a time. When the session is loaded into the TPM using TPM2_LoadContext(), it will have the same handle each time it is loaded.

EXAMPLE 2 If a TPM is only able to track 64 active sessions at a time, it could number those sessions using the values xx 00 01 00_{16} – xx 00 01 $3F_{16}$ where xx is either 02_{16} or 03_{16} depending on the session type.

7.3 Persistent Handle Sub-ranges

Persistent handles are assigned by the caller of TPM2_EvictControl(). Owner Authorization or Platform Authorization is required to authorize allocation of space for a persistent object. These entities are given separate ranges of persistent handles so that they do not have to allocate from a common range of handles.

NOTE While this “namespace” allocation of the handle ranges could have been handled by convention, TPM enforcement is used to prevent errors by the OS or malicious software from affecting the platform’s use of the NV memory.

The Owner is allocated persistent handles in the range of 81 00 00 00_{16} to 81 7F FF FF_{16} inclusive and the TPM will return an error if Owner Authorization is used to attempt to assign a persistent handle outside of this range.

November 8, 2019 **Copyright © TCG 2006-2020** Level 00 Revision 01.59

7.4 TPM_RH (Permanent Handles)

Table 28 lists the architecturally defined handles that cannot be changed. The handles include authorization handles, and special handles.

Table 28 — Definition of (TPM_HANDLE) TPM_RH Constants <S>

Name	Value	Type	Comments
TPM_RH_FIRST	0x40000000	R	
TPM_RH_SRK	0x40000000	R	not used[1]
TPM_RH_OWNER	0x40000001	K, A, P	handle references the Storage Primary Seed (SPS), the *ownerAuth*, and the *ownerPolicy*
TPM_RH_REVOKE	0x40000002	R	not used[1]
TPM_RH_TRANSPORT	0x40000003	R	not used[1]
TPM_RH_OPERATOR	0x40000004	R	not used[1]
TPM_RH_ADMIN	0x40000005	R	not used[1]
TPM_RH_EK	0x40000006	R	not used[1]
TPM_RH_NULL	0x40000007	K, A, P	a handle associated with the null hierarchy, an EmptyAuth *authValue*, and an Empty Policy *authPolicy*.
TPM_RH_UNASSIGNED	0x40000008	R	value reserved to the TPM to indicate a handle location that has not been initialized or assigned
TPM_RS_PW	0x40000009	S	authorization value used to indicate a password authorization session
TPM_RH_LOCKOUT	0x4000000A	A	references the authorization associated with the dictionary attack lockout reset
TPM_RH_ENDORSEMENT	0x4000000B	K, A, P	references the Endorsement Primary Seed (EPS), *endorsementAuth*, and *endorsementPolicy*
TPM_RH_PLATFORM	0x4000000C	K, A, P	references the Platform Primary Seed (PPS), *platformAuth*, and *platformPolicy*
TPM_RH_PLATFORM_NV	0x4000000D	C	for phEnableNV
TPM_RH_AUTH_00	0x40000010	A	Start of a range of authorization values that are vendor-specific. A TPM may support any of the values in this range as are needed for vendor-specific purposes. Disabled if ehEnable is CLEAR. NOTE “Any” includes “none”.
TPM_RH_AUTH_FF	0x4000010F	A	End of the range of vendor-specific authorization values.
TPM_RH_ACT_0	0x40000110	A,P	Start of the range of authenticated timers
TPM_RH_ACT_F	0x4000011F	A,P	End of the range of authenticated timers
TPM_RH_LAST	0x4000011F	R	the top of the reserved handle area This is set to allow TPM2_GetCapability() to know where to stop. It may vary as implementations add to the permanent handle area.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

Name	Value	Type	Comments
Type definitions: **R** – a reserved value **K** – a Primary Seed **A** – an authorization value **P** – a policy value **S** – a session handle **C** - a control Note 1 The handle is only used in a TPM that is compatible with a previous version of this specification. It is not used in any command defined in this version of the specification.			

7.5 TPM_HC (Handle Value Constants)

The definitions in Table 29 are used to define many of the interface data types.

These values, that indicate ranges, are informative and may be changed by an implementation. The TPM will always return the correct handle type as described in 7.2 Table 27:

- HMAC_SESSION_FIRST—HMAC_SESSION_LAST,
- LOADED_SESSION_FIRST—LOADED_SESSION_LAST,
- POLICY_SESSION_FIRST—POLICY_SESSION_LAST,
- TRANSIENT_FIRST—TRANSIENT_LAST,
- ACTIVE_SESSION_FIRST—ACTIVE_SESSION_LAST,
- PCR_FIRST—PCR_LAST

These values are input by the caller. The TPM implementation should support the entire range:

- PERSISTENT_FIRST—PERSISTENT_LAST,
- PLATFORM_PERSISTENT—PLATFORM_PERSISTENT+0x007FFFFF,
- NV_INDEX_FIRST—NV_INDEX_LAST,
- PERMANENT_FIRST—PERMANENT_LAST

NOTE PCR0 is architecturally defined to have a handle value of 0.

For the reference implementation, the handle range for sessions starts at the lowest allowed value for a session handle. The highest value for a session handle is determined by how many active sessions are allowed by the implementation. The MSO of the session handle will be set according to the session type.

A similar approach is used for transient objects with the first assigned handle at the bottom of the range defined by TPM_HT_TRANSIENT and the top of the range determined by the implementation-dependent value of MAX_LOADED_OBJECTS.

The first assigned handle for evict objects is also at the bottom of the allowed range defined by TPM_HT_PERSISTENT and the top of the range determined by the implementation-dependent value of MAX_EVICT_OBJECTS.

NOTE The values in Table 29 are intended to facilitate the process of making the handle larger than 32 bits in the future. It is intended that HR_MASK and HR_SHIFT are the only values that need change to resize the handle space.

 Copyright © TCG 2006-2020

Table 29 — Definition of (TPM_HANDLE) TPM_HC Constants <S>

Name	Value	Comments
HR_HANDLE_MASK	0x00FFFFFF	to mask off the HR
HR_RANGE_MASK	0xFF000000	to mask off the variable part
HR_SHIFT	24	
HR_PCR	(TPM_HT_PCR << HR_SHIFT)	
HR_HMAC_SESSION	(TPM_HT_HMAC_SESSION << HR_SHIFT)	
HR_POLICY_SESSION	(TPM_HT_POLICY_SESSION << HR_SHIFT)	
HR_TRANSIENT	(TPM_HT_TRANSIENT << HR_SHIFT)	
HR_PERSISTENT	(TPM_HT_PERSISTENT << HR_SHIFT)	
HR_NV_INDEX	(TPM_HT_NV_INDEX << HR_SHIFT)	
HR_PERMANENT	(TPM_HT_PERMANENT << HR_SHIFT)	
PCR_FIRST	(HR_PCR + 0)	first PCR
PCR_LAST	(PCR_FIRST + IMPLEMENTATION_PCR-1)	last PCR
HMAC_SESSION_FIRST	(HR_HMAC_SESSION + 0)	first HMAC session
HMAC_SESSION_LAST	(HMAC_SESSION_FIRST+MAX_ACTIVE_SESSIONS-1)	last HMAC session
LOADED_SESSION_FIRST	HMAC_SESSION_FIRST	used in GetCapability
LOADED_SESSION_LAST	HMAC_SESSION_LAST	used in GetCapability
POLICY_SESSION_FIRST	(HR_POLICY_SESSION + 0)	first policy session
POLICY_SESSION_LAST	(POLICY_SESSION_FIRST + MAX_ACTIVE_SESSIONS-1)	last policy session
TRANSIENT_FIRST	(HR_TRANSIENT + 0)	first transient object
ACTIVE_SESSION_FIRST	POLICY_SESSION_FIRST	used in GetCapability
ACTIVE_SESSION_LAST	POLICY_SESSION_LAST	used in GetCapability
TRANSIENT_LAST	(TRANSIENT_FIRST+MAX_LOADED_OBJECTS-1)	last transient object
PERSISTENT_FIRST	(HR_PERSISTENT + 0)	first persistent object
PERSISTENT_LAST	(PERSISTENT_FIRST + 0x00FFFFFF)	last persistent object
PLATFORM_PERSISTENT	(PERSISTENT_FIRST + 0x00800000)	first platform persistent object
NV_INDEX_FIRST	(HR_NV_INDEX + 0)	first allowed NV Index
NV_INDEX_LAST	(NV_INDEX_FIRST + 0x00FFFFFF)	last allowed NV Index
PERMANENT_FIRST	TPM_RH_FIRST	
PERMANENT_LAST	TPM_RH_LAST	
HR_NV_AC	((TPM_HT_NV_INDEX << HR_SHIFT) + 0xD00000)	AC aliased NV Index
NV_AC_FIRST	(HR_NV_AC + 0)	first NV Index aliased to Attached Component
NV_AC_LAST	(HR_NV_AC + 0x0000FFFF)	last NV Index aliased to Attached Component
HR_AC	(TPM_HT_AC << HR_SHIFT)	AC Handle
AC_FIRST	(HR_AC + 0)	first Attached Component

 Copyright © TCG 2006-2020

Name	Value	Comments
AC_LAST	(HR_AC + 0x0000FFFF)	last Attached Component

 Copyright © TCG 2006-2020

8 Attribute Structures

8.1 Description

Attributes are expressed as bit fields of varying size. An attribute field structure may be 1, 2, or 4 octets in length.

The bit numbers for an attribute structure are assigned with the number 0 assigned to the least-significant bit of the structure and the highest number assigned to the most-significant bit of the structure.

The least significant bit is determined by treating the attribute structure as an integer. The least-significant bit would be the bit that is set when the value of the integer is 1.

When any reserved bit in an attribute is SET, the TPM shall return TPM_RC_RESERVED_BITS. This response code is not shown in the tables for attributes.

8.2 TPMA_ALGORITHM

This structure defines the attributes of an algorithm.

Each algorithm has a fundamental attribute: *asymmetric*, *symmetric*, or *hash*. In some cases (e.g., TPM_ALG_RSA or TPM_ALG_AES), this is the only attribute.

A mode, method, or scheme may have an associated asymmetric, symmetric, or hash algorithm.

NOTE A hash algorithm that can be used directly is one that has only the *hash* attribute SET.

EXAMPLE A PCR bank or an object Name can only use an algorithm that has only the *hash* attribute SET.

Table 30 — Definition of (UINT32) TPMA_ALGORITHM Bits

Bit	Name	Definition
0	asymmetric	**SET (1):** an asymmetric algorithm with public and private portions **CLEAR (0):** not an asymmetric algorithm
1	symmetric	**SET (1):** a symmetric block cipher **CLEAR (0):** not a symmetric block cipher
2	hash	**SET (1):** a hash algorithm **CLEAR (0):** not a hash algorithm
3	object	**SET (1):** an algorithm that may be used as an object type **CLEAR (0):** an algorithm that is not used as an object type
7:4	Reserved	
8	signing	**SET (1):** a signing algorithm. The setting of *asymmetric*, *symmetric*, and *hash* will indicate the type of signing algorithm. **CLEAR (0):** not a signing algorithm
9	encrypting	**SET (1):** an encryption/decryption algorithm. The setting of *asymmetric*, *symmetric*, and *hash* will indicate the type of encryption/decryption algorithm. **CLEAR (0):** not an encryption/decryption algorithm
10	method	**SET (1):** a method such as a key derivative function (KDF) **CLEAR (0):** not a method
31:11	Reserved	

 Copyright © TCG 2006-2020

8.3 TPMA_OBJECT (Object Attributes)

8.3.1 Introduction

This attribute structure indicates an object's use, its authorization types, and its relationship to other objects.

The state of the attributes is determined when the object is created and they are never changed by the TPM. Additionally, the setting of these structures is reflected in the integrity value of the private area of an object in order to allow the TPM to detect modifications of the Protected Object when stored off the TPM.

8.3.2 Structure Definition

Table 31 — Definition of (UINT32) TPMA_OBJECT Bits

Bit	Name	Definition
0	Reserved	shall be zero
1	fixedTPM	**SET (1):** The hierarchy of the object, as indicated by its Qualified Name, may not change. **CLEAR (0):** The hierarchy of the object may change as a result of this object or an ancestor key being duplicated for use in another hierarchy. NOTE *fixedTPM* does not indicate that key material resides on a single TPM (see *sensitiveDataOrigin*).
2	stClear	**SET (1):** Previously saved contexts of this object may not be loaded after Startup(CLEAR). **CLEAR (0):** Saved contexts of this object may be used after a Shutdown(STATE) and subsequent Startup().
3	Reserved	shall be zero
4	fixedParent	**SET (1):** The parent of the object may not change. **CLEAR (0):** The parent of the object may change as the result of a TPM2_Duplicate() of the object.
5	sensitiveDataOrigin	**SET (1):** Indicates that, when the object was created with TPM2_Create() or TPM2_CreatePrimary(), the TPM generated all of the sensitive data other than the *authValue*. **CLEAR (0):** A portion of the sensitive data, other than the *authValue*, was provided by the caller.
6	userWithAuth	**SET (1):** Approval of USER role actions with this object may be with an HMAC session or with a password using the *authValue* of the object or a policy session. **CLEAR (0):** Approval of USER role actions with this object may only be done with a policy session.
7	adminWithPolicy	**SET (1):** Approval of ADMIN role actions with this object may only be done with a policy session. **CLEAR (0):** Approval of ADMIN role actions with this object may be with an HMAC session or with a password using the *authValue* of the object or a policy session.
9:8	Reserved	shall be zero
10	noDA	**SET (1):** The object is not subject to dictionary attack protections. **CLEAR (0):** The object is subject to dictionary attack protections.

 Copyright © TCG 2006-2020

Bit	Name	Definition
11	encryptedDuplication	**SET (1):** If the object is duplicated, then *symmetricAlg* shall not be TPM_ALG_NULL and *newParentHandle* shall not be TPM_RH_NULL. **CLEAR (0):** The object may be duplicated without an inner wrapper on the private portion of the object and the new parent may be TPM_RH_NULL.
15:12	Reserved	shall be zero
16	restricted	**SET (1):** Key usage is restricted to manipulate structures of known format; the parent of this key shall have *restricted* SET. **CLEAR (0):** Key usage is not restricted to use on special formats.
17	decrypt	**SET (1):** The private portion of the key may be used to decrypt. **CLEAR (0):** The private portion of the key may not be used to decrypt.
18	sign / encrypt	**SET (1):** For a symmetric cipher object, the private portion of the key may be used to encrypt. For other objects, the private portion of the key may be used to sign. **CLEAR (0):** The private portion of the key may not be used to sign or encrypt.
19	x509sign	**SET (1):** An asymmetric key that may not be used to sign with TPM2_Sign() **CLEAR (0):** A key that may be used with TPM2_Sign() if *sign* is SET NOTE: This attribute only has significance if *sign* is SET.
31:20	Reserved	shall be zero

8.3.3 Attribute Descriptions

8.3.3.1 Introduction

The following remaining paragraphs in 8.3.3 describe the use and settings for each of the TPMA_OBJECT attributes. The description includes checks that are performed on the *objectAttributes* when an object is created, when it is loaded, and when it is imported. In these descriptions:

Creation indicates settings for the *template* parameter in TPM2_Create() or TPM2_CreatePrimary()

Load indicates settings for the *inPublic* parameter in TPM2_Load()

Import indicates settings for the *objectPublic* parameter in TPM2_Import()

External indicates settings that apply to the *inPublic* parameter in TPM2_LoadExternal() if both the public and sensitive portions of the object are loaded

NOTE For TPM2_LoadExternal() when only the public portion of the object is loaded, the only attribute checks are the checks in the validation code following Table 31 and the reserved attributes check.

For any consistency error of attributes in TPMA_OBJECT, the TPM shall return TPM_RC_ATTRIBUTES.

 Copyright © TCG 2006-2020

8.3.3.2 Bit[1] – *fixedTPM*

When SET, the object cannot be duplicated for use on a different TPM, either directly or indirectly and the Qualified Name of the object cannot change. When CLEAR, the object's Qualified Name may change if the object or an ancestor is duplicated.

NOTE This attribute is the logical inverse of the migratable attribute in 1.2. That is, when this attribute is CLEAR, it is the equivalent to a 1.2 object with migratable SET.

Creation If *fixedTPM* is SET in the object's parent, then *fixedTPM* and *fixedParent* shall both be set to the same value in *template.* If *fixedTPM* is CLEAR in the parent, this attribute shall also be CLEAR in *template*.

NOTE For a Primary Object, the parent is considered to have *fixedTPM* SET.

Load If *fixedTPM* is SET in the object's parent, then *fixedTPM* and *fixedParent* shall both be set to the same value. If *fixedTPM* is CLEAR in the parent, this attribute shall also be CLEAR.

Import shall be CLEAR

External shall be CLEAR if both the public and sensitive portions are loaded or if fixedParent is CLEAR, otherwise may be SET or CLEAR

8.3.3.3 Bit[2] – *stClear*

If this attribute is SET, then saved contexts of this object will be invalidated on TPM2_Startup(TPM_SU_CLEAR). If the attribute is CLEAR, then the TPM shall not invalidate the saved context if the TPM received TPM2_Shutdown(TPM_SU_STATE). If the saved state is valid when checked at the next TPM2_Startup(), then the TPM shall continue to be able to use the saved contexts.

Creation may be SET or CLEAR in template

Load may be SET or CLEAR

Import may be SET or CLEAR

External may be SET or CLEAR

8.3.3.4 Bit[4] – *fixedParent*

If this attribute is SET, the object's parent may not be changed. That is, this object may not be the object of a TPM2_Duplicate(). If this attribute is CLEAR, then this object may be the object of a TPM2_Duplicate().

Creation may be SET or CLEAR in template

Load may be SET or CLEAR

Import shall be CLEAR

External shall be CLEAR if both the public and sensitive portions are loaded; otherwise it may be SET or CLEAR

8.3.3.5 Bit[5] – *sensitiveDataOrigin*

This attribute is SET for any key that was generated by TPM in TPM2_Create() or TPM2_CreatePrimary(). If CLEAR, it indicates that the sensitive part of the object (other than the *obfuscation* value) was provided by the caller.

 Copyright © TCG 2006-2020

NOTE 1 If the *fixedTPM* attribute is SET, then this attribute is authoritative and accurately reflects the source of the sensitive area data. If the *fixedTPM* attribute is CLEAR, then validation of this attribute requires evaluation of the properties of the ancestor keys.

Creation If *inSensitive.sensitive.data*.size is zero, then this attribute shall be SET in the template; otherwise, it shall be CLEAR in the template.

NOTE 2 The *inSensitive.sensitive.data.size* parameter is required to be zero for an asymmetric key so *sensitiveDataOrigin* is required to be SET.

NOTE 3 The *inSensitive.sensitive.data.size* parameter may not be zero for a data object so *sensitiveDataOrigin* is required to be CLEAR. A data object has *type* = TPM_ALG_KEYEDHASH and its *sign* and *decrypt* attributes are CLEAR.

Load may be SET or CLEAR

Import may be SET or CLEAR

External may be SET or CLEAR

8.3.3.6 Bit[6] – *userWithAuth*

If SET, authorization for operations that require USER role authorization may be given if the caller provides proof of knowledge of the *authValue* of the object with an HMAC authorization session or a password.

If this attribute is CLEAR, then HMAC or password authorizations may not be used for USER role authorizations.

NOTE 1 Regardless of the setting of this attribute, authorizations for operations that require USER role authorizations may be provided with a policy session that satisfies the object's *authPolicy*.

NOTE 2 Regardless of the setting of this attribute, the *authValue* may be referenced in a policy session or used to provide the *bind* value in TPM2_StartAuthSession(). However, if *userWithAuth* is CLEAR, then the object may be used as the bind object in TPM2_StartAuthSession() but the session cannot be used to authorize actions on the object. If this were allowed, then the *userWithAuth* control could be circumvented simply by using the object as the bind object.

Creation may be SET or CLEAR in template

Load may be SET or CLEAR

Import may be SET or CLEAR

External may be SET or CLEAR

8.3.3.7 Bit[7] – *adminWithPolicy*

If CLEAR, authorization for operations that require ADMIN role may be given if the caller provides proof of knowledge of the *authValue* of the object with an HMAC authorization session or a password.

If this attribute is SET, then then HMAC or password authorizations may not be used for ADMIN role authorizations.

NOTE 1 Regardless of the setting of this attribute, operations that require ADMIN role authorization may be provided by a policy session that satisfies the object's *authPolicy*.

NOTE 2 This attribute is similar to *userWithAuth* but the logic is a bit different. When *userWithAuth* is CLEAR, the *authValue* may not be used for USER mode authorizations. When *adminWithPolicy* is CLEAR, it means that the *authValue* may be used for ADMIN role. Policy may always be used regardless of the setting of *userWithAuth* or *adminWithPolicy*.

 Copyright © TCG 2006-2020

Actions that always require policy (TPM2_Duplicate()) are not affected by the setting of this attribute.

Creation may be SET or CLEAR in *template*

Load may be SET or CLEAR

Import may be SET or CLEAR

External may be SET or CLEAR

8.3.3.8 Bit[10] – *noDA*

If SET, then authorization failures for the object do not affect the dictionary attack protection logic and authorization of the object is not blocked if the TPM is in lockout.

Creation may be SET or CLEAR in *template*

Load may be SET or CLEAR

Import may be SET or CLEAR

External may be SET or CLEAR

8.3.3.9 Bit[11] – *encryptedDuplication*

If SET, then when the object is duplicated, the sensitive portion of the object is required to be encrypted with an inner wrapper and the new parent shall be an asymmetric key and not TPM_RH_NULL

NOTE 1 Enforcement of these requirements in TPM2_Duplicate() is by not allowing *symmetricAlg* to be TPM_ALG_NULL and not allowing *newParentHandle* to be TPM_RH_NULL.

This attribute shall not be SET in any object that has *fixedTPM* SET**.**

NOTE 2 This requirement means that *encryptedDuplication* may not be SET if the object cannot be directly or indirectly duplicated.

If an object's parent has *fixedTPM* SET, and the object is duplicable (*fixedParent* == CLEAR), then *encryptedDuplication* may be SET or CLEAR in the object.

NOTE 3 This allows the object at the boundary between duplicable and non-duplicable objects to have either setting.

If an object's parent has *fixedTPM* CLEAR, then the object is required to have the same setting of *encryptedDuplication* as its parent.

NOTE 4 This requirement forces all duplicable objects in a duplication group to have the same *encryptedDuplication* setting.

Creation shall be CLEAR if *fixedTPM* is SET. If *fixedTPM* is CLEAR, then this attribute shall have the same value as its parent unless *fixedTPM* is SET in the object's parent, in which case, it may be SET or CLEAR.

Load shall be CLEAR if *fixedTPM* is SET. If *fixedTPM* is CLEAR, then this attribute shall have the same value as its parent, unless *fixedTPM* is SET the parent, in which case, it may be SET or CLEAR.

Import if *fixedTPM* is SET in the object's new parent, then this attribute may be SET or CLEAR, otherwise, it shall have the same setting as the new parent.

External may be SET or CLEAR.

 Copyright © TCG 2006-2020

8.3.3.10 Bit[16] – *restricted*

This this attribute modifies the *decrypt* and *sign* attributes of an object.

NOTE A key with this object CLEAR may not be a parent for another object.

Creation shall be CLEAR in *template* if neither sign nor decrypt is SET in *template*.

Load shall be CLEAR if neither sign nor decrypt is SET in the object

Import may be SET or CLEAR

External shall be CLEAR

8.3.3.11 Bit[17] – *decrypt*

When SET, the private portion of this key can be used to decrypt an external blob. If *restricted* is SET, then the TPM will return an error if the external decrypted blob is not formatted as appropriate for the command.

NOTE 1 Since TPM-generated keys and sealed data will contain a hash and a structure tag, the TPM can ensure that it is not being used to improperly decrypt and return sensitive data that should not be returned. The only type of data that may be returned after decryption is a Sealed Data Object (a *keyedHash* object with *decrypt* and *sign* CLEAR).

When *restricted* is CLEAR, there are no restrictions on the use of the private portion of the key for decryption and the key may be used to decrypt and return any structure encrypted by the public portion of the key.

NOTE 2 A key with this attribute SET may be a parent for another object if *restricted* is SET and *sign* is CLEAR.

If *decrypt* is SET on an object with *type* set to TPM_ALG_KEYEDHASH, it indicates that the object is an XOR encryption key.

Creation may be SET or CLEAR in *template*

Load may be SET or CLEAR

Import may be SET or CLEAR

External may be SET or CLEAR

 Copyright © TCG 2006-2020

8.3.3.12 Bit[18] – *sign / encrypt*

When SET, the private portion of this key may be used to sign a digest if the key is an asymetric key or to encrypt a block of data if the key is a symmetric key. If *restricted* is SET, then the asymmetric key may only be used to sign a digest that was computed by the TPM. A restricted symmetric key may only be used to encrypt a data block. If a structure is generated by the TPM, it will begin with TPM_GENERATED_VALUE and the TPM may sign the digest of that structure. If the data is externally supplied and has TPM_GENERATED_VALUE as its first octets, then the TPM will not sign a digest of that data with a restricted signing key.

If *restricted* is CLEAR, then the key may be used to sign any digest or encrypt any data block, whether generated by the TPM or externally provided.

NOTE 1 Some asymmetric algorithms may not support both *sign* and *decrypt* being SET in the same key.

If *sign* is SET on an object with *type* set to TPM_ALG_KEYEDHASH, it indicates that the object is an HMAC key.

NOTE 2 A key with this attribute SET may not be a parent for another object.

Creation shall not be SET if *decrypt* and *restricted* are both SET

Load shall not be SET if *decrypt* and *restricted* are both SET

Import shall not be SET if *decrypt* and *restricted* are both SET

External shall not be SET if *decrypt* and *restricted* are both SET

8.3.3.13 Bit[19] – *x509sign*

When SET, the private portion of the asymmetric key may not be used as the signing key in TPM2_Sign(). This restriction is to ensure that the only digest signed by this key is a digest of a strucure that is specific to the TPM or an x509 certiticate.

NOTE 1 This attribute does not limit the use of the key in any command other than TPM2_Sign().

NOTE 2 This attribute was added in revision 01.53.

This attribute may not be SET if the object is not an asymmetric key or if *sign* is CLEAR.

Creation shall not be SET if *sign* is CLEAR or if the object is not an asymmetric key

Load shall not be SET if *sign* is CLEAR or if the object is not an asymmetric key

Import shall not be SET if *sign* is CLEAR or if the object is not an asymmetric key

External shall not be SET if *sign* is CLEAR or if the object is not an asymmetric key

 Copyright © TCG 2006-2020

8.4 TPMA_SESSION (Session Attributes)

This octet in each session is used to identify the session type, indicate its relationship to any handles in the command, and indicate its use in parameter encryption.

If a session is not being used for authorization, at least one of decrypt, encrypt, or audit must be SET.

In this revision, if *audit* is CLEAR, *auditExclusive* must be CLEAR in the command and will be CLEAR in the response. In a future, revision, this bit may have a different meaning if *audit* is CLEAR. See "Exclusive Audit Session" clause in TPM 2.0 Part 1.

In this revision, if *audit* is CLEAR, *auditReset* must be clear in the command and will be CLEAR in the response. In a future, revision, this bit may have a different meaning if *audit* is CLEAR.

decrypt may only be SET in one session per command. It may only be SET if the first parameter of the command is a sized buffer (TPM2B_).

encrypt may only be SET in one session per command. It may only be SET if the first parameter of the response is a sized buffer (TPM2B_).

audit may only be SET in one session per command or response.

Table 32 — Definition of (UINT8) TPMA_SESSION Bits <IN/OUT>

Bit	Name	Meaning
0	continueSession	**SET (1):** In a command, this setting indicates that the session is to remain active after successful completion of the command. In a response, it indicates that the session is still active. If SET in the command, this attribute shall be SET in the response. **CLEAR (0):** In a command, this setting indicates that the TPM should close the session and flush any related context when the command completes successfully. In a response, it indicates that the session is closed and the context is no longer active. This attribute has no meaning for a password authorization and the TPM will allow any setting of the attribute in the command and SET the attribute in the response. This attribute will only be CLEAR in one response for a logical session. If the attribute is CLEAR, the context associated with the session is no longer in use and the space is available. A session created after another session is ended may have the same handle but logically is not the same session. This attribute has no effect if the command does not complete successfully.
1	auditExclusive	**SET (1):** In a command, this setting indicates that the command should only be executed if the session is exclusive at the start of the command. In a response, it indicates that the session is exclusive. This setting is only allowed if the *audit* attribute is SET (TPM_RC_ATTRIBUTES). **CLEAR (0):** In a command, indicates that the session need not be exclusive at the start of the command. In a response, indicates that the session is not exclusive.
2	auditReset	**SET (1):** In a command, this setting indicates that the audit digest of the session should be initialized and the exclusive status of the session SET. This setting is only allowed if the *audit* attribute is SET (TPM_RC_ATTRIBUTES). **CLEAR (0):** In a command, indicates that the audit digest should not be initialized. This bit is always CLEAR in a response.
4:3	Reserved	shall be CLEAR

 Copyright © TCG 2006-2020

Bit	Name	Meaning
5	decrypt	**SET (1):** In a command, this setting indicates that the first parameter in the command is symmetrically encrypted using the parameter encryption scheme described in TPM 2.0 Part 1. The TPM will decrypt the parameter after performing any HMAC computations and before unmarshaling the parameter. In a response, the attribute is copied from the request but has no effect on the response. **CLEAR (0):** Session not used for encryption. For a password authorization, this attribute will be CLEAR in both the command and response. This attribute may be SET in a session that is not associated with a command handle. Such a session is provided for purposes of encrypting a parameter and not for authorization. This attribute may be SET in combination with any other session attributes.
6	encrypt	**SET (1):** In a command, this setting indicates that the TPM should use this session to encrypt the first parameter in the response. In a response, it indicates that the attribute was set in the command and that the TPM used the session to encrypt the first parameter in the response using the parameter encryption scheme described in TPM 2.0 Part 1. **CLEAR (0):** Session not used for encryption. For a password authorization, this attribute will be CLEAR in both the command and response. This attribute may be SET in a session that is not associated with a command handle. Such a session is provided for purposes of encrypting a parameter and not for authorization.
7	audit	**SET (1):** In a command or response, this setting indicates that the session is for audit and that *auditExclusive* and *auditReset* have meaning. This session may also be used for authorization, encryption, or decryption. The *encrypted* and *encrypt* fields may be SET or CLEAR. **CLEAR (0):** Session is not used for audit. If SET in the command, then this attribute will be SET in the response.

 Copyright © TCG 2006-2020

8.5 TPMA_LOCALITY (Locality Attribute)

In a TPMS_CREATION_DATA structure, this structure is used to indicate the locality of the command that created the object. No more than one of the locality attributes shall be set in the creation data.

When used in TPM2_PolicyLocality(), this structure indicates which localities are approved by the policy. When a policy is started, all localities are allowed. If TPM2_PolicyLocality() is executed, it indicates that the command may only be executed at specific localities. More than one locality may be selected.

EXAMPLE 1 TPM_LOC_TWO would indicate that only locality 2 is authorized.

EXAMPLE 2 TPM_LOC_ONE + TPM_LOC_TWO would indicate that locality 1 or 2 is authorized.

EXAMPLE 3 TPM_LOC_FOUR + TPM_LOC_THREE would indicate that localities 3 or 4 are authorized.

EXAMPLE 4 A value of 21_{16} would represent a locality of 33.

NOTE Locality values of 5 through 31 are not selectable.

If Extended is non-zero, then an extended locality is indicated and the TPMA_LOCALITY contains an integer value.

Table 33 — Definition of (UINT8) TPMA_LOCALITY Bits <IN/OUT>

Bit	Name	Definition
0	TPM_LOC_ZERO	
1	TPM_LOC_ONE	
2	TPM_LOC_TWO	
3	TPM_LOC_THREE	
4	TPM_LOC_FOUR	
7:5	Extended	If any of these bits is set, an extended locality is indicated

 Copyright © TCG 2006-2020

8.6 TPMA_PERMANENT

The attributes in this structure are persistent and are not changed as a result of _TPM_Init or any TPM2_Startup(). Some of the attributes in this structure may change as the result of specific Protected Capabilities. This structure may be read using TPM2_GetCapability(*capability* = TPM_CAP_TPM_PROPERTIES, *property* = TPM_PT_PERMANENT).

Table 34 — Definition of (UINT32) TPMA_PERMANENT Bits <OUT>

Bit	Parameter	Description
0	ownerAuthSet	**SET (1):** TPM2_HierarchyChangeAuth() with *ownerAuth* has been executed since the last TPM2_Clear(). **CLEAR (0):** o*wnerAuth* has not been changed since TPM2_Clear().
1	endorsementAuthSet	**SET (1):** TPM2_HierarchyChangeAuth() with *endorsementAuth* has been executed since the last TPM2_Clear(). **CLEAR (0):** endorsementAuth has not been changed since TPM2_Clear().
2	lockoutAuthSet	**SET (1):** TPM2_HierarchyChangeAuth() with *lockoutAuth* has been executed since the last TPM2_Clear(). **CLEAR (0):** *lockoutAuth* has not been changed since TPM2_Clear().
7:3	Reserved	
8	disableClear	**SET (1):** TPM2_Clear() is disabled. **CLEAR (0):** TPM2_Clear() is enabled. NOTE See “TPM2_ClearControl” in TPM 2.0 Part 3 for details on changing this attribute.
9	inLockout	**SET (1):** The TPM is in lockout, when *failedTries* is equal to *maxTries*.
10	tpmGeneratedEPS	**SET (1):** The EPS was created by the TPM. **CLEAR (0):** The EPS was created outside of the TPM using a manufacturer-specific process.
31:11	Reserved	

 Copyright © TCG 2006-2020

8.7 TPMA_STARTUP_CLEAR

This structure may be read using TPM2_GetCapability(*capability* = TPM_CAP_TPM_PROPERTIES, *property* = TPM_PT_STARTUP_CLEAR).

phEnable is SET on any TPM2_Startup. *shEnable*, *ehEnable*, and *phEnableNV* are SET on TPM Reset or TPM_Restart and preserved by TPM Resume.

Some of attributes may be changed as the result of specific Protected Capabilities.

Table 35 — Definition of (UINT32) TPMA_STARTUP_CLEAR Bits <OUT>

Bit	Parameter	Description
0	phEnable	**SET (1):** The platform hierarchy is enabled and *platformAuth* or *platformPolicy* may be used for authorization. **CLEAR (0):** *platformAuth* and *platformPolicy* may not be used for authorizations, and objects in the platform hierarchy, including persistent objects, cannot be used. NOTE See "TPM2_HierarchyControl" in TPM 2.0 Part 3 for details on changing this attribute.
1	shEnable	**SET (1):** The Storage hierarchy is enabled and *ownerAuth* or *ownerPolicy* may be used for authorization. NV indices defined using owner authorization are accessible. **CLEAR (0):** *ownerAuth* and *ownerPolicy* may not be used for authorizations, and objects in the Storage hierarchy, persistent objects, and NV indices defined using owner authorization cannot be used. NOTE See "TPM2_HierarchyControl" in TPM 2.0 Part 3 for details on changing this attribute.
2	ehEnable	**SET (1):** The EPS hierarchy is enabled and Endorsement Authorization may be used to authorize commands. **CLEAR (0):** Endorsement Authorization may not be used for authorizations, and objects in the endorsement hierarchy, including persistent objects, cannot be used. NOTE See "TPM2_HierarchyControl" in TPM 2.0 Part 3 for details on changing this attribute.
3	phEnableNV	**SET (1):** NV indices that have TPMA_NV_PLATFORMCREATE SET may be read or written. The platform can create define and undefine indices. **CLEAR (0):** NV indices that have TPMA_NV_PLATFORMCREATE SET may not be read or written (TPM_RC_HANDLE). The platform cannot define (TPM_RC_HIERARCHY) or undefined (TPM_RC_HANDLE) indices. NOTE See "TPM2_HierarchyControl" in TPM 2.0 Part 3 for details on changing this attribute. NOTE read refers to these commands: TPM2_NV_Read, TPM2_NV_ReadPublic, TPM_NV_Certify, TPM2_PolicyNV write refers to these commands: TPM2_NV_Write, TPM2_NV_Increment, TPM2_NV_Extend, TPM2_NV_SetBits NOTE The TPM must query the index TPMA_NV_PLATFORMCREATE attribute to determine whether phEnableNV is applicable. Since the TPM will return TPM_RC_HANDLE if the index does not exist, it also returns this error code if the index is disabled. Otherwise, the TPM would leak the existence of an index even when disabled.
30:4	Reserved	shall be zero

 Copyright © TCG 2006-2020

Bit	Parameter	Description
31	orderly	**SET (1):** The TPM received a TPM2_Shutdown() and a matching TPM2_Startup(). **CLEAR (0):** TPM2_Startup(TPM_SU_CLEAR) was not preceded by a TPM2_Shutdown() of any type. NOTE A shutdown is orderly if the TPM receives a TPM2_Shutdown() of any type followed by a TPM2_Startup() of any type. However, the TPM will return an error if TPM2_Startup(TPM_SU_STATE) was not preceded by TPM2_Shutdown(TPM_SU_STATE).

8.8 TPMA_MEMORY

This structure of this attribute is used to report the memory management method used by the TPM for transient objects and authorization sessions. This structure may be read using TPM2_GetCapability(*capability* = TPM_CAP_TPM_PROPERTIES, *property* = TPM_PT_MEMORY).

If the RAM memory is shared, then context save of a session may make it possible to load an additional transient object.

Table 36 — Definition of (UINT32) TPMA_MEMORY Bits <Out>

Bit	Name	Definition
0	sharedRAM	**SET (1):** indicates that the RAM memory used for authorization session contexts is shared with the memory used for transient objects **CLEAR (0):** indicates that the memory used for authorization sessions is not shared with memory used for transient objects
1	sharedNV	**SET (1):** indicates that the NV memory used for persistent objects is shared with the NV memory used for NV Index values **CLEAR (0):** indicates that the persistent objects and NV Index values are allocated from separate sections of NV
2	objectCopiedToRam	**SET (1):** indicates that the TPM copies persistent objects to a transient-object slot in RAM when the persistent object is referenced in a command. The TRM is required to make sure that an object slot is available. **CLEAR (0):** indicates that the TPM does not use transient-object slots when persistent objects are referenced
31:3	Reserved	shall be zero

 Copyright © TCG 2006-2020

8.9 TPMA_CC (Command Code Attributes)

8.9.1 Introduction

This structure defines the attributes of a command from a context management perspective. The fields of the structure indicate to the TPM Resource Manager (TRM) the number of resources required by a command and how the command affects the TPM's resources.

This structure is only used in a list returned by the TPM in response to TPM2_GetCapability(capability = TPM_CAP_COMMANDS).

For a command to the TPM, only the *commandIndex* field and *V* attribute are allowed to be non-zero.

8.9.2 Structure Definition

Table 37 — Definition of (TPM_CC) TPMA_CC Bits <OUT>

Bit	Name	Definition
15:0	commandIndex	indicates the command being selected
21:16	Reserved	shall be zero
22	nv	**SET (1):** indicates that the command may write to NV **CLEAR (0):** indicates that the command does not write to NV
23	extensive	**SET (1):** This command could flush any number of loaded contexts. **CLEAR (0):** no additional changes other than indicated by the *flushed* attribute
24	flushed	**SET (1):** The context associated with any transient handle in the command will be flushed when this command completes. **CLEAR (0):** No context is flushed as a side effect of this command.
27:25	cHandles	indicates the number of the handles in the handle area for this command
28	rHandle	**SET (1):** indicates the presence of the handle area in the response
29	V	**SET (1):** indicates that the command is vendor-specific **CLEAR (0):** indicates that the command is defined in a version of this specification
31:30	Res	allocated for software; shall be zero

8.9.3 Field Descriptions

8.9.3.1 Bits[15:0] – *commandIndex*

This is the command index of the command in the set of commands. The two sets are defined by the *V* attribute. If *V* is zero, then the *commandIndex* shall be in the set of commands defined in a version of this specification. If *V* is one, then the meaning of *commandIndex* is as determined by the TPM vendor.

8.9.3.2 Bit[22] – *nv*

If this attribute is SET, then the TPM may perform an NV write as part of the command actions. This write is independent of any write that may occur as a result of dictionary attack protection. If this attribute is CLEAR, then the TPM shall not perform an NV write as part of the command actions.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

8.9.3.3 Bit[23] – *extensive*

If this attribute is SET, then the TPM may flush many transient objects as a side effect of this command. In TPM 2.0 Part 3, a command that has this attribute is indicated by using a "{E}" decoration in the "Description" column of the *commandCode* parameter.

EXAMPLE See "TPM2_Clear" in TPM 2.0 Part 3.

NOTE The "{E}" decoration may be combined with other decorations such as "{NV}" in which case the decoration would be "{NV E}."

8.9.3.4 Bit[24] – *flushed*

If this attribute is SET, then the TPM will flush transient objects as a side effect of this command. Any transient objects listed in the handle area of the command will be flushed from TPM memory. Handles associated with persistent objects, sessions, PCR, or other fixed TPM resources are not flushed.

NOTE The TRM is expected to use this value to determine how many objects are loaded into transient TPM memory.

NOTE The "{F}" decoration may be combined with other decorations such as "{NV}" in which case the decoration would be "{NV F}."

If this attribute is SET for a command, and the handle of the command is associated with a hierarchy (TPM_RH_PLATFORM, TPM_RH_OWNER, or TPM_RH_ENDORSEMENT), all loaded objects in the indicated hierarchy are flushed.

The TRM is expected to know the behaviour of TPM2_ContextSave(), and sessions are flushed when context saved, but objects are not. The *flushed* attribute for that command shall be CLEAR.

In TPM 2.0 Part 3, a command that has this attribute is indicated by using a "{F}" decoration in the "Description" column of the *commandCode* parameter.

EXAMPLE See "TPM2_SequenceComplete" in TPM 2.0 Part 3."

8.9.3.5 Bits[27:25] – *cHandles*

This field indicates the number of handles in the handle area of the command. This number allows the TRM to enumerate the handles in the handle area and find the position of the authorizations (if any).

8.9.3.6 Bit[28] – *rHandle*

If this attribute is SET, then the response to this command has a handle area. This area will contain no more than one handle. This field is necessary to allow the TRM to locate the *parameterSize* field in the response, which is then used to locate the authorizations.

NOTE The TRM is expected to "virtualize" the handle value for any returned handle.

A TPM command is only allowed to have one handle in the response handle area.

8.9.3.7 Bit[29] – V

When this attribute is SET, it indicates that the command operation is defined by the TPM vendor. When CLEAR, it indicates that the command is defined by a version of this specification.

 Copyright © TCG 2006-2020

8.9.3.8 Bits[31:30] – Res

This field is reserved for system software. This field is required to be zero for a command to the TPM.

8.10 TPMA_MODES

This structure of this attribute is used to report that the TPM is designed for these modes. This structure may be read using TPM2_GetCapability(*capability* = TPM_CAP_TPM_PROPERTIES, *property* = TPM_PT_MODES).

NOTE: To determine the certification status of a TPM with the FIPS_140_2 attribute SET, consult the NIST Module Validation List at http://csrc.nist.gov/groups/STM/cmvp/validation.html.

Table 38 — Definition of (UINT32) TPMA_MODES Bits <Out>

Bit	Name	Definition
0	FIPS_140_2	**SET (1):** indicates that the TPM is designed to comply with all of the FIPS 140-2 requirements at Level 1 or higher.
31:1	Reserved	shall be zero

 Copyright © TCG 2006-2020

8.11 TPMA_X509_KEY_USAGE

These attributes are as specified in clause 4.2.1.3. of RFC 5280 *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.* For TPM2_CertifyX509, when a caller provides a DER encoded Key Usage in *partialCertificate*, the TPM will validate that the key to be certified meets the requirements of Key Usage.

RFC 5280 describes these attributes in terms of how the public key in the certificate should be used. The TPM needs to check that the attributes of the key allow the private part of the key to be used for a purpose that is complimentary to the use of the public key. That is, if the public key should be used to verify signatures, the private key needs to be able to create the signatures (have *sign* SET).

This structure is defined to provide labels of the attributes for use by the TPM code that validates the attributes. This structure is input to the TPM as a DER encoded structure and not in the normal, TPM-canonical form.

This structure is only input to the TPM in a DER-encoded structure and is not present on the interface in canonical TPM format.

Table 39 — Definition of (UINT32) TPMA_X509_KEY_USAGE Bits<>

Bit	Atrribute	Requirements
22:0	Reserved	
23	decipherOnly	Attributes.Decrypt SET
24	encipherOnly	Attributes.Decrypt SET
25	cRLSign	Attributes.sign SET
26	keyCertSign	Attributes.sign SET
27	keyAgreement	Attributes.Decrypt SET
28	dataEncipherment	Attributes.Decrypt SET
29	keyEncipherment	asymmetric key with *decrypt* and *restricted* SET – key has the attributes of a parent key
30	nonrepudiation/contentCommitment	*fixedTPM* SET in Subject Key (*objectHandle*)
31	digitalSignature	*sign* SET in Subject Key (*objectHandle*)

 Copyright © TCG 2006-2020

8.12 TPMA_ACT

This attribute is used to report the ACT state. This attribute may be read using TPM2_GetCapability(*capability* = TPM_CAP_ACT, *property* = TPM_RH_ACT_”x” where “x” is the ACT number (0-F)). The *signaled* value must be preserved across TPM Resume or if the TPM has not lost power. The *signaled* value may be preserved over a power cycle of a TPM.

NOTE: The ACT signaled value is reset to zero when the ACT is next accessed by TPM2_ACT_SetTimeout() with a non-zero *startTimeout*.

Table 39 — Definition of (UINT32) TPMA_ACT Bits

Bit	Name	Definition
0	signaled	**SET (1):** The ACT has signaled **CLEAR (0):** The ACT has not signaled
1	preserveSignaled	**SET (1):** The ACT signaled bit is preserved over a power cycle **CLEAR (0):** The ACT signaled bit is not preserved over a power cycle
31:2	Reserved	shall be zero

 Copyright © TCG 2006-2020

9 Interface Types

9.1 Introduction

Clause 8.11 contains definitions for interface types. An interface type is type checked when it is unmarshaled. These types are based on an underlying type that is indicated in the table title by the value in parentheses. When an interface type is used, the base type is unmarshaled and then checked to see if it has one of the allowed values.

9.2 TPMI_YES_NO

This interface type is used in place of a Boolean type in order to eliminate ambiguity in the handling of a octet that conveys a single bit of information. This type only has two allowed values, YES (1) and NO (0).

NOTE This list is not used as input to the TPM.

Table 40 — Definition of (BYTE) TPMI_YES_NO Type

Value	Description
NO	a value of 0
YES	a value of 1
#TPM_RC_VALUE	

9.3 TPMI_DH_OBJECT

The TPMI_DH_OBJECT interface type is a handle that references a loaded object. The handles in this set are used to refer to either transient or persistent object. The range of these values would change according to the TPM implementation.

NOTE These interface types should not be used by system software to qualify the keys produced by the TPM. The value returned by the TPM shall be used to reference the object.

Table 41 — Definition of (TPM_HANDLE) TPMI_DH_OBJECT Type

Values	Comments
{TRANSIENT_FIRST:TRANSIENT_LAST}	allowed range for transient objects
{PERSISTENT_FIRST:PERSISTENT_LAST}	allowed range for persistent objects
+TPM_RH_NULL	the conditional value
#TPM_RC_VALUE	

 Copyright © TCG 2006-2020

9.4 TPMI_DH_PARENT

The TPMI_DH_PARENT interface type is a handle that references an object that can be the parent of another object. The handles in this set may refer to either transient or persistent object or to Primary Seeds.

Table 42 — Definition of (TPM_HANDLE) TPMI_DH_PARENT Type

Values	Comments
{TRANSIENT_FIRST:TRANSIENT_LAST}	allowed range for transient objects
{PERSISTENT_FIRST:PERSISTENT_LAST}	allowed range for persistent objects
TPM_RH_OWNER	Storage hierarchy
TPM_RH_PLATFORM	Platform hierarchy
TPM_RH_ENDORSEMENT	Endorsement hierarchy
+TPM_RH_NULL	no hierarchy
#TPM_RC_VALUE	

9.5 TPMI_DH_PERSISTENT

The TPMI_DH_PERSISTENT interface type is a handle that references a location for a transient object. This type is used in TPM2_EvictControl() to indicate the handle to be assigned to the persistent object.

Table 43 — Definition of (TPM_HANDLE) TPMI_DH_PERSISTENT Type

Values	Comments
{PERSISTENT_FIRST:PERSISTENT_LAST}	allowed range for persistent objects
#TPM_RC_VALUE	

 Copyright © TCG 2006-2020

9.6 TPMI_DH_ENTITY

The TPMI_DH_ENTITY interface type is TPM-defined values that are used to indicate that the handle refers to an *authValue*. The range of these values would change according to the TPM implementation.

Table 44 — Definition of (TPM_HANDLE) TPMI_DH_ENTITY Type <IN>

Values	Comments
TPM_RH_OWNER	
TPM_RH_ENDORSEMENT	
TPM_RH_PLATFORM	
TPM_RH_LOCKOUT	
{TRANSIENT_FIRST : TRANSIENT_LAST}	range of object handles
{PERSISTENT_FIRST : PERSISTENT_LAST}	
{NV_INDEX_FIRST : NV_INDEX_LAST}	
{PCR_FIRST : PCR_LAST}	
{TPM_RH_AUTH_00 : TPM_RH_AUTH_FF}	range of vendor-specific authorization values
+TPM_RH_NULL	conditional value
#TPM_RC_VALUE	

9.7 TPMI_DH_PCR

This interface type consists of the handles that may be used as PCR references. The upper end of this range of values would change according to the TPM implementation.

NOTE 1 Typically, the 0th PCR will have a handle value of zero.

NOTE 2 The handle range for PCR is defined to be the same as the handle range for PCR in previous versions of TPM specifications.

Table 45 — Definition of (TPM_HANDLE) TPMI_DH_PCR Type <IN>

Values	Comments
{PCR_FIRST:PCR_LAST}	
+TPM_RH_NULL	conditional value
#TPM_RC_VALUE	

 Copyright © TCG 2006-2020

9.8 TPMI_SH_AUTH_SESSION

The TPMI_SH_AUTH_SESSION interface type is TPM-defined values that are used to indicate that the handle refers to an authorization session.

Table 46 — Definition of (TPM_HANDLE) TPMI_SH_AUTH_SESSION Type <IN/OUT>

Values	Comments
{HMAC_SESSION_FIRST : HMAC_SESSION_LAST}	range of HMAC authorization session handles
{POLICY_SESSION_FIRST: POLICY_SESSION_LAST}	range of policy authorization session handles
+TPM_RS_PW	a password authorization
#TPM_RC_VALUE	error returned if the handle is out of range

9.9 TPMI_SH_HMAC

This interface type is used for an authorization handle when the authorization session uses an HMAC.

Table 47 — Definition of (TPM_HANDLE) TPMI_SH_HMAC Type <IN/OUT>

Values	Comments
{HMAC_SESSION_FIRST: HMAC_SESSION_LAST}	range of HMAC authorization session handles
#TPM_RC_VALUE	error returned if the handle is out of range

9.10 TPMI_SH_POLICY

This interface type is used for a policy handle when it appears in a policy command.

Table 48 — Definition of (TPM_HANDLE) TPMI_SH_POLICY Type <IN/OUT>

Values	Comments
{POLICY_SESSION_FIRST: POLICY_SESSION_LAST}	range of policy authorization session handles
#TPM_RC_VALUE	error returned if the handle is out of range

9.11 TPMI_DH_CONTEXT

This type defines the handle values that may be used in TPM2_ContextSave() or TPM2_Flush().

Table 49 — Definition of (TPM_HANDLE) TPMI_DH_CONTEXT Type

Values	Comments
{HMAC_SESSION_FIRST : HMAC_SESSION_LAST}	
{POLICY_SESSION_FIRST:POLICY_SESSION_LAST}	
{TRANSIENT_FIRST:TRANSIENT_LAST}	
#TPM_RC_VALUE	

 Copyright © TCG 2006-2020

9.12 TPMI_DH_SAVED

This type defines the handle values that may be used in TPM2_ContextSave() or TPM2_FlushContext().

Table 50 — Definition of (TPM_HANDLE) TPMI_DH_SAVED Type

Values	Comments
{HMAC_SESSION_FIRST : HMAC_SESSION_LAST}	an HMAC session context
{POLICY_SESSION_FIRST:POLICY_SESSION_LAST}	a policy session context
0x80000000	an ordinary transient object
0x80000001	a sequence object
0x80000002	a transient object with the *stClear* attribute SET
#TPM_RC_VALUE	

9.13 TPMI_RH_HIERARCHY

The TPMI_RH_HIERARCHY interface type is used as the type of a handle in a command when the handle is required to be one of the hierarchy selectors.

Table 51 — Definition of (TPM_HANDLE) TPMI_RH_HIERARCHY Type

Values	Comments
TPM_RH_OWNER	Storage hierarchy
TPM_RH_PLATFORM	Platform hierarchy
TPM_RH_ENDORSEMENT	Endorsement hierarchy
+TPM_RH_NULL	no hierarchy
#TPM_RC_VALUE	response code returned when the unmarshaling of this type fails

9.14 TPMI_RH_ENABLES

The TPMI_RH_ENABLES interface type is used as the type of a handle in a command when the handle is required to be one of the hierarchy or NV enables.

Table 52 — Definition of (TPM_HANDLE) TPMI_RH_ENABLES Type

Values	Comments
TPM_RH_OWNER	Storage hierarchy
TPM_RH_PLATFORM	Platform hierarchy
TPM_RH_ENDORSEMENT	Endorsement hierarchy
TPM_RH_PLATFORM_NV	Platform NV
+TPM_RH_NULL	no hierarchy
#TPM_RC_VALUE	response code returned when the unmarshaling of this type fails

 Copyright © TCG 2006-2020

9.15 TPMI_RH_HIERARCHY_AUTH

This interface type is used as the type of a handle in a command when the handle is required to be one of the hierarchy selectors or the Lockout Authorization.

Table 53 — Definition of (TPM_HANDLE) TPMI_RH_HIERARCHY_AUTH Type <IN>

Values	Comments
TPM_RH_OWNER	Storage hierarchy
TPM_RH_PLATFORM	Platform hierarchy
TPM_RH_ENDORSEMENT	Endorsement hierarchy
TPM_RH_LOCKOUT	Lockout Authorization
#TPM_RC_VALUE	response code returned when the unmarshaling of this type fails

9.16 TPMI_RH_HIERARCHY_POLICY

This interface type is used as the type of a handle in a command when the handle is required to be one of the hierarchy selectors, the Lockout Authorization, or an ACT. This type is used in TPM2_SetPrimaryPolicy().

Table 54 — Definition of (TPM_HANDLE) TPMI_RH_HIERARCHY_POLICY Type <IN>

Values	Comments
TPM_RH_OWNER	Storage hierarchy
TPM_RH_PLATFORM	Platform hierarchy
TPM_RH_ENDORSEMENT	Endorsement hierarchy
TPM_RH_LOCKOUT	Lockout Authorization
{TPM_RH_ACT_0:TPM_RH_ACT_F}	Authenticated Countdown Timer
#TPM_RC_VALUE	response code returned when the unmarshaling of this type fails

9.17 TPMI_RH_PLATFORM

The TPMI_RH_PLATFORM interface type is used as the type of a handle in a command when the only allowed handle is TPM_RH_PLATFORM indicating that Platform Authorization is required.

Table 55 — Definition of (TPM_HANDLE) TPMI_RH_PLATFORM Type <IN>

Values	Comments
TPM_RH_PLATFORM	Platform hierarchy
#TPM_RC_VALUE	response code returned when the unmarshaling of this type fails

Copyright © TCG 2006-2020

9.18 TPMI_RH_OWNER

This interface type is used as the type of a handle in a command when the only allowed handle is TPM_RH_OWNER indicating that Owner Authorization is required.

Table 56 — Definition of (TPM_HANDLE) TPMI_RH_OWNER Type <IN>

Values	Comments
TPM_RH_OWNER	Owner hierarchy
+TPM_RH_NULL	may allow the null handle
#TPM_RC_VALUE	response code returned when the unmarshaling of this type fails

9.19 TPMI_RH_ENDORSEMENT

This interface type is used as the type of a handle in a command when the only allowed handle is TPM_RH_ENDORSEMENT indicating that Endorsement Authorization is required.

Table 57 — Definition of (TPM_HANDLE) TPMI_RH_ENDORSEMENT Type <IN>

Values	Comments
TPM_RH_ENDORSEMENT	Endorsement hierarchy
+TPM_RH_NULL	may allow the null handle
#TPM_RC_VALUE	response code returned when the unmarshaling of this type fails

9.20 TPMI_RH_PROVISION

The TPMI_RH_PROVISION interface type is used as the type of the handle in a command when the only allowed handles are either TPM_RH_OWNER or TPM_RH_PLATFORM indicating that either Platform Authorization or Owner Authorization are allowed.

In most cases, either Platform Authorization or Owner Authorization may be used to authorize the commands used for management of the resources of the TPM and this interface type will be used.

Table 58 — Definition of (TPM_HANDLE) TPMI_RH_PROVISION Type <IN>

Value	Comments
TPM_RH_OWNER	handle for Owner Authorization
TPM_RH_PLATFORM	handle for Platform Authorization
#TPM_RC_VALUE	response code returned when the unmarshaling of this type fails

 Copyright © TCG 2006-2020

9.21 TPMI_RH_CLEAR

The TPMI_RH_CLEAR interface type is used as the type of the handle in a command when the only allowed handles are either TPM_RH_LOCKOUT or TPM_RH_PLATFORM indicating that either Platform Authorization or Lockout Authorization are allowed.

This interface type is normally used for performing or controlling TPM2_Clear().

Table 59 — Definition of (TPM_HANDLE) TPMI_RH_CLEAR Type <IN>

Value	Comments
TPM_RH_LOCKOUT	handle for Lockout Authorization
TPM_RH_PLATFORM	handle for Platform Authorization
#TPM_RC_VALUE	response code returned when the unmarshaling of this type fails

9.22 TPMI_RH_NV_AUTH

This interface type is used to identify the source of the authorization for access to an NV location. The handle value of a TPMI_RH_NV_AUTH shall indicate that the authorization value is either Platform Authorization, Owner Authorization, or the *authValue.* This type is used in the commands that access an NV Index (commands of the form TPM2_NV_xxx) other than TPM2_NV_DefineSpace() and TPM2_NV_UndefineSpace().

Table 60 — Definition of (TPM_HANDLE) TPMI_RH_NV_AUTH Type <IN>

Value	Comments
TPM_RH_PLATFORM	Platform Authorization is allowed
TPM_RH_OWNER	Owner Authorization is allowed
{NV_INDEX_FIRST:NV_INDEX_LAST}	range for NV locations
#TPM_RC_VALUE	response code returned when unmarshaling of this type fails

9.23 TPMI_RH_LOCKOUT

The TPMI_RH_LOCKOUT interface type is used as the type of a handle in a command when the only allowed handle is TPM_RH_LOCKOUT indicating that Lockout Authorization is required.

Table 61 — Definition of (TPM_HANDLE) TPMI_RH_LOCKOUT Type <IN>

Value	Comments
TPM_RH_LOCKOUT	handle for Lockout Authorization
#TPM_RC_VALUE	response code returned when the unmarshaling of this type fails

 Copyright © TCG 2006-2020

9.24 TPMI_RH_NV_INDEX

This interface type is used to identify an NV location. This type is used in the NV commands.

Table 62 — Definition of (TPM_HANDLE) TPMI_RH_NV_INDEX Type <IN/OUT>

Value	Comments
{NV_INDEX_FIRST:NV_INDEX_LAST}	Range of NV Indexes
#TPM_RC_VALUE	error returned if the handle is out of range

9.25 TPMI_RH_AC

This interface type is used to identify an attached component. This type is used in the AC commands.

Table 63 — Definition of (TPM_HANDLE) TPMI_RH_AC Type <IN>

Value	Comments
{AC_FIRST:AC_LAST}	Range of AC handles
#TPM_RC_VALUE	error returned if the handle is out of range

 Copyright © TCG 2006-2020

9.26 TPMI_RH_ACT

This interface type is used to identify the ACT instance used in TPM2_ACT_SetTimeout().

Table 64 — Definition of (TPM_HANDLE) TPMI_RH_ACT Type

Value	Comments
{TPM_RH_ACT_0:TPM_RH_ACT_F}	handles for the Authenticated Countdown Timers
#TPM_RC_VALUE	response code returned when the unmarshaling of this type fails

9.27 TPMI_ALG_HASH

A TPMI_ALG_HASH is an interface type of all the hash algorithms implemented on a specific TPM. The selector in Table 65 indicates all of the hash algorithms that have an algorithm ID assigned by the TCG and does not indicate the algorithms that will be accepted by a TPM.

NOTE When implemented, each of the algorithm entries is delimited by #ifdef and #endif so that, if the algorithm is not implemented in a specific TPM, that algorithm is not included in the interface type.

Table 65 — Definition of (TPM_ALG_ID) TPMI_ALG_HASH Type

Values	Comments
TPM_ALG_!ALG.H	all hash algorithms defined by the TCG
+TPM_ALG_NULL	
#TPM_RC_HASH	

9.28 TPMI_ALG_ASYM (Asymmetric Algorithms)

A TPMI_ALG_ASYM is an interface type of all the asymmetric algorithms implemented on a specific TPM. Table 66 lists each of the asymmetric algorithms that have an algorithm ID assigned by the TCG.

Table 66 — Definition of (TPM_ALG_ID) TPMI_ALG_ASYM Type

Values	Comments
TPM_ALG_!ALG.AO	all asymmetric object types
+TPM_ALG_NULL	
#TPM_RC_ASYMMETRIC	

 Copyright © TCG 2006-2020

9.29 TPMI_ALG_SYM (Symmetric Algorithms)

A TPMI_ALG_SYM is an interface type of all the symmetric algorithms that have an algorithm ID assigned by the TCG and are implemented on the TPM.

NOTE The validation code produced by an example script will produce a CASE statement with a case for each of the values in the "Values" column. The case for a value is delimited by a #ifdef/#endif pair so that if the algorithm is not implemented on the TPM, then the case for the algorithm is not generated, and use of the algorithm will cause a TPM error (TPM_RC_SYMMETRIC).

Table 67 — Definition of (TPM_ALG_ID) TPMI_ALG_SYM Type

Values	Comments
TPM_ALG_!ALG.S	all symmetric block ciphers
TPM_ALG_XOR	required
+TPM_ALG_NULL	required to be present in all versions of this table
#TPM_RC_SYMMETRIC	

9.30 TPMI_ALG_SYM_OBJECT

A TPMI_ALG_SYM_OBJECT is an interface type of all the TCG-defined symmetric algorithms that may be used as companion symmetric encryption algorithm for an asymmetric object. All algorithms in this list shall be block ciphers usable in Cipher Feedback (CFB).

NOTE TPM_ALG_XOR is not allowed in this list.

Table 68 — Definition of (TPM_ALG_ID) TPMI_ALG_SYM_OBJECT Type

Values	Comments
TPM_ALG_!ALG.S	all symmetric block ciphers
+TPM_ALG_NULL	required to be present in all versions of this table
#TPM_RC_SYMMETRIC	

9.31 TPMI_ALG_SYM_MODE

A TPMI_ALG_SYM_MODE is an interface type of all the TCG-defined block-cipher modes of operation.

Table 69 — Definition of (TPM_ALG_ID) TPMI_ALG_SYM_MODE Type

Values	Comments
TPM_ALG_!ALG.SE	all symmetric block cipher encryption/decryption modes
TPM_ALG_!ALG.SX	all symmetric block cipher MAC modes
+TPM_ALG_NULL	
#TPM_RC_MODE	

 Copyright © TCG 2006-2020

9.32 TPMI_ALG_KDF (Key and Mask Generation Functions)

A TPMI_ALG_KDF is an interface type of all the key derivation functions implemented on a specific TPM.

Table 70 — Definition of (TPM_ALG_ID) TPMI_ALG_KDF Type

Values	Comments
TPM_ALG_!ALG.HM	all defined hash-based key and mask generation functions
+TPM_ALG_NULL	
#TPM_RC_KDF	

9.33 TPMI_ALG_SIG_SCHEME

This is the definition of the interface type for any signature scheme.

Table 71 — Definition of (TPM_ALG_ID) TPMI_ALG_SIG_SCHEME Type

Values	Comments
TPM_ALG_!ALG.ax	all asymmetric signing schemes including anonymous schemes
TPM_ALG_HMAC	present on all TPM
+TPM_ALG_NULL	
#TPM_RC_SCHEME	response code when a signature scheme is not correct

9.34 TPMI_ECC_KEY_EXCHANGE

This is the definition of the interface type for an ECC key exchange scheme.

NOTE Because of the "{ECC}" in the table title, the only values in this table will be those that are dependent on ECC being implemented, even if they otherwise have the correct type attributes.

Table 72 — Definition of (TPM_ALG_ID){ECC} TPMI_ECC_KEY_EXCHANGE Type

Values	Comments
TPM_ALG_!ALG.AM	any ECC key exchange method
TPM_ALG_SM2	SM2 is typed as signing but may be used as a key-exchange protocol
+TPM_ALG_NULL	
#TPM_RC_SCHEME	response code when a key exchange scheme is not correct

 Copyright © TCG 2006-2020

9.35 TPMI_ST_COMMAND_TAG

This interface type is used for the command tags.

The response code for a bad command tag has the same value as the TPM 1.2 response code (TPM_BAD_TAG). This value is used in case the software is not compatible with this specification and an unexpected response code might have unexpected side effects.

Table 73 — Definition of (TPM_ST) TPMI_ST_COMMAND_TAG Type

Values	Comments
TPM_ST_NO_SESSIONS	
TPM_ST_SESSIONS	
#TPM_RC_BAD_TAG	

9.36 TPMI_ALG_MAC_SCHEME

A TPMI_ALG_MAC_SCHEME is an interface type of all the TCG-defined symmetric algorithms that may be used as companion symmetric! signing algorithm.

Table 74 — Definition of (TPM_ALG_ID) TPMI_ALG_MAC_SCHEME Type

Values	Comments
TPM_ALG_!ALG.SX	all symmetric block cipher MAC algorithms
TPM_ALG_!ALG.H	all hash algorithms defined by the TCG
+TPM_ALG_NULL	required to be present in all versions of this table
#TPM_RC_SYMMETRIC	

9.37 TPMI_ALG_CIPHER_MODE

A TPMI_ALG_CIPHER_MODE is an interface type of all the symmetric block cipher, encryption/decryption modes that are listed in the TCG algorithm registry.

Table 75 — Definition of (TPM_ALG_ID) TPMI_ALG_CIPHER_MODE Type

Values	Comments
TPM_ALG_!ALG.SE	all symmetric block cipher algorithms
+TPM_ALG_NULL	required to be present in all versions of this table
#TPM_RC_MODE	

 Copyright © TCG 2006-2020

10 Structure Definitions

10.1 TPMS_EMPTY

This structure is used as a placeholder. In some cases, a union will have a selector value with no data to unmarshal when that type is selected. Rather than leave the entry empty, TPMS_EMPTY may be selected.

NOTE The tool chain will special case this structure and create the marshaling and unmarshaling code for this structure but not create a type definition. The unmarshaling code for this structure will return TPM_RC_SUCCESS and the marshaling code will return 0.

Table 76 — Definition of TPMS_EMPTY Structure <IN/OUT>

Parameter	Type	Description
		a structure with no member

10.2 TPMS_ALGORITHM_DESCRIPTION

This structure is a return value for a TPM2_GetCapability() that reads the installed algorithms.

Table 77 — Definition of TPMS_ALGORITHM_DESCRIPTION Structure <OUT>

Parameter	Type	Description
alg	TPM_ALG_ID	an algorithm
attributes	TPMA_ALGORITHM	the attributes of the algorithm

10.3 Hash/Digest Structures

10.3.1 TPMU_HA (Hash)

A TPMU_HA is a union of all the hash algorithms implemented on a TPM.

NOTE 1 The !ALG.H and !ALG.H values represent all algorithms defined in the TCG registry as being type “H”.

NOTE 2 If processed by an automated tool, each entry of the table should be qualified (with #ifdef/#endif) so that if the hash algorithm is not implemented on the TPM, the parameter associated with that hash is not present. This will keep the union from being larger than the largest digest of a hash implemented on that TPM.

Table 78 — Definition of TPMU_HA Union <IN/OUT >

Parameter	Type	Selector	Description
!ALG.H [!ALG.H_DIGEST_SIZE]	BYTE	TPM_ALG_!ALG.H	all hashes
null		TPM_ALG_NULL	

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.3.2 TPMT_HA

Table 79 shows the basic hash-agile structure used in this specification. To handle hash agility, this structure uses the *hashAlg* parameter to indicate the algorithm used to compute the digest and, by implication, the size of the digest.

When transmitted, only the number of octets indicated by *hashAlg* is sent.

NOTE In the reference code, when a TPMT_HA is allocated, the digest field is large enough to support the largest hash algorithm in the TPMU_HA union.

Table 79 — Definition of TPMT_HA Structure <IN/OUT>

Parameter	Type	Description
hashAlg	+TPMI_ALG_HASH	selector of the hash contained in the *digest* that implies the size of the *digest* NOTE The leading "+" on the type indicates that this structure should pass an indication to the unmarshaling function for TPMI_ALG_HASH so that TPM_ALG_NULL will be allowed if a use of a TPMT_HA allows TPM_ALG_NULL.
[hashAlg] digest	TPMU_HA	the digest data

10.4 Sized Buffers

10.4.1 Introduction

The "TPM2B_" prefix is used for a structure that has a size field followed by a data buffer with the indicated number of octets. The *size* field is 16 bits.

When the type of the second parameter in a TPM2B_ structure is BYTE, the TPM shall unmarshal the indicated number of octets, which may be zero.

When the type of the second parameter in the TPM2B_ structure is not BYTE, the value of the *size* field shall either be zero indicating that no structure is to be unmarshaled; or it shall be identical to the number of octets unmarshaled for the second parameter.

NOTE 1 If the TPM2B_ defines a structure and not an array of octets, then the structure is self-describing and the TPM will be able to determine how many octets are in the structure when it is unmarshaled. If that number of octets is not equal to the size parameter, then it is an error.

NOTE 2 The reason that a structure may be put into a TPM2B_ is that the parts of the structure may be handled as separate opaque blocks by the application/system software. Rather than require that all of the structures in a command or response be marshaled or unmarshaled sequentially, the size field allows the structure to be manipulated as an opaque block. Placing a structure in a TPM2B_ also makes it possible to use parameter encryption on the structure.

If a TPM2B_ is encrypted, the TPM will encrypt/decrypt the data field of the TPM2B_ but not the *size* parameter. The TPM will encrypt/decrypt the number of octets indicated by the *size* field.

NOTE 3 In the reference implementation, a TPM2B type is defined that is a 16-bit size field followed by a single byte of data. The TPM2B_ is then defined as a union that contains a TPM2B (union member 'b') and the structure in the definition table (union member 't'). This union is used for internally generated structures so that there is a way to define a structure of the correct size (forced by the 't' member) while giving a way to pass the structure generically as a 'b'. Most function calls use the 't' member so that the compiler will generate a warning if there is a type error (a TPM2B_ of the wrong type). Having the type checked helps avoid many issues with buffer overflow caused by a too small buffer being passed to a function.

 Copyright © TCG 2006-2020

10.4.2 TPM2B_DIGEST

This structure is used for a sized buffer that cannot be larger than the largest digest produced by any hash algorithm implemented on the TPM.

As with all sized buffers, the size is checked to see if it is within the prescribed range. If not, the response code is TPM_RC_SIZE.

NOTE For any structure, like the one below, that contains an implied size check, it is implied that TPM_RC_SIZE is a possible response code and the response code will not be listed in the table.

Table 80 — Definition of TPM2B_DIGEST Structure

Parameter	Type	Description
size	UINT16	size in octets of the *buffer* field; may be 0
buffer[size]{:sizeof(TPMU_HA)}	BYTE	the buffer area that can be no larger than a digest

10.4.3 TPM2B_DATA

This structure is used for a data buffer that is required to be no larger than the size of the Name of an object.

Table 81 — Definition of TPM2B_DATA Structure

Parameter	Type	Description
size	UINT16	size in octets of the *buffer* field; may be 0
buffer[size]{:sizeof(TPMT_HA)}	BYTE	

10.4.4 TPM2B_NONCE

Table 82 — Definition of Types for TPM2B_NONCE

Type	Name	Description
TPM2B_DIGEST	TPM2B_NONCE	size limited to the same as the digest structure

10.4.5 TPM2B_AUTH

This structure is used for an authorization value and limits an *authValue* to being no larger than the largest digest produced by a TPM. In order to ensure consistency within an object, the *authValue* may be no larger than the size of the digest produced by the object's *nameAlg*. This ensures that any TPM that can load the object will be able to handle the *authValue* of the object.

Table 83 — Definition of Types for TPM2B_AUTH

Type	Name	Description
TPM2B_DIGEST	TPM2B_AUTH	size limited to the same as the digest structure

 Copyright © TCG 2006-2020

10.4.6 TPM2B_OPERAND

This type is a sized buffer that can hold an operand for a comparison with an NV Index location. The maximum size of the operand is implementation dependent but a TPM is required to support an operand size that is at least as big as the digest produced by any of the hash algorithms implemented on the TPM.

Table 84 — Definition of Types for TPM2B_OPERAND

Type	Name	Description
TPM2B_DIGEST	TPM2B_OPERAND	size limited to the same as the digest structure

10.4.7 TPM2B_EVENT

This type is a sized buffer that can hold event data.

Table 85 — Definition of TPM2B_EVENT Structure

Parameter	Type	Description
size	UINT16	size of the operand *buffer*
buffer [size] {:1024}	BYTE	the operand

10.4.8 TPM2B_MAX_BUFFER

This type is a sized buffer that can hold a maximally sized buffer for commands that use a large data buffer such as TPM2_Hash(), TPM2_SequenceUpdate(), or TPM2_FieldUpgradeData().

NOTE The above list is not comprehensive and other commands may use this buffer type.

MAX_DIGEST_BUFFER is TPM-dependent but is required to be at least 1,024.

Table 86 — Definition of TPM2B_MAX_BUFFER Structure

Parameter	Type	Description
size	UINT16	size of the buffer
buffer [size] {:MAX_DIGEST_BUFFER}	BYTE	the operand

10.4.9 TPM2B_MAX_NV_BUFFER

This type is a sized buffer that can hold a maximally sized buffer for NV data commands such as TPM2_NV_Read(), TPM2_NV_Write(), and TPM2_NV_Certify().

Table 87 — Definition of TPM2B_MAX_NV_BUFFER Structure

Parameter	Type	Description
size	UINT16	size of the buffer
buffer [size] {:MAX_NV_BUFFER_SIZE}	BYTE	the operand NOTE MAX_NV_BUFFER_SIZE is TPM-dependent

 Copyright © TCG 2006-2020

10.4.10 TPM2B_TIMEOUT

This TPM-dependent structure is used to provide the timeout value for an authorization. The *size* shall be 8 or less.

Table 88 — Definition of TPM2B_TIMEOUT Structure

Type	Name	Description
size	UINT16	size of the timeout value
buffer[size]{:sizeof(UINT64)}	BYTE	the timeout value

NOTE In the reference implementation the MSb is used as a flag to indicate whether a ticket expires on TPM Reset or TPM Restart.

10.4.11 TPM2B_IV

This structure is used for passing an initial value for a symmetric block cipher to or from the TPM. The size is set to be the largest block size of any implemented symmetric cipher implemented on the TPM.

Table 89 — Definition of TPM2B_IV Structure <IN/OUT>

Parameter	Type	Description
size	UINT16	size of the IV value This value is fixed for a TPM implementation.
buffer[size]{:MAX_SYM_BLOCK_SIZE}	BYTE	the IV value

10.5 Names

10.5.1 Introduction

The Name of an entity is used in place of the handle in authorization computations. The substitution occurs in *cpHash* and *policyHash* computations.

For an entity that is defined by a public area (objects and NV Indexes), the Name is the hash of the public structure that defines the entity. The hash is done using the *nameAlg* of the entity.

NOTE For an object, a TPMT_PUBLIC defines the entity. For an NV Index, a TPMS_NV_PUBLIC defines the entity.

For entities not defined by a public area, the Name is the handle that is used to refer to the entity.

10.5.2 TPMU_NAME

Table 90 — Definition of TPMU_NAME Union <>

Parameter	Type	Selector	Description
digest	TPMT_HA		when the Name is a digest
handle	TPM_HANDLE		when the Name is a handle

 Copyright © TCG 2006-2020

10.5.3 TPM2B_NAME

This buffer holds a Name for any entity type.

The type of Name in the structure is determined by context and the *size* parameter. If *size* is four, then the Name is a handle. If *size* is zero, then no Name is present. Otherwise, the size shall be the size of a TPM_ALG_ID plus the size of the digest produced by the indicated hash algorithm.

Table 91 — Definition of TPM2B_NAME Structure

Parameter	Type	Description
size	UINT16	size of the Name structure
name[size]{:sizeof(TPMU_NAME)}	BYTE	the Name structure

10.6 PCR Structures

10.6.1 TPMS_PCR_SELECT

This structure provides a standard method of specifying a list of PCR.

PCR numbering starts at zero.

pcrSelect is an array of octets. The octet containing the bit corresponding to a specific PCR is found by dividing the PCR number by 8.

EXAMPLE 1 The bit in *pcrSelect* corresponding to PCR 19 is in *pcrSelect* [2] (19/8 = 2).

The least significant bit in a octet is bit number 0. The bit in the octet associated with a PCR is the remainder after division by 8.

EXAMPLE 2 The bit in *pcrSelect* [2] corresponding to PCR 19 is bit 3 (19 mod 8). If *sizeofSelect* is 3, then the *pcrSelect* array that would specify PCR 19 and no other PCR is 00 00 08_{16}.

Each bit in *pcrSelect* indicates whether the corresponding PCR is selected (1) or not (0). If the *pcrSelect* is all zero bits, then no PCR is selected.

sizeofSelect indicates the number of octets in *pcrSelect*. The allowable value for *sizeofSelect* is determined by the number of PCR required by the applicable platform-specific specification and the number of PCR implemented in the TPM. The minimum value for *sizeofSelect* is:

$$PCR_SELECT_MIN \coloneqq (PLATFORM_PCR + 7) / 8 \quad (1)$$

where

PLATFORM_PCR the number of PCR required by the platform-specific specification

The maximum value for sizeofSelect is:

$$PCR_SELECT_MAX \coloneqq (IMPLEMENTATION_PCR + 7) / 8 \quad (2)$$

where

IMPLEMENTATION_PCR the number of PCR implemented on the TPM

 Copyright © TCG 2006-2020

If the TPM implements more PCR than there are bits in *pcrSelect*, the additional PCR are not selected.

EXAMPLE 3 If the applicable platform-specific specification requires that the TPM have a minimum of 24 PCR but the TPM implements 32, then a PCR select of 3 octets would imply that PCR 24-31 are not selected.

Table 92 — Definition of TPMS_PCR_SELECT Structure

Parameter	Type	Description
sizeofSelect {PCR_SELECT_MIN:}	UINT8	the size in octets of the *pcrSelect* array
pcrSelect [sizeofSelect] {:PCR_SELECT_MAX}	BYTE	the bit map of selected PCR
#TPM_RC_VALUE		

10.6.2 TPMS_PCR_SELECTION

Table 93 — Definition of TPMS_PCR_SELECTION Structure

Parameter	Type	Description
hash	TPMI_ALG_HASH	the hash algorithm associated with the selection
sizeofSelect {PCR_SELECT_MIN:}	UINT8	the size in octets of the *pcrSelect* array
pcrSelect [sizeofSelect] {:PCR_SELECT_MAX}	BYTE	the bit map of selected PCR
#TPM_RC_VALUE		

10.7 Tickets

10.7.1 Introduction

Tickets are evidence that the TPM has previously processed some information. A ticket is an HMAC over the data using a secret key known only to the TPM. A ticket is a way to expand the state memory of the TPM. A ticket is only usable by the TPM that produced it.

The formulations for tickets shown in 10.7 are to be used by a TPM that is compliant with this specification.

The method of creating the ticket data is:

$$\mathbf{HMAC}_{contexAlg}(proof, (ticketType \mid\mid param \{ \mid\mid param \{...\}\})) \quad (3)$$

where

$\mathbf{HMAC}_{contexAlg}()$ an HMAC using the hash used for context integrity

$proof$ a TPM secret value (depends on hierarchy)

$ticketType$ a value to differentiate the tickets

$param$ one or more values that were checked by the TPM

 Copyright © TCG 2006-2020

The proof value used for each hierarchy is shown in Table 94.

Table 94 — Values for *proof* Used in Tickets

Hierarchy	proof	Description
Null	nullProof	a value that changes with every TPM Reset
Platform	phProof	a value that changes with each change of the PPS
Owner	shProof	a value that changes with each change of the SPS
Endorsement	ehProof	a value that changes with each change of either the EPS or SPS

The format for a ticket is shown in Table 95. This is a template for the tickets shown in the remainder of this clause 10.7.

Table 95 — General Format of a Ticket

Parameter	Type	Description
tag	TPM_ST	structure tag indicating the type of the ticket
hierarchy	TPMI_RH_HIERARCHY+	the hierarchy of the proof value
digest	TPM2B_DIGEST	the HMAC over the ticket-specific data

10.7.2 A NULL Ticket

When a command requires a ticket and no ticket is available, the caller is required to provide a structure with a ticket *tag* that is correct for the context. The *hierarchy* shall be set to TPM_RH_NULL, and *digest* shall be the Empty Buffer (a buffer with a size field of zero). This construct is the NULL Ticket. When a response indicates that a ticket is returned, the TPM may return a NULL Ticket.

NOTE Because each use of a ticket requires that the structure tag for the ticket be appropriate for the use, there is no single representation of a NULL Ticket that will work in all circumstances. Minimally, a NULL ticket will have a structure type that is appropriate for the context.

10.7.3 TPMT_TK_CREATION

This ticket is produced by TPM2_Create() or TPM2_CreatePrimary(). It is used to bind the creation data to the object to which it applies. The ticket is computed by

$$\mathbf{HMAC}_{contextAlg}(proof, (\text{TPM_ST_CREATION} \mathbin{||} name \mathbin{||} \mathbf{H}_{nameAlg}(\text{TPMS_CREATION_DATA}))) \quad (4)$$

where

$\mathbf{HMAC}_{contextAlg}()$	an HMAC using the context integrity hash algorithm
$proof$	a TPM secret value associated with the hierarchy associated with name
TPM_ST_CREATION	a value used to ensure that the ticket is properly used
$name$	the Name of the object to which the creation data is to be associated
$\mathbf{H}_{nameAlg}()$	hash using the *nameAlg* of the created object
TPMS_CREATION_DATA	the creation data structure associated with name

 Copyright © TCG 2006-2020

Table 96 — Definition of TPMT_TK_CREATION Structure

Parameter	Type	Description
tag {TPM_ST_CREATION}	TPM_ST	ticket structure tag
#TPM_RC_TAG		error returned when *tag* is not TPM_ST_CREATION
hierarchy	TPMI_RH_HIERARCHY+	the hierarchy containing *name*
digest	TPM2B_DIGEST	This shall be the HMAC produced using a proof value of *hierarchy*.

EXAMPLE A NULL Creation Ticket is the tuple <TPM_ST_CREATION, TPM_RH_NULL, 0x0000>.

10.7.4 TPMT_TK_VERIFIED

This ticket is produced by TPM2_VerifySignature(). This formulation is used for multiple ticket uses. The ticket provides evidence that the TPM has validated that a digest was signed by a key with the Name of keyName. The ticket is computed by

$$\mathbf{HMAC}_{contextAlg}(proof, (\text{TPM_ST_VERIFIED} \mathbin{||} digest \mathbin{||} keyName)) \tag{5}$$

where

$\mathbf{HMAC}_{contextAlg}()$	an HMAC using the context integrity hash
$proof$	a TPM secret value associated with the hierarchy associated with *keyName*
TPM_ST_VERIFIED	a value used to ensure that the ticket is properly used
$digest$	the signed digest
$keyName$	Name of the key that signed digest

Table 97 — Definition of TPMT_TK_VERIFIED Structure

Parameter	Type	Description
tag {TPM_ST_VERIFIED}	TPM_ST	ticket structure tag
#TPM_RC_TAG		error returned when *tag* is not TPM_ST_VERIFIED
hierarchy	TPMI_RH_HIERARCHY+	the hierarchy containing *keyName*
digest	TPM2B_DIGEST	This shall be the HMAC produced using a proof value of *hierarchy*.

EXAMPLE A NULL Verified Ticket is the tuple <TPM_ST_VERIFIED, TPM_RH_NULL, 0x0000>.

Level 00 Revision 01.59 **Copyright © TCG 2006-2020** November 8, 2019

10.7.5 TPMT_TK_AUTH

This ticket is produced by TPM2_PolicySigned() and TPM2_PolicySecret() when the authorization has an expiration time. If *nonceTPM* was provided in the policy command, t*he* ticket is computed by

$$\mathbf{HMAC}_{contextAlg}(proof, (\text{TPM_ST_AUTH_xxx} \mid\mid cpHash \mid\mid policyRef \mid\mid authName \mid\mid timeout \mid\mid [timeEpoch] \mid\mid [resetCount])) \quad (6)$$

where

$\mathbf{HMAC}_{contextAlg}()$	an HMAC using the context integrity hash
$proof$	a TPM secret value associated with the hierarchy of the object associated with *authName*
TPM_ST_AUTH_xxx	either TPM_ST_AUTH_SIGNED or TPM_ST_AUTH_SECRET; used to ensure that the ticket is properly used
$cpHash$	optional hash of the authorized command
$policyRef$	optional reference to a policy value
$authName$	Name of the object that signed the authorization
$timeout$	implementation-specific value indicating when the authorization expires
$timeEpoch$	implementation-specific representation of the $timeEpoch$ at the time the ticket was created

NOTE 1 Not included if *timeout* is zero.

$resetCount$	implementation-specific representation of the TPM's *totalResetCount*

NOTE 2 Not included it *timeout* is zero or if *nonceTPM* was include in the authorization.

Table 98 — Definition of TPMT_TK_AUTH Structure

Parameter	Type	Description
tag {TPM_ST_AUTH_SIGNED, TPM_ST_AUTH_SECRET}	TPM_ST	ticket structure tag
#TPM_RC_TAG		error returned when *tag* is not TPM_ST_AUTH
hierarchy	TPMI_RH_HIERARCHY+	the hierarchy of the object used to produce the ticket
digest	TPM2B_DIGEST	This shall be the HMAC produced using a proof value of *hierarchy*.

EXAMPLE A NULL Auth Ticket is the tuple <TPM_ST_AUTH_SIGNED, TPM_RH_NULL, 0x0000> or the tuple <TPM_ST_AUTH_SIGNED, TPM_RH_NULL, 0x0000>

 Copyright © TCG 2006-2020

10.7.6 TPMT_TK_HASHCHECK

This ticket is produced by TPM2_SequenceComplete() or TPM2_Hash() when the message that was digested did not start with TPM_GENERATED_VALUE. The ticket is computed by

$$\mathbf{HMAC}_{contexAlg}(proof, (\text{TPM_ST_HASHCHECK} \mid\mid digest)) \quad (7)$$

where

$\mathbf{HMAC}_{contexAlg}\,()$	an HMAC using the context integrity hash
$proof$	a TPM secret value associated with the hierarchy indicated by the command
TPM_ST_HASHCHECK	a value used to ensure that the ticket is properly used
$digest$	the digest of the data

Table 99 — Definition of TPMT_TK_HASHCHECK Structure

Parameter	Type	Description
tag {TPM_ST_HASHCHECK}	TPM_ST	ticket structure tag
#TPM_RC_TAG		error returned when is not TPM_ST_HASHCHECK
hierarchy	TPMI_RH_HIERARCHY+	the hierarchy
digest	TPM2B_DIGEST	This shall be the HMAC produced using a proof value of *hierarchy*.

10.8 Property Structures

10.8.1 TPMS_ALG_PROPERTY

This structure is used to report the properties of an algorithm identifier. It is returned in response to a TPM2_GetCapability() with *capability* = TPM_CAP_ALG.

Table 100 — Definition of TPMS_ALG_PROPERTY Structure <OUT>

Parameter	Type	Description
alg	TPM_ALG_ID	an algorithm identifier
algProperties	TPMA_ALGORITHM	the attributes of the algorithm

10.8.2 TPMS_TAGGED_PROPERTY

This structure is used to report the properties that are UINT32 values. It is returned in response to a TPM2_GetCapability().

Table 101 — Definition of TPMS_TAGGED_PROPERTY Structure <OUT>

Parameter	Type	Description
property	TPM_PT	a property identifier
value	UINT32	the value of the property

Level 00 Revision 01.59 **Copyright © TCG 2006-2020** November 8, 2019

10.8.3 TPMS_TAGGED_PCR_SELECT

This structure is used in TPM2_GetCapability() to return the attributes of the PCR.

Table 102 — Definition of TPMS_TAGGED_PCR_SELECT Structure <OUT>

Parameter	Type	Description
tag	TPM_PT_PCR	the property identifier
sizeofSelect {PCR_SELECT_MIN:}	UINT8	the size in octets of the *pcrSelect* array
pcrSelect [sizeofSelect] {:PCR_SELECT_MAX}	BYTE	the bit map of PCR with the identified property

10.8.4 TPMS_TAGGED_POLICY

This structure is used in TPM2_GetCapability() to return the policy associated with a permanent handle.

Table 103 — Definition of TPMS_TAGGED_POLICY Structure <OUT>

Parameter	Type	Description
handle	TPM_HANDLE	a permanent handle
policyHash	TPMT_HA	the policy algorithm and hash

10.8.5 TPMS_ACT_DATA

This structure is used in TPM2_GetCapability() to return the ACT data.

Table 104 — Definition of TPMS_ACT_DATA Structure <OUT>

Parameter	Type	Description
handle	TPM_HANDLE	a permanent handle
timeout	UINT32	the current timeout of the ACT
attributes	TPMA_ACT	the state of the ACT

 Copyright © TCG 2006-2020

10.9 Lists

10.9.1 TPML_CC

A list of command codes may be input to the TPM or returned by the TPM depending on the command.

Table 105 — Definition of TPML_CC Structure

Parameter	Type	Description
count	UINT32	number of commands in the *commandCode* list; may be 0
commandCodes[count]{:MAX_CAP_CC}	TPM_CC	a list of command codes The maximum only applies to a command code list in a command. The response size is limited only by the size of the parameter buffer.
#TPM_RC_SIZE		response code when count is greater than the maximum allowed list size

10.9.2 TPML_CCA

This list is only used in TPM2_GetCapability(capability = TPM_CAP_COMMANDS).

The values in the list are returned in TPMA_CC->*commandIndex* order (see Table 37) with vendor-specific commands returned after other commands. Because of the other attributes, the commands may not be returned in strict numerical order.

Table 106 — Definition of TPML_CCA Structure <OUT>

Parameter	Type	Description
count	UINT32	number of values in the *commandAttributes* list; may be 0
commandAttributes[count]{:MAX_CAP_CC}	TPMA_CC	a list of command codes attributes

10.9.3 TPML_ALG

This list is returned by TPM2_IncrementalSelfTest().

Table 107 — Definition of TPML_ALG Structure

Parameter	Type	Description
count	UINT32	number of algorithms in the *algorithms* list; may be 0
algorithms[count]{:MAX_ALG_LIST_SIZE}	TPM_ALG_ID	a list of algorithm IDs The maximum only applies to an algorithm list in a command. The response size is limited only by the size of the parameter buffer.
#TPM_RC_SIZE		response code when *count* is greater than the maximum allowed list size

 Copyright © TCG 2006-2020

10.9.4 TPML_HANDLE

This structure is used when the TPM returns a list of loaded handles when the *capability* in TPM2_GetCapability() is TPM_CAP_HANDLE.

NOTE 1 MAX_CAP_HANDLES = (MAX_CAP_DATA / sizeof(TPM_HANDLE))

NOTE 2 This list is not used as input to the TPM.

Table 108 — Definition of TPML_HANDLE Structure <OUT>

Name	Type	Description
count	UINT32	the number of handles in the list may have a value of 0
handle[count]{: MAX_CAP_HANDLES}	TPM_HANDLE	an array of handles
#TPM_RC_SIZE		response code when *count* is greater than the maximum allowed list size

10.9.5 TPML_DIGEST

This list is used to convey a list of digest values. This type is used in TPM2_PolicyOR() and in TPM2_PCR_Read().

Table 109 — Definition of TPML_DIGEST Structure

Parameter	Type	Description
count {2:}	UINT32	number of digests in the list, minimum is two for TPM2_PolicyOR().
digests[count]{:8}	TPM2B_DIGEST	a list of digests For TPM2_PolicyOR(), all digests will have been computed using the digest of the policy session. For TPM2_PCR_Read(), each digest will be the size of the digest for the bank containing the PCR.
#TPM_RC_SIZE		response code when *count* is not at least two or is greater than eight

 Copyright © TCG 2006-2020

10.9.6 TPML_DIGEST_VALUES

This list is used to convey a list of digest values. This type is returned by TPM2_PCR_Event() and TPM2_EventSequenceComplete() and is an input for TPM2_PCR_Extend().

NOTE 1 This construct limits the number of hashes in the list to the number of digests implemented in the TPM rather than the number of PCR banks. This allows extra values to appear in a call to TPM2_PCR_Extend().

NOTE 2 The digest for an unimplemented hash algorithm may not be in a list because the TPM may not recognize the algorithm as being a hash and it may not know the digest size.

Table 110 — Definition of TPML_DIGEST_VALUES Structure

Parameter	Type	Description
count	UINT32	number of digests in the list
digests[count]{:HASH_COUNT}	TPMT_HA	a list of tagged digests
#TPM_RC_SIZE		response code when *count* is greater than the possible number of banks

10.9.7 TPML_PCR_SELECTION

This list is used to indicate the PCR that are included in a selection when more than one PCR value may be selected.

This structure is an input parameter to TPM2_PolicyPCR() to indicate the PCR that will be included in the digest of PCR for the authorization. The structure is used in TPM2_PCR_Read() command to indicate the PCR values to be returned and in the response to indicate which PCR are included in the list of returned digests. The structure is an output parameter from TPM2_Create() and indicates the PCR used in the digest of the PCR state when the object was created. The structure is also contained in the attestation structure of TPM2_Quote().

When this structure is used to select PCR to be included in a digest, the selected PCR are concatenated to create a “message” containing all of the PCR, and then the message is hashed using the context-specific hash algorithm.

Table 111 — Definition of TPML_PCR_SELECTION Structure

Parameter	Type	Description
count	UINT32	number of selection structures A value of zero is allowed.
pcrSelections[count]{:HASH_COUNT}	TPMS_PCR_SELECTION	list of selections
#TPM_RC_SIZE		response code when *count* is greater than the possible number of banks

10.9.8 TPML_ALG_PROPERTY

This list is used to report on a list of algorithm attributes. It is returned in a TPM2_GetCapability().

NOTE MAX_CAP_ALGS = MAX_CAP_DATA / sizeof(TPMS_ALG_PROPERTY).

 Copyright © TCG 2006-2020

Table 112 — Definition of TPML_ALG_PROPERTY Structure <OUT>

Parameter	Type	Description
count	UINT32	number of algorithm properties structures A value of zero is allowed.
algProperties[count]{:MAX_CAP_ALGS}	TPMS_ALG_PROPERTY	list of properties

10.9.9 TPML_TAGGED_TPM_PROPERTY

This list is used to report on a list of properties that are TPMS_TAGGED_PROPERTY values. It is returned by a TPM2_GetCapability().

NOTE MAX_TPM_PROPERTIES = MAX_CAP_DATA / sizeof(TPMS_TAGGED_PROPERTY).

Table 113 — Definition of TPML_TAGGED_TPM_PROPERTY Structure <OUT>

Parameter	Type	Description
count	UINT32	number of properties A value of zero is allowed.
tpmProperty[count]{:MAX_TPM_PROPERTIES}	TPMS_TAGGED_PROPERTY	an array of tagged properties

10.9.10 TPML_TAGGED_PCR_PROPERTY

This list is used to report on a list of properties that are TPMS_PCR_SELECT values. It is returned by a TPM2_GetCapability().

NOTE MAX_PCR_PROPERTIES = MAX_CAP_DATA / sizeof(TPMS_TAGGED_PCR_SELECT).

Table 114 — Definition of TPML_TAGGED_PCR_PROPERTY Structure <OUT>

Parameter	Type	Description
count	UINT32	number of properties A value of zero is allowed.
pcrProperty[count]{:MAX_PCR_PROPERTIES}	TPMS_TAGGED_PCR_SELECT	a tagged PCR selection

10.9.11 TPML_ECC_CURVE

This list is used to report the ECC curve ID values supported by the TPM. It is returned by a TPM2_GetCapability().

NOTE MAX_ECC_CURVES = MAX_CAP_DATA / sizeof(TPM_ECC_CURVE).

Table 115 — Definition of {ECC} TPML_ECC_CURVE Structure <OUT>

Parameter	Type	Description
count	UINT32	number of curves A value of zero is allowed.
eccCurves[count]{:MAX_ECC_CURVES}	TPM_ECC_CURVE	array of ECC curve identifiers

 Copyright © TCG 2006-2020

10.9.12 TPML_TAGGED_POLICY

This list is used to report the authorization policy values for permanent handles. This is list may be generated by TPM2_GetCapabiltiy(). A permanent handle that cannot have a policy is not included in the list.

NOTE MAX_TAGGED_POLICIES = MAX_CAP_DATA / sizeof(TPMS_TAGGED_POLICY).

Table 116 — Definition of TPML_TAGGED_POLICY Structure <OUT>

Parameter	Type	Description
count	UINT32	number of tagged policies A value of zero is allowed.
policies[count]{:MAX_TAGGED_POLICIES}	TPMS_TAGGED_POLICY	array of tagged policies

10.9.13 TPML_ACT_DATA

This list is used to report the timeout and state for the ACT. This list may be generated by TPM2_GetCapabilty(). Only implemented ACT are present in the list

NOTE MAX_ACT_DATA = MAX_CAP_DATA / sizeof(TPMS_ACT_DATA).

Table 117 — Definition of TPML_ACT_DATA Structure <OUT>

Parameter	Type	Description
count	UINT32	number of ACT instances A value of zero is allowed.
actData[count]{:MAX_ACT_DATA}	TPMS_ACT_DATA	array of ACT data

10.10 Capabilities Structures

It is required that each parameter in this union be a list (TPML).

The number of returned elements in each list is determined by the size of each list element and the maximum size set by the vendor as the capability buffer (MAX_CAP_BUFFER in TPM_PT_MAX_CAP_BUFFER). The maximum number of bytes in a list is:

```
MAX_CAP_DATA = (MAX_CAP_BUFFER – sizeof(TPM_CAP) – sizeof(UINT32)
```
(8)

The maximum number of entries is then the number of complete list elements that will fit in MAX_CAP_DATA.

EXAMPLE For a 1024-octet MAX_CAP_BUFFER a response containing a TPML_HANDLE could have (1024 - 4 – 4) / 4 = 254 handles.

Level 00 Revision 01.59 **Copyright © TCG 2006-2020** November 8, 2019

10.10.1 TPMU_CAPABILITIES

Table 118 — Definition of TPMU_CAPABILITIES Union <OUT>

Parameter	Type	Selector	Description
algorithms	TPML_ALG_PROPERTY	TPM_CAP_ALGS	
handles	TPML_HANDLE	TPM_CAP_HANDLES	
command	TPML_CCA	TPM_CAP_COMMANDS	
ppCommands	TPML_CC	TPM_CAP_PP_COMMANDS	
auditCommands	TPML_CC	TPM_CAP_AUDIT_COMMANDS	
assignedPCR	TPML_PCR_SELECTION	TPM_CAP_PCRS	
tpmProperties	TPML_TAGGED_TPM_PROPERTY	TPM_CAP_TPM_PROPERTIES	
pcrProperties	TPML_TAGGED_PCR_PROPERTY	TPM_CAP_PCR_PROPERTIES	
eccCurves	TPML_ECC_CURVE	TPM_CAP_ECC_CURVES	TPM_ALG_ECC
authPolicies	TPML_TAGGED_POLICY	TPM_CAP_AUTH_POLICIES	
actData	TPML_ACT_DATA	TPM_CAP_ACT	

10.10.2 TPMS_CAPABILITY_DATA

This data area is returned in response to a TPM2_GetCapability().

Table 119 — Definition of TPMS_CAPABILITY_DATA Structure <OUT>

Parameter	Type	Description
capability	TPM_CAP	the capability
[capability]data	TPMU_CAPABILITIES	the capability data

 Copyright © TCG 2006-2020

10.11 Clock/Counter Structures

10.11.1 TPMS_CLOCK_INFO

This structure is used in each of the attestation commands.

Table 120 — Definition of TPMS_CLOCK_INFO Structure

Parameter	Type	Description
clock	UINT64	time value in milliseconds that advances while the TPM is powered NOTE The interpretation of the time-origin (*clock*=0) is out of the scope of this specification, although Coordinated Universal Time (UTC) is expected to be a common convention. This structure element is used to report on the TPM's Clock value. This value is reset to zero when the Storage Primary Seed is changed (TPM2_Clear()). This value may be advanced by TPM2_ClockSet().
resetCount	UINT32	number of occurrences of TPM Reset since the last TPM2_Clear()
restartCount	UINT32	number of times that TPM2_Shutdown() or _TPM_Hash_Start have occurred since the last TPM Reset or TPM2_Clear().
safe	TPMI_YES_NO	no value of *Clock* greater than the current value of *Clock* has been previously reported by the TPM. Set to YES on TPM2_Clear().

 Copyright © TCG 2006-2020

10.11.2 *Clock*

Clock is a monotonically increasing counter that advances whenever power is applied to the TPM. The value of *Clock* may be set forward with TPM2_ClockSet() if Owner Authorization or Platform Authorization is provided. The value of *Clock* is incremented each millisecond.

TPM2_Clear() will set *Clock* to zero.

Clock will be non-volatile but may have a volatile component that is updated every millisecond with the non-volatile component updated at a lower rate. The reference for the millisecond timer is the TPM oscillator. If the implementation uses a volatile component, the non-volatile component shall be updated no less frequently than every 2^{22} milliseconds (~69.9 minutes). The update rate of the non-volatile portion of *Clock* shall be reported by a TPM2_GetCapability() with *capability* = TPM_CAP_TPM_PROPERTIES and *property* = TPM_PT_CLOCK_UPDATE.

10.11.3 *ResetCount*

This counter shall increment on each TPM Reset. This counter shall be reset to zero by TPM2_Clear().

10.11.4 *RestartCount*

This counter shall increment by one for each TPM Restart or TPM Resume. The *restartCount* shall be reset to zero on a TPM Reset or TPM2_Clear().

10.11.5 *Safe*

This parameter is set to YES when the value reported in *Clock* is guaranteed to be greater than any previous value for the current Owner. It is set to NO when the value of *Clock* may have been reported in a previous attestation or access.

EXAMPLE 1 If *Safe* was NO at TPM2_Shutdown() and *Clock* does not update unless a command is received, *Safe* will be NO if a TPM2_Startup() was preceded by TPM2_Shutdown() with no intervening commands. If *Clock* updates independent of commands, the non-volatile bits of *Clock* can be updated, so *Safe* can be YES at TPM2_Startup().

EXAMPLE 2 This parameter will be YES after the non-volatile bits of *Clock* have been updated at the end of an update interval.

If a TPM implementation does not implement *Clock*, *Safe* shall always be NO and TPMS_CLOCK_INFO.*clock* shall always be zero.

This parameter will be set to YES by TPM2_Clear().

10.11.6 TPMS_TIME_INFO

This structure is used in, e.g., the TPM2_GetTime() attestation and TPM2_ReadClock().

The *Time* value reported in this structure is reset whenever power to the *Time* circuit is reestablished. If required, an implementation may reset the value of *Time* any time before the TPM returns after TPM2_Startup(). The value of *Time* shall increment continuously while power is applied to the TPM.

Table 121 — Definition of TPMS_TIME_INFO Structure

Parameter	Type	Description
time	UINT64	time in milliseconds since the *TIme* circuit was last reset This structure element is used to report on the TPM's *Time* value.
clockInfo	TPMS_CLOCK_INFO	a structure containing the clock information

 Copyright © TCG 2006-2020

10.12 TPM Attestation Structures

10.12.1 Introduction

Clause 10.12 describes the structures that are used when a TPM creates a structure to be signed. The signing structures follow a standard format TPM2B_ATTEST with case-specific information embedded.

10.12.2 TPMS_TIME_ATTEST_INFO

This structure is used when the TPM performs TPM2_GetTime.

Table 122 — Definition of TPMS_TIME_ATTEST_INFO Structure <OUT>

Parameter	Type	Description
time	TPMS_TIME_INFO	the *Time*, *Clock*, *resetCount*, *restartCount*, and *Safe* indicator
firmwareVersion	UINT64	a TPM vendor-specific value indicating the version number of the firmware

10.12.3 TPMS_CERTIFY_INFO

This is the attested data for TPM2_Certify().

Table 123 — Definition of TPMS_CERTIFY_INFO Structure <OUT>

Parameter	Type	Description
name	TPM2B_NAME	Name of the certified object
qualifiedName	TPM2B_NAME	Qualified Name of the certified object

10.12.4 TPMS_QUOTE_INFO

This is the *attested* data for TPM2_Quote().

Table 124 — Definition of TPMS_QUOTE_INFO Structure <OUT>

Parameter	Type	Description
pcrSelect	TPML_PCR_SELECTION	information on *algID*, PCR selected and digest
pcrDigest	TPM2B_DIGEST	digest of the selected PCR using the hash of the signing key

 Copyright © TCG 2006-2020

10.12.5 TPMS_COMMAND_AUDIT_INFO

This is the *attested* data for TPM2_GetCommandAuditDigest().

Table 125 — Definition of TPMS_COMMAND_AUDIT_INFO Structure <OUT>

Parameter	Type	Description
auditCounter	UINT64	the monotonic audit counter
digestAlg	TPM_ALG_ID	hash algorithm used for the command audit
auditDigest	TPM2B_DIGEST	the current value of the audit digest
commandDigest	TPM2B_DIGEST	digest of the command codes being audited using *digestAlg*

10.12.6 TPMS_SESSION_AUDIT_INFO

This is the *attested* data for TPM2_GetSessionAuditDigest().

Table 126 — Definition of TPMS_SESSION_AUDIT_INFO Structure <OUT>

Parameter	Type	Description
exclusiveSession	TPMI_YES_NO	current exclusive status of the session TRUE if all of the commands recorded in the *sessionDigest* were executed without any intervening TPM command that did not use this audit session
sessionDigest	TPM2B_DIGEST	the current value of the session audit digest

10.12.7 TPMS_CREATION_INFO

This is the *attested* data for TPM2_CertifyCreation().

Table 127 — Definition of TPMS_CREATION_INFO Structure <OUT>

Parameter	Type	Description
objectName	TPM2B_NAME	Name of the object
creationHash	TPM2B_DIGEST	creationHash

10.12.8 TPMS_NV_CERTIFY_INFO

This structure contains the Name and contents of the selected NV Index that is certified by TPM2_NV_Certify().

Table 128 — Definition of TPMS_NV_CERTIFY_INFO Structure <OUT>

Parameter	Type	Description
indexName	TPM2B_NAME	Name of the NV Index
offset	UINT16	the *offset* parameter of TPM2_NV_Certify()
nvContents	TPM2B_MAX_NV_BUFFER	contents of the NV Index

 Copyright © TCG 2006-2020

10.12.9 TPMS_NV_DIGEST_CERTIFY_INFO

This structure contains the Name and hash of the contents of the selected NV Index that is certified by TPM2_NV_Certify(). The data is hashed using hash of the signing scheme.

NOTE This structure was added in revision 01.53 to support alternate TPM2_NV_Certify() behavior.

Table 129 — Definition of TPMS_NV_DIGEST_CERTIFY_INFO Structure <OUT>

Parameter	Type	Description
indexName	TPM2B_NAME	Name of the NV Index
nvDigest	TPM2B_DIGEST	hash of the contents of the index

10.12.10 TPMI_ST_ATTEST

Table 130 — Definition of (TPM_ST) TPMI_ST_ATTEST Type <OUT>

Value	Description
TPM_ST_ATTEST_CERTIFY	generated by TPM2_Certify()
TPM_ST_ATTEST_QUOTE	generated by TPM2_Quote()
TPM_ST_ATTEST_SESSION_AUDIT	generated by TPM2_GetSessionAuditDigest()
TPM_ST_ATTEST_COMMAND_AUDIT	generated by TPM2_GetCommandAuditDigest()
TPM_ST_ATTEST_TIME	generated by TPM2_GetTime()
TPM_ST_ATTEST_CREATION	generated by TPM2_CertifyCreation()
TPM_ST_ATTEST_NV	generated by TPM2_NV_Certify()
TPM_ST_ATTEST_NV_DIGEST	generated by TPM2_NV_Certify()

10.12.11 TPMU_ATTEST

Table 131 — Definition of TPMU_ATTEST Union <OUT>

Parameter	Type	Selector
certify	TPMS_CERTIFY_INFO	TPM_ST_ATTEST_CERTIFY
creation	TPMS_CREATION_INFO	TPM_ST_ATTEST_CREATION
quote	TPMS_QUOTE_INFO	TPM_ST_ATTEST_QUOTE
commandAudit	TPMS_COMMAND_AUDIT_INFO	TPM_ST_ATTEST_COMMAND_AUDIT
sessionAudit	TPMS_SESSION_AUDIT_INFO	TPM_ST_ATTEST_SESSION_AUDIT
time	TPMS_TIME_ATTEST_INFO	TPM_ST_ATTEST_TIME
nv	TPMS_NV_CERTIFY_INFO	TPM_ST_ATTEST_NV
nvDigest	TPMS_NV_DIGEST_CERTIFY_INFO	TPM_ST_ATTEST_NV_DIGEST

Copyright © TCG 2006-2020

10.12.12 TPMS_ATTEST

This structure is used on each TPM-generated signed structure. The signature is over this structure.

When the structure is signed by a key in the Storage hierarchy, the values of *clockInfo.resetCount*, *clockInfo.restartCount*, and *firmwareVersion* are obfuscated with a per-key obfuscation value.

Table 132 — Definition of TPMS_ATTEST Structure <OUT>

Parameter	Type	Description
magic	TPM_GENERATED	the indication that this structure was created by a TPM (always TPM_GENERATED_VALUE)
type	TPMI_ST_ATTEST	type of the attestation structure
qualifiedSigner	TPM2B_NAME	Qualified Name of the signing key
extraData	TPM2B_DATA	external information supplied by caller NOTE A TPM2B_DATA structure provides room for a digest and a method indicator to indicate the components of the digest. The definition of this method indicator is outside the scope of this specification.
clockInfo	TPMS_CLOCK_INFO	Clock, resetCount, restartCount, and Safe
firmwareVersion	UINT64	TPM-vendor-specific value identifying the version number of the firmware
[type]attested	TPMU_ATTEST	the type-specific attestation information

 Copyright © TCG 2006-2020

10.12.13 TPM2B_ATTEST

This sized buffer to contain the signed structure. The *attestationData* is the signed portion of the structure. The *size* parameter is not signed.

Table 133 — Definition of TPM2B_ATTEST Structure <OUT>

Parameter	Type	Description
size	UINT16	size of the *attestationData* structure
attestationData[size]{:sizeof(TPMS_ATTEST)}	BYTE	the signed structure

10.13 Authorization Structures

10.13.1 Introduction

The structures in 10.13 are used for all authorizations. One or more of these structures will be present in a command or response that has a tag of TPM_ST_SESSIONS.

10.13.2 TPMS_AUTH_COMMAND

This is the format used for each of the authorizations in the session area of a command.

Table 134 — Definition of TPMS_AUTH_COMMAND Structure <IN>

Parameter	Type	Description
sessionHandle	TPMI_SH_AUTH_SESSION+	the session handle
nonce	TPM2B_NONCE	the session nonce, may be the Empty Buffer
sessionAttributes	TPMA_SESSION	the session attributes
hmac	TPM2B_AUTH	either an HMAC, a password, or an EmptyAuth

10.13.3 TPMS_AUTH_RESPONSE

This is the format for each of the authorizations in the session area of the response. If the TPM returns TPM_RC_SUCCESS, then the session area of the response contains the same number of authorizations as the command and the authorizations are in the same order.

Table 135 — Definition of TPMS_AUTH_RESPONSE Structure <OUT>

Parameter	Type	Description
nonce	TPM2B_NONCE	the session nonce, may be the Empty Buffer
sessionAttributes	TPMA_SESSION	the session attributes
hmac	TPM2B_AUTH	either an HMAC or an EmptyAuth

 Copyright © TCG 2006-2020

11 Algorithm Parameters and Structures

11.1 Symmetric

11.1.1 Introduction

Clause 11.1 defines the parameters and structures for describing symmetric algorithms.

11.1.2 TPMI_!ALG.S_KEY_BITS

This interface type defines the supported key sizes for a symmetric algorithm. This type is used to allow the unmarshaling routine to generate the proper validation code for the supported key sizes. An implementation that supports different key sizes would have a different set of selections.

Each implemented algorithm would have a value for the implemented key sizes for that implemented algorithm. That value would have a name in the form !ALG_KEY_SIZES_BITS where "!ALG" would represent the characteristic name of the algorithm (such as "AES).

NOTE 1 Key size is expressed in bits.

Table 136 — Definition of {!ALG.S} (TPM_KEY_BITS) TPMI_!ALG.S_KEY_BITS Type

Parameter	Description
$!ALG.S_KEY_SIZES_BITS	number of bits in the key
#TPM_RC_VALUE	error when key size is not supported

11.1.3 TPMU_SYM_KEY_BITS

This union is used to collect the symmetric encryption key sizes.

The *xor* entry is a hash algorithms selector and not a key size in bits. This overload is used in order to avoid an additional level of indirection with another union and another set of selectors.

The *xor* entry is only selected in a TPMT_SYM_DEF, which is used to select the parameter encryption value.

Table 137 — Definition of TPMU_SYM_KEY_BITS Union

Parameter	Type	Selector	Description
!ALG.S	TPMI_!ALG.S_KEY_BITS	TPM_ALG_!ALG.S	all symmetric algorithms
sym	TPM_KEY_BITS		this entry is used by the reference code to refer to the key bits field in a way that is independent of the symmetric algorithm
xor	TPMI_ALG_HASH	TPM_ALG_XOR	overload for using *xor* NOTE TPM_ALG_NULL is not allowed
null		TPM_ALG_NULL	

 Copyright © TCG 2006-2020

11.1.4 TPMU_SYM_MODE

This is the union of all modes for all symmetric algorithms.

NOTE This union definition allows the mode value in a TPMT_SYM_DEF to be empty when the selector is TPM_ALG_XOR because the XOR algorithm does not have a mode.

Table 138 — Definition of TPMU_SYM_MODE Union

Parameter	Type	Selector	Description
!ALG.S	TPMI_ALG_SYM_MODE+	TPM_ALG_!ALG.S	
sym	TPMI_ALG_SYM_MODE+		this entry is used by the reference code to refer to the mode field in a way that is independent of the symmetric algorithm
xor		TPM_ALG_XOR	no mode selector
null		TPM_ALG_NULL	no mode selector

11.1.5 TPMU_SYM_DETAILS

This union allows additional parameters to be added for a symmetric cipher. Currently, no additional parameters are required for any of the symmetric algorithms.

NOTE The “x” character in the table title will suppress generation of this type as the parser is not, at this time, able to generate the proper values (a union of all empty data types). When an algorithm is added that requires additional parameterization, the Type column will contain a value and the “x” may be removed.

Table 139 —xDefinition of TPMU_SYM_DETAILS Union

Parameter	Type	Selector	Description
!ALG.S		TPM_ALG_!ALG	
sym			this entry is used by the reference code to refer to the details field in a way that is independent of the symmetric algorithm
xor		TPM_ALG_XOR	
null		TPM_ALG_NULL	

 Copyright © TCG 2006-2020

11.1.6 TPMT_SYM_DEF

The TPMT_SYM_DEF structure is used to select an algorithm to be used for parameter encryption in those cases when different symmetric algorithms may be selected.

Table 140 — Definition of TPMT_SYM_DEF Structure

Parameter	Type	Description
algorithm	+TPMI_ALG_SYM	indicates a symmetric algorithm
[algorithm]keyBits	TPMU_SYM_KEY_BITS	a supported key size
[algorithm]mode	TPMU_SYM_MODE	the mode for the key
//[algorithm]details	TPMU_SYM_DETAILS	contains additional algorithm details NOTE This is commented out at this time as the parser may not produce the proper code for a union if none of the selectors produces any data.

11.1.7 TPMT_SYM_DEF_OBJECT

This structure is used when different symmetric block cipher (not XOR) algorithms may be selected. If the Object can be an ordinary parent (not a derivation parent), this must be the first field in the Object's parameter (see 12.2.3.7) field.

Table 141 — Definition of TPMT_SYM_DEF_OBJECT Structure

Parameter	Type	Description
algorithm	+TPMI_ALG_SYM_OBJECT	selects a symmetric block cipher When used in the parameter area of a parent object, this shall be a supported block cipher and not TPM_ALG_NULL
[algorithm]keyBits	TPMU_SYM_KEY_BITS	the key size
[algorithm]mode	TPMU_SYM_MODE	default mode When used in the parameter area of a parent object, this shall be TPM_ALG_CFB.
//[algorithm]details	TPMU_SYM_DETAILS	contains the additional algorithm details, if any NOTE This is commented out at this time as the parser may not produce the proper code for a union if none of the selectors produces any data.

11.1.8 TPM2B_SYM_KEY

This structure is used to hold a symmetric key in the sensitive area of an asymmetric object.

The number of bits in the key is in *keyBits* in the public area. When *keyBits* is not an even multiple of 8 bits, the unused bits of *buffer* will be the most significant bits of *buffer*[0] and *size* will be rounded up to the number of octets required to hold all bits of the key.

NOTE MAX_SYM_KEY_BYTES will be the larger of the largest symmetric key supported by the TPM and the largest digest produced by any hashing algorithm implemented on the TPM.

 Copyright © TCG 2006-2020

Table 142 — Definition of TPM2B_SYM_KEY Structure

Parameter	Type	Description
size	UINT16	size, in octets, of the buffer containing the key; may be zero
buffer [size] {:MAX_SYM_KEY_BYTES}	BYTE	the key

11.1.9 TPMS_SYMCIPHER_PARMS

This structure contains the parameters for a symmetric block cipher object.

Table 143 — Definition of TPMS_SYMCIPHER_PARMS Structure

Parameter	Type	Description
sym	TPMT_SYM_DEF_OBJECT	a symmetric block cipher

11.1.10 TPM2B_LABEL

This buffer holds a *label* or *context* value. For interoperability and backwards compatibility, LABEL_MAX_BUFFER is the minimum of the largest digest on the device and the largest ECC parameter (MAX_ECC_KEY_BYTES) but no more than 32 bytes.

All implementations are required to support at least one hash algorithm that produces a digest of 32 bytes or larger; and any implementation that supports ECC is required to support at least one curve with a key size of 32-bytes or larger.

NOTE Although the maximum size allowed for a *label* or *context* is 32 bytes, the object data structure needs to be sized to allow a 32-byte value.

Table 144 — Definition of TPM2B_LABEL Structure

Parameter	Type	Description
size	UINT16	
buffer[size]{:LABEL_MAX_BUFFER}	BYTE	symmetric data for a created object or the *label* and *context* for a derived object

11.1.11 TPMS_DERIVE

This structure contains the *label* and *context* fields for a derived object. These values are used in the derivation KDF. The values in the *unique* field of *inPublic* area template take precedence over the values in the *inSensitive* parameter.

Table 145 — Definition of TPMS_DERIVE Structure

Parameter	Type	Description
label	TPM2B_LABEL	
context	TPM2B_LABEL	

 Copyright © TCG 2006-2020

11.1.12 TPM2B_DERIVE

Table 146 — Definition of TPM2B_DERIVE Structure

Parameter	Type	Description
size	UINT16	
buffer[size]{: sizeof(TPMS_DERIVE)}	BYTE	symmetric data for a created object or the *label* and *context* for a derived object

11.1.13 TPMU_SENSITIVE_CREATE

This structure allows a TPM2B_SENSITIVE_CREATE structure to carry either a TPM2B_SENSITVE_DATA or a TPM2B_DERIVE structure. The contents of the union are determined by context. When an object is being derived, the derivation values are present.

For interoperability, MAX_SYM_DATA should be 128.

NOTE No marshaling code is automatically generated for this union as it has no selectors that would allow the code to know the context and which member to unmarshal.

Table 147 — Definition of TPMU_SENSITIVE_CREATE Union <>

Parameter	Type	Selector	Description
create[MAX_SYM_DATA]	BYTE		sensitive data for a created symmetric Object
derive	TPMS_DERIVE		*label* and *context* for a derived Object

11.1.14 TPM2B_SENSITIVE_DATA

This buffer wraps the TPMU_SENSITIVE_CREATE structure.

Table 148 — Definition of TPM2B_SENSITIVE_DATA Structure

Parameter	Type	Description
size	UINT16	
buffer[size]{: sizeof(TPMU_SENSITIVE_CREATE)}	BYTE	symmetric data for a created object or the *label* and *context* for a derived object

 Copyright © TCG 2006-2020

11.1.15 TPMS_SENSITIVE_CREATE

This structure defines the values to be placed in the sensitive area of a created object. This structure is only used within a TPM2B_SENSITIVE_CREATE structure.

NOTE When sent to the TPM or unsealed, data is usually encrypted using parameter encryption.

If *data.size* is not zero, and the object is not a *keyedHash*, *data.size* must match the size indicated in the *keySize* of *public.parameters*. If the object is a *keyedHash*, *data.size* may be any value up to the maximum allowed in a TPM2B_SENSITIVE_DATA.

For an asymmetric object, data shall be an Empty Buffer and *sensitiveDataOrigin* shall be SET.

Table 149 — Definition of TPMS_SENSITIVE_CREATE Structure <IN>

Parameter	Type	Description
userAuth	TPM2B_AUTH	the USER auth secret value
data	TPM2B_SENSITIVE_DATA	data to be sealed, a key, or derivation values

11.1.16 TPM2B_SENSITIVE_CREATE

This structure contains the sensitive creation data in a sized buffer. This structure is defined so that both the *userAuth* and *data* values of the TPMS_SENSITIVE_CREATE may be passed as a single parameter for parameter encryption purposes.

Table 150 — Definition of TPM2B_SENSITIVE_CREATE Structure <IN, S>

Parameter	Type	Description
size=	UINT16	size of *sensitive* in octets (may not be zero) NOTE The *userAuth* and data parameters in this buffer may both be zero length but the minimum size of this parameter will be the sum of the size fields of the two parameters of the TPMS_SENSITIVE_CREATE.
sensitive	TPMS_SENSITIVE_CREATE	data to be sealed or a symmetric key value.

11.1.17 TPMS_SCHEME_HASH

This structure is the scheme data for schemes that only require a hash to complete their definition.

Table 151 — Definition of TPMS_SCHEME_HASH Structure

Parameter	Type	Description
hashAlg	TPMI_ALG_HASH	the hash algorithm used to digest the message

 Copyright © TCG 2006-2020

11.1.18 TPMS_SCHEME_ECDAA

This definition is for split signing schemes that require a commit count.

Table 152 — Definition of {ECC} TPMS_SCHEME_ECDAA Structure

Parameter	Type	Description
hashAlg	TPMI_ALG_HASH	the hash algorithm used to digest the message
count	UINT16	the counter value that is used between TPM2_Commit() and the sign operation

11.1.19 TPMI_ALG_KEYEDHASH_SCHEME

This is the list of values that may appear in a *keyedHash* as the *scheme* parameter.

Table 153 — Definition of (TPM_ALG_ID) TPMI_ALG_KEYEDHASH_SCHEME Type

Values	Comments
TPM_ALG_HMAC	the "signing" scheme
TPM_ALG_XOR	the "obfuscation" scheme
+TPM_ALG_NULL	
#TPM_RC_VALUE	

11.1.20 HMAC_SIG_SCHEME

Table 154 — Definition of Types for HMAC_SIG_SCHEME

Type	Name	Description
TPMS_SCHEME_HASH	TPMS_SCHEME_HMAC	

11.1.21 TPMS_SCHEME_XOR

This structure is for the XOR encryption scheme.

NOTE Prior to revision 01.47, the TPM_ALG_NULL hash algorithm was permitted. This produced a zero length key. The TPM_ALG_NULL *hashAlg* now returns TPM_RC_HASH.

Table 155 — Definition of TPMS_SCHEME_XOR Structure

Parameter	Type	Description
hashAlg	TPMI_ALG_HASH	the hash algorithm used to digest the message
kdf	TPMI_ALG_KDF+	the key derivation function

 Copyright © TCG 2006-2020

11.1.22 TPMU_SCHEME_KEYEDHASH

Table 156 — Definition of TPMU_SCHEME_KEYEDHASH Union <IN/OUT >

Parameter	Type	Selector	Description
hmac	TPMS_SCHEME_HMAC	TPM_ALG_HMAC	the "signing" scheme
xor	TPMS_SCHEME_XOR	TPM_ALG_XOR	the "obfuscation" scheme
null		TPM_ALG_NULL	

11.1.23 TPMT_KEYEDHASH_SCHEME

This structure is used for a hash signing object.

Table 157 — Definition of TPMT_KEYEDHASH_SCHEME Structure

Parameter	Type	Description
scheme	+TPMI_ALG_KEYEDHASH_SCHEME	selects the scheme
[scheme]details	TPMU_SCHEME_KEYEDHASH	the scheme parameters

Level 00 Revision 01.59 **Copyright © TCG 2006-2020** November 8, 2019

11.2 Asymmetric

11.2.1 Signing Schemes

11.2.1.1 Introduction

These structures are used to define the method in which the signature is to be created. These schemes would appear in an object's public area and in commands where the signing scheme is variable.

Every scheme is required to indicate a hash that is used in digesting the message.

11.2.1.2 RSA Signature Schemes

These are the RSA schemes that only need a hash algorithm as a scheme parameter.

For the TPM_ALG_RSAPSS signing scheme, the same hash algorithm is used for digesting TPM-generated data (an attestation structure) and in the KDF used for the masking operation. The salt size is always the largest salt value that will fit into the available space.

Table 158 — Definition of {RSA} Types for RSA Signature Schemes

Type	Name	Description
TPMS_SCHEME_HASH	TPMS_SIG_SCHEME_!ALG.AX	

11.2.1.3 ECC Signature Schemes

Most of the ECC signature schemes only require a hash algorithm to complete the definition and can be typed as TPMS_SCHEME_HASH. Anonymous algorithms also require a count value so they are typed to be TPMS_SCHEME_ECDAA.

Table 159 — Definition of {ECC} Types for ECC Signature Schemes

Type	Name	Description
TPMS_SCHEME_HASH	TPMS_SIG_SCHEME_!ALG.AX	all asymmetric signing schemes
TPMS_SCHEME_ECDAA	TPMS_SIG_SCHEME_!ALG.AXN	schemes that need a hash and a count

 Copyright © TCG 2006-2020

11.2.1.4 TPMU_SIG_SCHEME

This is the union of all of the signature schemes.

NOTE The TPMS_SIG_SCHEME_!ALG is determined by Table 158 or Table 159 and will be either a TPMS_SCHEME_HASH or a TPMS_SCHEME_ECDAA.

Table 160 — Definition of TPMU_SIG_SCHEME Union <IN/OUT >

Parameter	Type	Selector	Description
!ALG.ax	TPMS_SIG_SCHEME_!ALG	TPM_ALG_!ALG	all signing schemes including anonymous schemes
hmac	TPMS_SCHEME_HMAC	TPM_ALG_HMAC	the HMAC scheme
any	TPMS_SCHEME_HASH		selector that allows access to digest for any signing scheme
null		TPM_ALG_NULL	no scheme or default

11.2.1.5 TPMT_SIG_SCHEME

Table 161 — Definition of TPMT_SIG_SCHEME Structure

Parameter	Type	Description
scheme	+TPMI_ALG_SIG_SCHEME	scheme selector
[scheme]details	TPMU_SIG_SCHEME	scheme parameters

11.2.2 Encryption Schemes

11.2.2.1 Introduction

These structures are used to indicate the algorithm used for the encrypting process. These schemes would appear in an object's public area.

NOTE With ECC, the only encryption is with a key exchange of a symmetric key or seed.

11.2.2.2 RSA Encryption Schemes

These are the RSA encryption schemes that only need a hash algorithm as a controlling parameter.

NOTE: These types do not appear in the reference code in the specification but are used in the unmarshaling code.

Table 162 — Definition of Types for {RSA} Encryption Schemes

Type	Name	Description
TPMS_SCHEME_HASH	TPMS_ENC_SCHEME_!ALG.AEH	schemes that only need a hash
TPMS_EMPTY	TPMS_ENC_SCHEME_!ALG.AE	schemes that need nothing

 Copyright © TCG 2006-2020

11.2.2.3 ECC Key Exchange Schemes

These are the ECC schemes that only need a hash algorithm as a controlling parameter.

NOTE: These types do not appear in the reference code in the specification but are used in the unmarshaling code.

Table 163 — Definition of Types for {ECC} ECC Key Exchange

Type	Name	Description
TPMS_SCHEME_HASH	TPMS_KEY_SCHEME_!ALG.AM	schemes that need a hash

11.2.3 Key Derivation Schemes

11.2.3.1 Introduction

These structures are used to define the key derivation for symmetric secret sharing using asymmetric methods. A secret sharing scheme is required in any asymmetric key with the *decrypt* attribute SET.

These schemes would appear in an object's public area and in commands where the secret sharing scheme is variable.

Each scheme includes a symmetric algorithm and a KDF selection.

The qualifying value for each of the KDF schemes is the hash algorithm.

NOTE: These types do not appear in the reference code in the specification but are used in the unmarshaling code.

Table 164 — Definition of Types for KDF Schemes

Type	Name	Description
TPMS_SCHEME_HASH	TPMS_SCHEME_!ALG.HM	hash-based key- or mask-generation functions

11.2.3.2 TPMU_KDF_SCHEME

Table 165 — Definition of TPMU_KDF_SCHEME Union <IN/OUT>

Parameter	Type	Selector	Description
!ALG.HM	TPMS_SCHEME_!ALG.HM	TPM_ALG_!ALG.HM	
null		TPM_ALG_NULL	

11.2.3.3 TPMT_KDF_SCHEME

Table 166 — Definition of TPMT_KDF_SCHEME Structure

Parameter	Type	Description
scheme	+TPMI_ALG_KDF	scheme selector
[scheme]details	TPMU_KDF_SCHEME	scheme parameters

 Copyright © TCG 2006-2020

11.2.3.4 TPMI_ALG_ASYM_SCHEME

List of all of the scheme types for any asymmetric algorithm.

NOTE 1 This is the selector value used to define TPMT_ASYM_SCHEME.

NOTE 2 Most tokens are exclusive in order to filter out SM2 and other multi-protocol algorithm identifiers. The inclusive token “ax” will include those algorithms.

Table 167 — Definition of (TPM_ALG_ID) TPMI_ALG_ASYM_SCHEME Type <IO>

Values	Comments
TPM_ALG_!ALG.am	key exchange methods
TPM_ALG_!ALG.ax	all signing including anonymous
TPM_ALG_!ALG.ae	encrypting schemes
+TPM_ALG_NULL	
#TPM_RC_VALUE	

11.2.3.5 TPMU_ASYM_SCHEME

This union of all asymmetric schemes is used in each of the asymmetric scheme structures. The actual scheme structure is defined by the interface type used for the selector (TPMI_ALG_ASYM_SCHEME).

EXAMPLE The TPMT_RSA_SCHEME structure uses the TPMU_ASYM_SCHEME union but the selector type is TPMI_ALG_RSA_SCHEME. This means that the only elements of the union that can be selected for the TPMT_RSA_SCHEME are those that are in TPMI_RSA_SCHEME.

Table 168 — Definition of TPMU_ASYM_SCHEME Union

Parameter	Type	Selector	Description
!ALG.am	TPMS_KEY_SCHEME_!ALG	TPM_ALG_!ALG	
!ALG.ax	TPMS_SIG_SCHEME_!ALG	TPM_ALG_!ALG	signing and anonymous signing
!ALG.ae	TPMS_ENC_SCHEME_!ALG	TPM_ALG_!ALG	schemes with no hash
anySig	TPMS_SCHEME_HASH		
null		TPM_ALG_NULL	no scheme or default This selects the NULL Signature.

11.2.3.6 TPMT_ASYM_SCHEME

This structure is defined to allow overlay of all of the schemes for any asymmetric object. This structure is not sent on the interface. It is defined so that common functions may operate on any similar scheme structure.

EXAMPLE Since many schemes have a hash algorithm as their defining parameter, a common function can use the digest selector to select the hash of the scheme without a need to cast or use a large switch statement.

 Copyright © TCG 2006-2020

Table 169 — Definition of TPMT_ASYM_SCHEME Structure <>

Parameter	Type	Description
scheme	+TPMI_ALG_ASYM_SCHEME	scheme selector
[scheme]details	TPMU_ASYM_SCHEME	scheme parameters

11.2.4 RSA

11.2.4.1 TPMI_ALG_RSA_SCHEME

The list of values that may appear in the scheme parameter of a TPMS_RSA_PARMS structure.

Table 170 — Definition of (TPM_ALG_ID) {RSA} TPMI_ALG_RSA_SCHEME Type

Values	Comments
TPM_ALG_!ALG.ae.ax	encrypting and signing algorithms
+TPM_ALG_NULL	
#TPM_RC_VALUE	

11.2.4.2 TPMT_RSA_SCHEME

Table 171 — Definition of {RSA} TPMT_RSA_SCHEME Structure

Parameter	Type	Description
scheme	+TPMI_ALG_RSA_SCHEME	scheme selector
[scheme]details	TPMU_ASYM_SCHEME	scheme parameters

11.2.4.3 TPMI_ALG_RSA_DECRYPT

The list of values that are allowed in a decryption scheme selection as used in TPM2_RSA_Encrypt() and TPM2_RSA_Decrypt().

Table 172 — Definition of (TPM_ALG_ID) {RSA} TPMI_ALG_RSA_DECRYPT Type

Values	Comments
TPM_ALG_!ALG.ae	all RSA encryption algorithms
+TPM_ALG_NULL	
#TPM_RC_VALUE	

11.2.4.4 TPMT_RSA_DECRYPT

Table 173 — Definition of {RSA} TPMT_RSA_DECRYPT Structure

Parameter	Type	Description
scheme	+TPMI_ALG_RSA_DECRYPT	scheme selector
[scheme]details	TPMU_ASYM_SCHEME	scheme parameters

 Copyright © TCG 2006-2020

11.2.4.5 TPM2B_PUBLIC_KEY_RSA

This sized buffer holds the largest RSA public key supported by the TPM.

NOTE The reference implementation only supports key sizes of 1,024 and 2,048 bits.

Table 174 — Definition of {RSA} TPM2B_PUBLIC_KEY_RSA Structure

Parameter	Type	Description
size	UINT16	size of the buffer The value of zero is only valid for create.
buffer[size] {: MAX_RSA_KEY_BYTES}	BYTE	Value

11.2.4.6 TPMI_RSA_KEY_BITS

This holds the value that is the maximum size allowed for an RSA key.

NOTE 1 An implementation is allowed to provide limited support for smaller RSA key sizes. That is, a TPM may be able to accept a smaller RSA key size in TPM2_LoadExternal() when only the public area is loaded but not accept that smaller key size in any command that loads both the public and private portions of an RSA key. This would allow the TPM to validate signatures using the smaller key but would prevent the TPM from using the smaller key size for any other purpose.

NOTE 2 The definition for RSA_KEY_SIZES_BITS used in the reference implementation is found in TPM 2.0 Part 4, Implementation.h

Table 175 — Definition of {RSA} (TPM_KEY_BITS) TPMI_RSA_KEY_BITS Type

Parameter	Description
$RSA_KEY_SIZES_BITS	the number of bits in the supported key
#TPM_RC_VALUE	error when key size is not supported

11.2.4.7 TPM2B_PRIVATE_KEY_RSA

This sized buffer holds the largest RSA prime number supported by the TPM.

NOTE 1 All primes are required to have exactly half the number of significant bits as the public modulus, and the square of each prime is required to have the same number of significant bits as the public modulus.

NOTE 2 RSA_PRIVATE_SIZE is a vendor specific value that can be (MAX_RSA_KEY_BYTES / 2) or ((MAX_RSA_KEY_BYTES * 5) ./ 2. The larger size would only apply to keys that have *fixedTPM* parents. The larger size was added in revision 01.53.

Table 176 — Definition of {RSA} TPM2B_PRIVATE_KEY_RSA Structure

Parameter	Type	Description
size	UINT16	
buffer[size]{:RSA_PRIVATE_SIZE }	BYTE	

 Copyright © TCG 2006-2020

11.2.5 ECC

11.2.5.1 TPM2B_ECC_PARAMETER

This sized buffer holds the largest ECC parameter (coordinate) supported by the TPM.

Table 177 — Definition of TPM2B_ECC_PARAMETER Structure

Parameter	Type	Description
size	UINT16	size of *buffer*
buffer[size] {:MAX_ECC_KEY_BYTES}	BYTE	the parameter data

11.2.5.2 TPMS_ECC_POINT

This structure holds two ECC coordinates that, together, make up an ECC point.

Table 178 — Definition of {ECC} TPMS_ECC_POINT Structure

Parameter	Type	Description
x	TPM2B_ECC_PARAMETER	X coordinate
y	TPM2B_ECC_PARAMETER	Y coordinate

11.2.5.3 TPM2B_ECC_POINT

This structure is defined to allow a point to be a single sized parameter so that it may be encrypted.

NOTE If the point is to be omitted, the X and Y coordinates need to be individually set to Empty Buffers. The minimum value for size will be four. It is checked indirectly by unmarshaling of the TPMS_ECC_POINT. If the type of *point* were BYTE, then *size* could have been zero. However, this would complicate the process of marshaling the structure.

Table 179 — Definition of {ECC} TPM2B_ECC_POINT Structure

Parameter	Type	Description
size=	UINT16	size of the remainder of this structure
point	TPMS_ECC_POINT	coordinates
#TPM_RC_SIZE		error returned if the unmarshaled size of *point* is not exactly equal to *size*

 Copyright © TCG 2006-2020

11.2.5.4 TPMI_ALG_ECC_SCHEME

Table 180 — Definition of (TPM_ALG_ID) {ECC} TPMI_ALG_ECC_SCHEME Type

Values	Comments
TPM_ALG_!ALG.ax	the ecc signing schemes
TPM_ALG_!ALG.am	key exchange methods
+TPM_ALG_NULL	
#TPM_RC_SCHEME	

11.2.5.5 TPMI_ECC_CURVE

This type enumerates the ECC curves implemented by the TPM.

Table 181 — Definition of {ECC} (TPM_ECC_CURVE) TPMI_ECC_CURVE Type

Parameter	Description
$ECC_CURVES	the list of implemented curves
#TPM_RC_CURVE	error when curve is not supported

11.2.5.6 TPMT_ECC_SCHEME

Table 182 — Definition of (TPMT_SIG_SCHEME) {ECC} TPMT_ECC_SCHEME Structure

Parameter	Type	Description
scheme	+TPMI_ALG_ECC_SCHEME	scheme selector
[scheme]details	TPMU_ASYM_SCHEME	scheme parameters

 Copyright © TCG 2006-2020

11.2.5.7 TPMS_ALGORITHM_DETAIL_ECC

This structure is used to report on the curve parameters of an ECC curve. It is returned by TPM2_ECC_Parameters().

Table 183 — Definition of {ECC} TPMS_ALGORITHM_DETAIL_ECC Structure <OUT>

Parameter	Type	Description
curveID	TPM_ECC_CURVE	identifier for the curve
keySize	UINT16	Size in bits of the key
kdf	TPMT_KDF_SCHEME+	if not TPM_ALG_NULL, the required KDF and hash algorithm used in secret sharing operations
sign	TPMT_ECC_SCHEME+	If not TPM_ALG_NULL, this is the mandatory signature scheme that is required to be used with this curve.
p	TPM2B_ECC_PARAMETER	Fp (the modulus)
a	TPM2B_ECC_PARAMETER	coefficient of the linear term in the curve equation
b	TPM2B_ECC_PARAMETER	constant term for curve equation
gX	TPM2B_ECC_PARAMETER	x coordinate of base point G
gY	TPM2B_ECC_PARAMETER	y coordinate of base point G
n	TPM2B_ECC_PARAMETER	order of G
h	TPM2B_ECC_PARAMETER	cofactor (a size of zero indicates a cofactor of 1)

11.3 Signatures

11.3.1 TPMS_SIGNATURE_RSA

Table 184 — Definition of {RSA} TPMS_SIGNATURE_RSA Structure

Parameter	Type	Description
hash	TPMI_ALG_HASH	the hash algorithm used to digest the message TPM_ALG_NULL is not allowed.
sig	TPM2B_PUBLIC_KEY_RSA	The signature is the size of a public key.

Table 185 — Definition of Types for {RSA} Signature

Type	Name	Description
TPMS_SIGNATURE_RSA	TPMS_SIGNATURE_!ALG.ax	

Copyright © TCG 2006-2020

11.3.2 TPMS_SIGNATURE_ECC

Table 186 — Definition of {ECC} TPMS_SIGNATURE_ECC Structure

Parameter	Type	Description
hash	TPMI_ALG_HASH	the hash algorithm used in the signature process TPM_ALG_NULL is not allowed.
signatureR	TPM2B_ECC_PARAMETER	
signatureS	TPM2B_ECC_PARAMETER	

Table 187 — Definition of Types for {ECC} TPMS_SIGNATURE_ECC

Type	Name	Description
TPMS_SIGNATURE_ECC	TPMS_SIGNATURE_!ALG.ax	

11.3.3 TPMU_SIGNATURE

A TPMU_SIGNATURE_COMPOSITE is a union of the various signatures that are supported by a particular TPM implementation. The union allows substitution of any signature algorithm wherever a signature is required in a structure.

NOTE All TPM are required to support a hash algorithm and the HMAC algorithm.

When a symmetric algorithm is used for signing, the signing algorithm is assumed to be an HMAC based on the indicated hash algorithm. The HMAC key will either be referenced as part of the usage or will be implied by context.

Table 188 — Definition of TPMU_SIGNATURE Union <IN/OUT>

Parameter	Type	Selector	Description
!ALG.ax	TPMS_SIGNATURE_!ALG.ax	TPM_ALG_!ALG.ax	all asymmetric signatures
hmac	TPMT_HA	TPM_ALG_HMAC	HMAC signature (required to be supported)
any	TPMS_SCHEME_HASH		used to access the hash
null		TPM_ALG_NULL	the NULL signature

11.3.4 TPMT_SIGNATURE

Table 189 shows the basic algorithm-agile structure when a symmetric or asymmetric signature is indicated. The *sigAlg* parameter indicates the algorithm used for the signature. This structure is output from commands such as the attestation commands and TPM2_Sign, and is an input to commands such as TPM2_VerifySignature(), TPM2_PolicySigned(), and TPM2_FieldUpgradeStart().

Table 189 — Definition of TPMT_SIGNATURE Structure

Parameter	Type	Description
sigAlg	+TPMI_ALG_SIG_SCHEME	selector of the algorithm used to construct the signature
[sigAlg]signature	TPMU_SIGNATURE	This shall be the actual signature information.

 Copyright © TCG 2006-2020

11.4 Key/Secret Exchange

11.4.1 Introduction

The structures in 11.4 are used when a key or secret is being exchanged. The exchange may be in

- TPM2_StartAuthSession() where the secret is injected for salting the session,
- TPM2_Duplicate(), TPM2_Import, or TPM2_Rewrap() where the secret is the symmetric encryption key for the outer wrapper of a duplication blob, or
- TPM2_ActivateIdentity() or TPM2_CreateIdentity() where the secret is the symmetric encryption key for the credential blob.

Particulars are described in TPM 2.0 Part 1.

11.4.2 TPMU_ENCRYPTED_SECRET

This structure is used to hold either an ephemeral public point for ECDH, an OAEP-encrypted block for RSA, or a symmetrically encrypted value. This structure is defined for the limited purpose of determining the size of a TPM2B_ENCRYPTED_SECRET.

The symmetrically encrypted value may use either CFB or XOR encryption.

NOTE Table 190 is illustrative. It would be modified depending on the algorithms supported in the TPM.

Table 190 — Definition of TPMU_ENCRYPTED_SECRET Union

Parameter	Type	Selector	Description
ecc[sizeof(TPMS_ECC_POINT)]	BYTE	TPM_ALG_ECC	
rsa[MAX_RSA_KEY_BYTES]	BYTE	TPM_ALG_RSA	
symmetric[sizeof(TPM2B_DIGEST)]	BYTE	TPM_ALG_SYMCIPHER	
keyedHash[sizeof(TPM2B_DIGEST)]	BYTE	TPM_ALG_KEYEDHASH	Any symmetrically encrypted secret value will be limited to be no larger than a digest.

11.4.3 TPM2B_ENCRYPTED_SECRET

Table 191 — Definition of TPM2B_ENCRYPTED_SECRET Structure

Parameter	Type	Description
size	UINT16	size of the secret value
secret[size] {:sizeof(TPMU_ENCRYPTED_SECRET)}	BYTE	secret

 Copyright © TCG 2006-2020

12 Key/Object Complex

12.1 Introduction

An object description requires a TPM2B_PUBLIC structure and may require a TPMT_SENSITIVE structure. When the structure is stored off the TPM, the TPMT_SENSITIVE structure is encrypted within a TPM2B_PRIVATE structure.

When the object requires two components for its description, those components are loaded as separate parameters in the TPM2_Load() command. When the TPM creates an object that requires both components, the TPM will return them as separate parameters from the TPM2_Create() operation.

The TPM may produce multiple different TPM2B_PRIVATE structures for a single TPM2B_PUBLIC structure. Creation of a modified TPM2B_PRIVATE structure requires that the full structure be loaded with the TPM2_Load() command, modification of the TPMT_SENSITIVE data, and output of a new TPM2B_PRIVATE structure.

12.2 Public Area Structures

12.2.1 Description

Clause 12.2 defines the TPM2B_PUBLIC structure and the higher-level substructure that may be contained in a TPM2B_PUBLIC. The higher-level structures that are currently defined for inclusion in a TPM2B_PUBLIC are the

- structures for asymmetric keys,
- structures for symmetric keys, and
- structures for sealed data.

12.2.2 TPMI_ALG_PUBLIC

Table 192 — Definition of (TPM_ALG_ID) TPMI_ALG_PUBLIC Type

Values	Comments
TPM_ALG_!ALG.o	All object types
#TPM_RC_TYPE	response code when a public type is not supported

 Copyright © TCG 2006-2020

12.2.3 Type-Specific Parameters

12.2.3.1 Description

The public area contains two fields (*parameters* and *unique*) that vary by object type. The *parameters* field varies according to the *type* of the object but the contents may be the same across multiple instances of a particular *type*. The unique field format also varies according to the type of the object and will also be unique for each instance.

For a symmetric key (*type* == TPM_ALG_SYMCIPHER), HMAC key (*type* == TPM_ALG_KEYEDHASH) or data object (also, *type* == TPM_ALG_KEYEDHASH), the contents of *unique* shall be computed from components of the sensitive area of the object as follows:

$$unique \coloneqq \mathbf{H}_{nameAlg}(seedValue \mid\mid sensitive) \quad (9)$$

where

$\mathbf{H}_{nameAlg}()$	the hash algorithm used to compute the Name of the object
$seedValue$	the digest-sized obfuscation value in the sensitive area of a symmetric key or symmetric data object found in a TPMT_SENSITIVE.*seedValue.buffer*
$sensitive$	the secret key/data of the object in the TPMT_SENSITIVE.*sensitive.any.buffer*

12.2.3.2 TPMU_PUBLIC_ID

This is the union of all values allowed in in the *unique* field of a TPMT_PUBLIC.

NOTE The derive member cannot be unmarshaled in a TPMU_PUBLIC_ID. It is placed in this structure so that the maximum size of a TPM2B_TEMPLATE will be computed correctly.

Table 193 — Definition of TPMU_PUBLIC_ID Union <IN/OUT>

Parameter	Type	Selector	Description
keyedHash	TPM2B_DIGEST	TPM_ALG_KEYEDHASH	
sym	TPM2B_DIGEST	TPM_ALG_SYMCIPHER	
rsa	TPM2B_PUBLIC_KEY_RSA	TPM_ALG_RSA	
ecc	TPMS_ECC_POINT	TPM_ALG_ECC	
derive	TPMS_DERIVE		only allowed for TPM2_CreateLoaded when *parentHandle* is a Derivation Parent.

 Copyright © TCG 2006-2020

12.2.3.3 TPMS_KEYEDHASH_PARMS

This structure describes the parameters that would appear in the public area of a KEYEDHASH object.

NOTE Although the names are the same, the types of the structures are not the same as for asymmetric parameter lists.

Table 194 — Definition of TPMS_KEYEDHASH_PARMS Structure

Parameter	Type	Description
scheme	TPMT_KEYEDHASH_SCHEME+	Indicates the signing method used for a *keyedHash* signing object. This field also determines the size of the data field for a data object created with TPM2_Create() or TPM2_CreatePrimary().

12.2.3.4 TPMS_ASYM_PARMS

This structure contains the common public area parameters for an asymmetric key. The first two parameters of the parameter definition structures of an asymmetric key shall have the same two first components.

NOTE The sign parameter may have a different type in order to allow different schemes to be selected for each asymmetric type but the first parameter of each scheme definition shall be a TPM_ALG_ID for a valid signing scheme.

Table 195 — Definition of TPMS_ASYM_PARMS Structure <>

Parameter	Type	Description
symmetric	TPMT_SYM_DEF_OBJECT+	the companion symmetric algorithm for a restricted decryption key and shall be set to a supported symmetric algorithm This field is optional for keys that are not decryption keys and shall be set to TPM_ALG_NULL if not used.
scheme	TPMT_ASYM_SCHEME+	for a key with the *sign* attribute SET, a valid signing scheme for the key type for a key with the *decrypt* attribute SET, a valid key exchange protocol for a key with sign and decrypt attributes, shall be TPM_ALG_NULL

 Copyright © TCG 2006-2020

12.2.3.5 TPMS_RSA_PARMS

A TPM compatible with this specification and supporting RSA shall support two primes and an *exponent* of zero. An exponent of zero indicates that the exponent is the default of $2^{16} + 1$. Support for other values is optional. Use of other exponents in duplicated keys is not recommended because the resulting keys would not be interoperable with other TPMs.

NOTE Implementations are not required to check that *exponent* is the default exponent. They may fail to load the key if *exponent* is not zero. The reference implementation allows the values listed in the table.

Table 196 — Definition of {RSA} TPMS_RSA_PARMS Structure

Parameter	Type	Description
symmetric	TPMT_SYM_DEF_OBJECT+	for a restricted decryption key, shall be set to a supported symmetric algorithm, key size, and mode. if the key is not a restricted decryption key, this field shall be set to TPM_ALG_NULL.
scheme	TPMT_RSA_SCHEME+	scheme.scheme shall be: for an unrestricted signing key, either TPM_ALG_RSAPSS TPM_ALG_RSASSA or TPM_ALG_NULL for a restricted signing key, either TPM_ALG_RSAPSS or TPM_ALG_RSASSA for an unrestricted decryption key, TPM_ALG_RSAES, TPM_ALG_OAEP, or TPM_ALG_NULL unless the object also has the *sign* attribute for a restricted decryption key, TPM_ALG_NULL NOTE When both sign and decrypt are SET, restricted shall be CLEAR and scheme shall be TPM_ALG_NULL.
keyBits	TPMI_RSA_KEY_BITS	number of bits in the public modulus
exponent	UINT32	the public exponent A prime number greater than 2.

 Copyright © TCG 2006-2020

12.2.3.6 TPMS_ECC_PARMS

This structure contains the parameters for prime modulus ECC.

Table 197 — Definition of {ECC} TPMS_ECC_PARMS Structure

Parameter	Type	Description
symmetric	TPMT_SYM_DEF_OBJECT+	for a restricted decryption key, shall be set to a supported symmetric algorithm, key size. and mode. if the key is not a restricted decryption key, this field shall be set to TPM_ALG_NULL.
scheme	TPMT_ECC_SCHEME+	If the *sign* attribute of the key is SET, then this shall be a valid signing scheme. NOTE If the *sign* parameter in *curveID* indicates a mandatory scheme, then this field shall have the same value. If the *decrypt* attribute of the key is SET, then this shall be a valid key exchange scheme or TPM_ALG_NULL. If the key is a Storage Key, then this field shall be TPM_ALG_NULL.
curveID	TPMI_ECC_CURVE	ECC curve ID
kdf	TPMT_KDF_SCHEME+	an optional key derivation scheme for generating a symmetric key from a Z value If the *kdf* parameter associated with *curveID* is not TPM_ALG_NULL then this is required to be NULL. NOTE There are currently no commands where this parameter has effect and, in the reference code, this field needs to be set to TPM_ALG_NULL.

12.2.3.7 TPMU_PUBLIC_PARMS

Table 198 defines the possible parameter definition structures that may be contained in the public portion of a key. If the Object can be a parent, the first field must be a TPMT_SYM_DEF_OBJECT. See 11.1.7.

Table 198 — Definition of TPMU_PUBLIC_PARMS Union <IN/OUT>

Parameter	Type	Selector	Description(1)
keyedHashDetail	TPMS_KEYEDHASH_PARMS	TPM_ALG_KEYEDHASH	sign | decrypt | neither(2)
symDetail	TPMS_SYMCIPHER_PARMS	TPM_ALG_SYMCIPHER	sign | decrypt | neither(2)
rsaDetail	TPMS_RSA_PARMS	TPM_ALG_RSA	decrypt + sign(2)
eccDetail	TPMS_ECC_PARMS	TPM_ALG_ECC	decrypt + sign(2)
asymDetail	TPMS_ASYM_PARMS		common scheme structure for RSA and ECC keys
NOTES 1) Description column indicates which of TPMA_OBJECT.*decrypt* or TPMA_OBJECT.*sign* may be set. 2) "+" indicates that both may be set but one shall be set. "|" indicates the optional settings.			

 Copyright © TCG 2006-2020

12.2.3.8 TPMT_PUBLIC_PARMS

This structure is used in TPM2_TestParms() to validate that a set of algorithm parameters is supported by the TPM.

Table 199 — Definition of TPMT_PUBLIC_PARMS Structure

Parameter	Type	Description
type	TPMI_ALG_PUBLIC	the algorithm to be tested
[type]parameters	TPMU_PUBLIC_PARMS	the algorithm details

12.2.4 TPMT_PUBLIC

Table 200 defines the public area structure. The Name of the object is *nameAlg* concatenated with the digest of this structure using *nameAlg*.

Table 200 — Definition of TPMT_PUBLIC Structure

Parameter	Type	Description
type	TPMI_ALG_PUBLIC	"algorithm" associated with this object
nameAlg	+TPMI_ALG_HASH	algorithm used for computing the Name of the object NOTE The "+" indicates that the instance of a TPMT_PUBLIC may have a "+" to indicate that the *nameAlg* may be TPM_ALG_NULL.
objectAttributes	TPMA_OBJECT	attributes that, along with *type*, determine the manipulations of this object
authPolicy	TPM2B_DIGEST	optional policy for using this key The policy is computed using the *nameAlg* of the object. NOTE Shall be the Empty Policy if no authorization policy is present.
[type]parameters	TPMU_PUBLIC_PARMS	the algorithm or structure details
[type]unique	TPMU_PUBLIC_ID	the unique identifier of the structure For an asymmetric key, this would be the public key.

12.2.5 TPM2B_PUBLIC

This sized buffer is used to embed a TPMT_PUBLIC in a load command and in any response that returns a public area.

Table 201 — Definition of TPM2B_PUBLIC Structure

Parameter	Type	Description
size=	UINT16	size of publicArea NOTE The "=" will force the TPM to try to unmarshal a TPMT_PUBLIC and check that the unmarshaled size matches the value of *size*. If all the required fields of a TPMT_PUBLIC are not present, the TPM will return an error (generally TPM_RC_SIZE) when attempting to unmarshal the TPMT_PUBLIC.
publicArea	+TPMT_PUBLIC	the public area NOTE The "+" indicates that the caller may specify that use of TPM_ALG_NULL is allowed for *nameAlg.*

 Copyright © TCG 2006-2020

12.2.6 TPM2B_TEMPLATE

This sized buffer is used to embed a TPMT_TEMPLATE for TPM2_CreateLoaded().

Unmarshaling of this structure is fairly complex due to requirements for backwards compatibility. Unlike a TPM2B_PUBLIC, this structure is unmarshaled as an array of bytes that is passed to the action code. The action code will then unmarshal the embedded structure.

If the parent is not a derivation parent, this structure is unmarshaled normally. If the parent is a derivation parent, *unique* is unmarshaled as a TPMS_DERIVE structure (*label* and *context*). See 12.2.3.2.

Table 202 — Definition of TPM2B_TEMPLATE Structure

Parameter	Type	Description
size	UINT16	size of publicArea
buffer[size]{:sizeof(TPMT_PUBLIC)}	BYTE	the public area

12.3 Private Area Structures

12.3.1 Introduction

The structures in 12.2.6 define the contents and construction of the private portion of a TPM object. A TPM2B_PRIVATE along with a TPM2B_PUBLIC are needed to describe a TPM object.

A TPM2B_PRIVATE area may be encrypted by different symmetric algorithms or, in some cases, not encrypted at all.

12.3.2 Sensitive Data Structures

12.3.2.1 Introduction

The structures in 12.3.2 define the presumptive internal representations of the sensitive areas of the various entities. A TPM may store the sensitive information in any desired format but when constructing a TPM_PRIVATE, the formats in 12.3.2 shall be used.

12.3.2.2 TPM2B_PRIVATE_VENDOR_SPECIFIC

This structure is defined for coding purposes. For IO to the TPM, the sensitive portion of the key will be in a canonical form. For an RSA key, this will be one of the prime factors of the public modulus. After loading, it is typical that other values will be computed so that computations using the private key will not need to start with just one prime factor. This structure can be used to store the results of such vendor-specific calculations.

The value for PRIVATE_VENDOR_SPECIFIC_BYTES is determined by the vendor.

Table 203 — Definition of TPM2B_PRIVATE_VENDOR_SPECIFIC Structure

Parameter	Type	Description
size	UINT16	
buffer[size]{:PRIVATE_VENDOR_SPECIFIC_BYTES}	BYTE	

Level 00 Revision 01.59 **Copyright © TCG 2006-2020** November 8, 2019

12.3.2.3 TPMU_SENSITIVE_COMPOSITE

Table 204 — Definition of TPMU_SENSITIVE_COMPOSITE Union <IN/OUT>

Parameter	Type	Selector	Description
rsa	TPM2B_PRIVATE_KEY_RSA	TPM_ALG_RSA	a prime factor of the public key
ecc	TPM2B_ECC_PARAMETER	TPM_ALG_ECC	the integer private key
bits	TPM2B_SENSITIVE_DATA	TPM_ALG_KEYEDHASH	the private data
sym	TPM2B_SYM_KEY	TPM_ALG_SYMCIPHER	the symmetric key
any	TPM2B_PRIVATE_VENDOR_SPECIFIC		vendor-specific size for key storage

12.3.2.4 TPMT_SENSITIVE

authValue shall not be larger than the size of the digest produced by the *nameAlg* of the object. *seedValue* shall be the size of the digest produced by the *nameAlg* of the object.

Table 205 — Definition of TPMT_SENSITIVE Structure

Parameter	Type	Description
sensitiveType	TPMI_ALG_PUBLIC	identifier for the sensitive area This shall be the same as the *type* parameter of the associated public area.
authValue	TPM2B_AUTH	user authorization data The authValue may be a zero-length string.
seedValue	TPM2B_DIGEST	for a parent object, the optional protection seed; for other objects, the obfuscation value
[sensitiveType]sensitive	TPMU_SENSITIVE_COMPOSITE	the type-specific private data

12.3.3 TPM2B_SENSITIVE

The TPM2B_SENSITIVE structure is used as a parameter in TPM2_LoadExternal(). It is an unencrypted sensitive area but it may be encrypted using parameter encryption.

NOTE 1 When this structure is unmarshaled, the *sensitiveType* determines what type of value is unmarshaled. Each value of *sensitiveType* is associated with a TPM2B. It is the maximum size for each of the TPM2B values that will determine if the unmarshal operation is successful. Since there is no selector for the *any* or *vendor* options for the union, the maximum input and output sizes for a TPM2B_SENSITIVE are not affected by the sizes of those parameters.

NOTE 2 The unmarshaling function validates that *size* equals the size of the value that is unmarshaled.

Table 206 — Definition of TPM2B_SENSITIVE Structure <IN/OUT>

Parameter	Type	Description
size	UINT16	size of the *private* structure
sensitiveArea	TPMT_SENSITIVE	an unencrypted sensitive area

 Copyright © TCG 2006-2020

12.3.4 Encryption

A TPMS_SENSITIVE is the input to the encryption process. All TPMS_ENCRYPT structures are CFB-encrypted using a key and Initialization Vector (IV) that are derived from a seed value.

The method of generating the key and IV is described in "Protected Storage" subclause "Symmetric Encryption." in TPM 2.0 Part 1.

12.3.5 Integrity

The integrity computation is used to ensure that a protected object is not modified when stored in memory outside of the TPM.

The method of protecting the integrity of the sensitive area is described in "Protected Storage" subclause "Integrity" in TPM 2.0 Part 1.

12.3.6 _PRIVATE

This structure is defined to size the contents of a TPM2B_PRIVATE. This structure is not directly marshaled or unmarshaled.

For TPM2_Duplicate() and TPM2_Import(), the TPM2B_PRIVATE may contain multiply encrypted data and two integrity values. In some cases, the sensitive data is not encrypted and the integrity value is not present.

For TPM2_Load() and TPM2_Create(), *integrityInner* is always present.

If *integrityInner* is present, it and *sensitive* are encrypted as a single block.

When an integrity value is not needed, it is not present and it is not represented by an Empty Buffer.

Table 207 — Definition of _PRIVATE Structure <>

Parameter	Type	Description
integrityOuter	TPM2B_DIGEST	
integrityInner	TPM2B_DIGEST	could also be a TPM2B_IV
sensitive	TPM2B_SENSITIVE	the sensitive area

12.3.7 TPM2B_PRIVATE

The TPM2B_PRIVATE structure is used as a parameter in multiple commands that create, load, and modify the sensitive area of an object.

When the TPM returns a TPM2B_PRIVATE structure, the TPM pads the TPM2B_AUTH to its maximum size.

Table 208 — Definition of TPM2B_PRIVATE Structure <IN/OUT>

Parameter	Type	Description
size	UINT16	size of the *private* structure
buffer[size] {:sizeof(_PRIVATE)}	BYTE	an encrypted private area

Level 00 Revision 01.59 **Copyright © TCG 2006-2020** November 8, 2019

12.4 Identity Object

12.4.1 Description

An identity object is used to convey credential protection value (CV) to a TPM that can load the object associated with the object. The CV is encrypted to a storage key on the target TPM, and if the credential integrity checks and the proper object is loaded in the TPM, then the TPM will return the CV.

12.4.2 TPMS_ID_OBJECT

This structure is used for sizing the TPM2B_ID_OBJECT.

Table 209 — Definition of TPMS_ID_OBJECT Structure <>

Parameter	Type	Description
integrityHMAC	TPM2B_DIGEST	HMAC using the nameAlg of the storage key on the target TPM
encIdentity	TPM2B_DIGEST	credential protector information returned if name matches the referenced object All of the *encIdentity* is encrypted, including the size field. NOTE The TPM is not required to check that the size is not larger than the digest of the *nameAlg*. However, if the size is larger, the ID object may not be usable on a TPM that has no digest larger than produced by *nameAlg*.

12.4.3 TPM2B_ID_OBJECT

This structure is an output from TPM2_MakeCredential() and is an input to TPM2_ActivateCredential().

Table 210 — Definition of TPM2B_ID_OBJECT Structure <IN/OUT>

Parameter	Type	Description
size	UINT16	size of the *credential* structure
credential[size]{:sizeof(TPMS_ID_OBJECT)}	BYTE	an encrypted credential area

 Copyright © TCG 2006-2020

13 NV Storage Structures

13.1 TPM_NV_INDEX

A TPM_NV_INDEX is used to reference a defined location in NV memory. The format of the Index is changed from TPM 1.2 in order to include the Index in the reserved handle space. Handles in this range use the digest of the public area of the Index as the Name of the entity in authorization computations

The 32-bit TPM 1.2 NV Index format is shown in Figure 4. In order to allow the Index to fit into the 24 bits available in the reserved handle space, the Index value format is changed as shown in Figure 5.

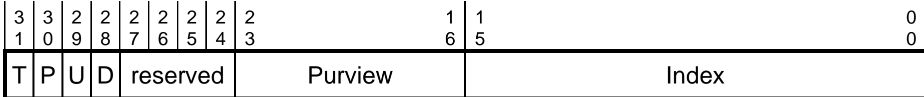


Figure 4 — TPM 1.2 TPM_NV_INDEX

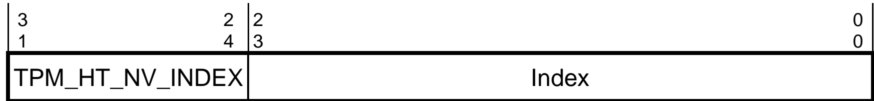


Figure 5 — TPM 2.0 TPM_NV_INDEX

NOTE This TPM_NV_INDEX format does not retain the Purview field and the D bit is not a part of an Index handle as in TPM 1.2. The TPMA_NV_PLATFORMCREATE attribute is a property of an Index that provides functionality similar to the D bit.

A valid Index handle will have an MSO of TPM_HT_NV_INDEX.

NOTE This structure is not used. It is defined here to indicate how the fields of the handle are assigned. The exemplary unmarshaling code unmarshals a TPM_HANDLE and validates that it is in the range for a TPM_NV_INDEX.

Table 211 — Definition of (UINT32) TPM_NV_INDEX Bits <>

Bit	Name	Definition
23:00	index	The Index of the NV location
31:24	RH_NV	constant value of TPM_HT_NV_INDEX indicating the NV Index range

 Copyright © TCG 2006-2020

Some prior versions of this specification contained a table here (Options for space Field of TPM_NV_INDEX) that assigned subsets of the index field to different entities. Since this assignment was a convention and not an architectural element of the TPM, the table was removed and the information is now contained in a registry document that is maintained by the TCG.

13.2 TPM_NT

This table lists the values of the TPM_NT field of a TPMA_NV. See Table 214 for usage.

Table 212 — Definition of TPM_NT Constants

Name	Value	Description
TPM_NT_ORDINARY	0x0	Ordinary – contains data that is opaque to the TPM that can only be modified using TPM2_NV_Write().
TPM_NT_COUNTER	0x1	Counter – contains an 8-octet value that is to be used as a counter and can only be modified with TPM2_NV_Increment()
TPM_NT_BITS	0x2	Bit Field – contains an 8-octet value to be used as a bit field and can only be modified with TPM2_NV_SetBits().
TPM_NT_EXTEND	0x4	Extend – contains a digest-sized value used like a PCR. The Index can only be modified using TPM2_NV_Extend(). The extend will use the nameAlg of the Index.
TPM_NT_PIN_FAIL	0x8	PIN Fail - contains *pinCount* that increments on a PIN authorization failure and a *pinLimit*
TPM_NT_PIN_PASS	0x9	PIN Pass - contains *pinCount* that increments on a PIN authorization success and a *pinLimit*

All other TPM_NT values are reserved and TPM2_NV_DefineSpace() returns TPM_RC_ATTRIBUTES.

NOTE 1 These values are compatible with previous versions of this specification, which used a bit map for this field.

NOTE 2 This field described by Table 212 is 4 bits.

13.3 TPMS_NV_PIN_COUNTER_PARAMETERS

This is the data that can be written to and read from a TPM_NT_PIN_PASS or TPM_NT_PIN_FAIL non-volatile index. *pinCount* is the most significant octets. *pinLimit* is the least significant octets.

Table 213 — Definition of TPMS_NV_PIN_COUNTER_PARAMETERS Structure

Parameter	Type	Description
pinCount	UINT32	This counter shows the current number of successful authValue authorization attempts to access a TPM_NT_PIN_PASS index or the current number of unsuccessful authValue authorization attempts to access a TPM_NT_PIN_FAIL index.
pinLimit	UINT32	This threshold is the value of *pinCount* at which the authValue authorization of the host TPM_NT_PIN_PASS or TPM_NT_PIN_FAIL index is locked out.

13.4 TPMA_NV (NV Index Attributes)

This structure allows the TPM to keep track of the data and permissions to manipulate an NV Index.

 Copyright © TCG 2006-2020

The platform controls (TPMA_NV_PPWRITE and TPMA_NV_PPREAD) and owner controls (TPMA_NV_OWNERWRITE and TPMA_NV_OWNERREAD) give the platform and owner access to NV Indexes using Platform Authorization or Owner Authorization rather than the *authValue* or *authPolicy* of the Index.

If access to an NV Index is to be restricted based on PCR, then an appropriate *authPolicy* shall be provided.

NOTE *platformAuth* or *ownerAuth* can be provided in any type of authorization session or as a password.

If TPMA_NV_AUTHREAD is SET, then the Index may be read if the Index *authValue* is provided. If TPMA_NV_POLICYREAD is SET, then the Index may be read if the Index *authPolicy* is satisfied.

At least one of TPMA_NV_PPREAD, TPMA_NV_OWNERREAD, TPMA_NV_AUTHREAD, or TPMA_NV_POLICYREAD shall be SET.

If TPMA_NV_AUTHWRITE is SET, then the Index may be written if the Index *authValue* is provided. If TPMA_NV_POLICYWRITE is SET, then the Index may be written if the Index *authPolicy* is satisfied.

At least one of TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE TPMA_NV_AUTHWRITE, or TPMA_NV_POLICYWRITE shall be SET.

If TPMA_NV_WRITELOCKED is SET, then the Index may not be written. If TPMA_NV_WRITEDEFINE is SET, TPMA_NV_WRITELOCKED may not be CLEAR except by deleting and redefining the Index. If TPMA_NV_WRITEDEFINE is CLEAR, then TPMA_NV_WRITELOCKED will be CLEAR on the next TPM2_Startup(TPM_SU_CLEAR).

NOTE If TPMA_NV_WRITELOCKED is SET, but TPMA_NV_WRITTEN is CLEAR, then TPMA_NV_WRITELOCKED is CLEAR by TPM Reset or TPM Restart. This action occurs even if the TPMA_NV_WRITEDEFINE attribute is SET. This action prevents an NV Index from being defined that can never be written, and permits a use case where an Index is defined, but the user wants to prohibit writes until after a reboot.

If TPMA_NV_READLOCKED is SET, then the Index may not be read. TPMA_NV_READLOCKED will be CLEAR on the next TPM2_Startup(TPM_SU_CLEAR).

NOTE The TPM is expected to maintain indicators to indicate that the Index is temporarily locked. The state of these indicators is reported in the TPMA_NV_READLOCKED and TPMA_NV_WRITELOCKED attributes.

If the TPM_NT is TPM_NT_EXTEND, then writes to the Index will cause an update of the Index using the extend operation with the *nameAlg* used to create the digest.

If TPM_NT is TPM_NT_PIN_FAIL, TPMA_NV_NO_DA must be SET. This removes ambiguity over which Dictionary Attack defense protects a TPM_NV_PIN_FAIL's *authValue*.

When the Index is created (TPM2_NV_DefineSpace()), TPMA_NV_WRITELOCKED, TPMA_NV_READLOCKED, and TPMA_NV_WRITTEN shall all be CLEAR in the parameter that defines the attributes of the created Index.

Level 00 Revision 01.59 **Copyright © TCG 2006-2020** November 8, 2019

Table 214 — Definition of (UINT32) TPMA_NV Bits

Bit	Name	Description
0	TPMA_NV_PPWRITE	**SET (1):** The Index data can be written if Platform Authorization is provided. **CLEAR (0):** Writing of the Index data cannot be authorized with Platform Authorization.
1	TPMA_NV_OWNERWRITE	**SET (1):** The Index data can be written if Owner Authorization is provided. **CLEAR (0):** Writing of the Index data cannot be authorized with Owner Authorization.
2	TPMA_NV_AUTHWRITE	**SET (1):** Authorizations to change the Index contents that require USER role may be provided with an HMAC session or password. **CLEAR (0):** Authorizations to change the Index contents that require USER role may not be provided with an HMAC session or password.
3	TPMA_NV_POLICYWRITE	**SET (1):** Authorizations to change the Index contents that require USER role may be provided with a policy session. **CLEAR (0):** Authorizations to change the Index contents that require USER role may not be provided with a policy session. NOTE TPM2_NV_ChangeAuth() always requires that authorization be provided in a policy session.
7:4	TPM_NT	The type of the index. NOTE A TPM is not required to support all TPM_NT values
9:8	Reserved	shall be zero reserved for future use
10	TPMA_NV_POLICY_DELETE	**SET (1):** Index may not be deleted unless the *authPolicy* is satisfied using TPM2_NV_UndefineSpaceSpecial(). **CLEAR (0):** Index may be deleted with proper platform or owner authorization using TPM2_NV_UndefineSpace(). NOTE An Index with this attribute and a policy that cannot be satisfied (e.g., an Empty Policy) cannot be deleted.
11	TPMA_NV_WRITELOCKED	**SET (1):** Index cannot be written. **CLEAR (0):** Index can be written.
12	TPMA_NV_WRITEALL	**SET (1):** A partial write of the Index data is not allowed. The write size shall match the defined space size. **CLEAR (0):** Partial writes are allowed. This setting is required if the .*dataSize* of the Index is larger than NV_MAX_BUFFER_SIZE for the implementation.
13	TPMA_NV_WRITEDEFINE	**SET (1):** TPM2_NV_WriteLock() may be used to prevent further writes to this location. **CLEAR (0):** TPM2_NV_WriteLock() does not block subsequent writes if TPMA_NV_WRITE_STCLEAR is also CLEAR.
14	TPMA_NV_WRITE_STCLEAR	**SET (1):** TPM2_NV_WriteLock() may be used to prevent further writes to this location until the next TPM Reset or TPM Restart. **CLEAR (0):** TPM2_NV_WriteLock() does not block subsequent writes if TPMA_NV_WRITEDEFINE is also CLEAR.
15	TPMA_NV_GLOBALLOCK	**SET (1):** If TPM2_NV_GlobalWriteLock() is successful, TPMA_NV_WRITELOCKED is set. **CLEAR (0):** TPM2_NV_GlobalWriteLock() has no effect on the writing of the data at this Index.

 Copyright © TCG 2006-2020

Bit	Name	Description
16	TPMA_NV_PPREAD	**SET (1):** The Index data can be read if Platform Authorization is provided. **CLEAR (0):** Reading of the Index data cannot be authorized with Platform Authorization.
17	TPMA_NV_OWNERREAD	**SET (1):** The Index data can be read if Owner Authorization is provided. **CLEAR (0):** Reading of the Index data cannot be authorized with Owner Authorization.
18	TPMA_NV_AUTHREAD	**SET (1):** The Index data may be read if the *authValue* is provided. **CLEAR (0):** Reading of the Index data cannot be authorized with the Index *authValue*.
19	TPMA_NV_POLICYREAD	**SET (1):** The Index data may be read if the *authPolicy* is satisfied. **CLEAR (0):** Reading of the Index data cannot be authorized with the Index *authPolicy*.
24:20	Reserved	shall be zero reserved for future use
25	TPMA_NV_NO_DA	**SET (1):** Authorization failures of the Index do not affect the DA logic and authorization of the Index is not blocked when the TPM is in Lockout mode. **CLEAR (0):** Authorization failures of the Index will increment the authorization failure counter and authorizations of this Index are not allowed when the TPM is in Lockout mode.
26	TPMA_NV_ORDERLY	**SET (1):** NV Index state is only required to be saved when the TPM performs an orderly shutdown (TPM2_Shutdown()). **CLEAR (0):** NV Index state is required to be persistent after the command to update the Index completes successfully (that is, the NV update is synchronous with the update command).
27	TPMA_NV_CLEAR_STCLEAR	**SET (1):** TPMA_NV_WRITTEN for the Index is CLEAR by TPM Reset or TPM Restart. **CLEAR (0):** TPMA_NV_WRITTEN is not changed by TPM Restart. NOTE This attribute may only be SET if TPM_NT is not TPM_NT_COUNTER.
28	TPMA_NV_READLOCKED	**SET (1):** Reads of the Index are blocked until the next TPM Reset or TPM Restart. **CLEAR (0):** Reads of the Index are allowed if proper authorization is provided.
29	TPMA_NV_WRITTEN	**SET (1):** Index has been written. **CLEAR (0):** Index has not been written.
30	TPMA_NV_PLATFORMCREATE	**SET (1):** This Index may be undefined with Platform Authorization but not with Owner Authorization. **CLEAR (0):** This Index may be undefined using Owner Authorization but not with Platform Authorization. The TPM will validate that this attribute is SET when the Index is defined using Platform Authorization and will validate that this attribute is CLEAR when the Index is defined using Owner Authorization.
31	TPMA_NV_READ_STCLEAR	**SET (1):** TPM2_NV_ReadLock() may be used to SET TPMA_NV_READLOCKED for this Index. **CLEAR (0):** TPM2_NV_ReadLock() has no effect on this Index.

 Copyright © TCG 2006-2020

13.5 TPMS_NV_PUBLIC

This structure describes an NV Index.

Table 215 — Definition of TPMS_NV_PUBLIC Structure

Name	Type	Description
nvIndex	TPMI_RH_NV_INDEX	the handle of the data area
nameAlg	TPMI_ALG_HASH	hash algorithm used to compute the name of the Index and used for the *authPolicy.* For an extend index, the hash algorithm used for the extend.
attributes	TPMA_NV	the Index attributes
authPolicy	TPM2B_DIGEST	optional access policy for the Index The policy is computed using the *nameAlg* NOTE Shall be the Empty Policy if no authorization policy is present.
dataSize{:MAX_NV_INDEX_SIZE}	UINT16	the size of the data area The maximum size is implementation-dependent. The minimum maximum size is platform-specific.
#TPM_RC_SIZE		response code returned when the requested size is too large for the implementation

13.6 TPM2B_NV_PUBLIC

This structure is used when a TPMS_NV_PUBLIC is sent on the TPM interface.

Table 216 — Definition of TPM2B_NV_PUBLIC Structure

Name	Type	Description
size=	UINT16	size of *nvPublic*
nvPublic	TPMS_NV_PUBLIC	the public area

 Copyright © TCG 2006-2020

14 Context Data

14.1 Introduction

Clause 14 defines the contents of the TPM2_ContextSave() response parameters and TPM2_ContextLoad() command parameters.

If the parameters provided by the caller in TPM2_ContextLoad() do not match the values returned by the TPM when the context was saved, the integrity check of the TPM2B_CONTEXT will fail and the object or session will not be loaded.

14.2 TPM2B_CONTEXT_SENSITIVE

This structure holds the object or session context data. When saved, the full structure is encrypted.

NOTE This is an informative table that is included in the specification only to allow calculation of the maximum size for TPM2B_CONTEXT_DATA.

Table 217 — Definition of TPM2B_CONTEXT_SENSITIVE Structure <IN/OUT>

Parameter	Type	Description
size	UINT16	
buffer[size]{:MAX_CONTEXT_SIZE}	BYTE	the sensitive data

14.3 TPMS_CONTEXT_DATA

This structure holds the integrity value and the encrypted data for a context.

NOTE This is an informative table that is included in the specification only to allow calculation of the maximum size for TPM2B_CONTEXT_DATA.

Table 218 — Definition of TPMS_CONTEXT_DATA Structure <IN/OUT>

Parameter	Type	Description
integrity	TPM2B_DIGEST	the integrity value
encrypted	TPM2B_CONTEXT_SENSITIVE	the sensitive area

14.4 TPM2B_CONTEXT_DATA

This structure is used in a TPMS_CONTEXT.

Table 219 — Definition of TPM2B_CONTEXT_DATA Structure <IN/OUT>

Parameter	Type	Description
size	UINT16	
buffer[size] {:sizeof(TPMS_CONTEXT_DATA)}	BYTE	

Copyright © TCG 2006-2020

14.5 TPMS_CONTEXT

This structure is used in TPM2_ContextLoad() and TPM2_ContextSave(). If the values of the TPMS_CONTEXT structure in TPM2_ContextLoad() are not the same as the values when the context was saved (TPM2_ContextSave()), then the TPM shall not load the context.

Saved object contexts shall not be loaded as long as the associated hierarchy is disabled.

Saved object contexts are invalidated when the Primary Seed of their hierarchy changes. Objects in the Endorsement hierarchy are invalidated when either the EPS or SPS is changed.

When an object has the *stClear* attribute, it shall not be possible to reload the context or any descendant object after a TPM Reset or TPM Restart.

NOTE 1 The reference implementation prevents reloads after TPM Restart by including the current value of a *clearCount* in the saved object context. When an object is loaded, this value is compared with the current value of the *clearCount* if the object has the *stClear* attribute. If the values are not the same, then the object cannot be loaded.

A sequence value is contained within *contextBlob*, the integrity-protected part of the saved context. The sequence value is repeated in the *sequence* parameter of the TPMS_CONTEXT structure. The *sequence* parameter, along with other values, is used in the generation the protection values of the context.

NOTE 2 The reference implementation prepends the *sequence* value to the *contextBlob* before, for example, the SESSION structure for sessions or the OBJECT structure for transient objects.

If the integrity value of the context is valid, but the *sequence* value of the decrypted context does not match the value in the *sequence* parameter, then TPM shall enter the failure mode because this is indicative of a specific type of attack on the context values.

NOTE 3 If the integrity value is correct, but the decryption fails and produces the wrong value for sequence, this implies that either the TPM is faulty or an external entity is able to forge an integrity value for the context but they have insufficient information to know the encryption key of the context. Since the TPM generated the valid context, then there is no reason for the sequence value in the context to be decrypted incorrectly other than the TPM is faulty or the TPM is under attack. In either case, it is appropriate for the TPM to enter failure more.

Table 220 — Definition of TPMS_CONTEXT Structure

Name	Type	Description
sequence	UINT64	the sequence number of the context NOTE Transient object contexts and session contexts used different counters.
savedHandle	TPMI_DH_SAVED	a handle indicating if the context is a session, object, or sequence object (see Table 221 — Context Handle Values
hierarchy	TPMI_RH_HIERARCHY+	the hierarchy of the context
contextBlob	TPM2B_CONTEXT_DATA	the context data and integrity HMAC

 Copyright © TCG 2006-2020

14.6 Parameters of TPMS_CONTEXT

14.6.1 *sequence*

The *sequence* parameter is used to differentiate the contexts and to allow the TPM to create a different encryption key for each context. Objects and sessions use different sequence counters. The sequence counter for objects (transient and sequence) is incremented when an object context is saved, and the sequence counter for sessions increments when a session is created or when it is loaded (TPM2_ContextLoad()). The session sequence number is the *contextID* counter.

For a session, the sequence number also allows the TRM to find the "older" contexts so that they may be refreshed if the *contextID* are too widely separated.

If an input value for *sequence* is larger than the value used in any saved context, the TPM shall return an error (TPM_RC_VALUE) and do no additional processing of the context.

If the context is a session context and the input value for sequence is less than the current value of *contextID* minus the maximum range for sessions, the TPM shall return an error (TPM_RC_VALUE) and do no additional processing of the context.

14.6.2 *savedHandle*

For a session, this is the handle that was assigned to the session when it was created. For a transient object, the handle will have one of the values shown in Table 221.

If the handle type for *savedHandle* is TPM_HT_TRANSIENT, then the low order bits are used to differentiate static objects from sequence objects.

If an input value for handle is outside of the range of values used by the TPM, the TPM shall return an error (TPM_RC_VALUE) and do no additional processing of the context.

Table 221 — Context Handle Values

Value	Description
0x02xxxxxx	an HMAC session context
0x03xxxxxx	a policy session context
0x80000000	an ordinary transient object
0x80000001	a sequence object
0x80000002	a transient object with the *stClear* attribute SET

 Copyright © TCG 2006-2020

14.6.3 *hierarchy*

This is the hierarchy (TPMI_RH_HIERARCHY) for the saved context and determines the proof value used in the construction of the encryption and integrity values for the context. For session and sequence contexts, the hierarchy is TPM_RC_NULL. The hierarchy for a transient object may be TPM_RH_NULL but it is not required.

14.7 Context Protection

14.7.1 Context Integrity

The integrity of the context blob is protected by an HMAC. The integrity value is constructed such that changes to the component values will invalidate the context and prevent it from being loaded.

Previously saved contexts for objects in the Platform hierarchy shall not be loadable after the PPS is changed.

Previously saved contexts for objects in the Storage hierarchy shall not be loadable after the SPS is changed.

Previously saved contexts for objects in the Endorsement hierarchy shall not be loadable after either the EPS or SPS is changed.

Previously saved sessions shall not be loadable after the SPS changes.

Previously saved contexts for objects that have their *stClear* attribute SET shall not be loadable after a TPM Restart. If a Storage Key has its *stClear* attribute SET, the descendants of this key shall not be loadable after TPM Restart.

Previously saved contexts for a session and objects shall not be loadable after a TPM Reset.

A saved context shall not be loaded if its HMAC is not valid. The equation for computing the HMAC for a context is found in “Context Integrity Protection” in TPM 2.0 Part 1.

14.7.2 Context Confidentiality

The context data of sessions and objects shall be protected by symmetric encryption using CFB. The method for computing the IV and encryption key is found in “Context Confidentiality Protection” in TPM 2.0 Part 1.

 Copyright © TCG 2006-2020

15 Creation Data

15.1 TPMS_CREATION_DATA

This structure provides information relating to the creation environment for the object. The creation data includes the parent Name, parent Qualified Name, and the digest of selected PCR. These values represent the environment in which the object was created. Creation data allows a relying party to determine if an object was created when some appropriate protections were present.

When the object is created, the structure shown in Table 222 is generated and a ticket is computed over this data.

If the parent is a permanent handle (TPM_RH_OWNER, TPM_RH_PLATFORM, TPM_RH_ENDORSEMENT, or TPM_RH_NULL), then *parentName* and *parentQualifiedName* will be set to the parent handle value and *parentNameAlg* will be TPM_ALG_NULL.

Table 222 — Definition of TPMS_CREATION_DATA Structure <OUT>

Parameter	Type	Description
pcrSelect	TPML_PCR_SELECTION	list indicating the PCR included in *pcrDigest*
pcrDigest	TPM2B_DIGEST	digest of the selected PCR using *nameAlg* of the object for which this structure is being created *pcrDigest.size* shall be zero if the *pcrSelect* list is empty.
locality	TPMA_LOCALITY	the locality at which the object was created
parentNameAlg	TPM_ALG_ID	*nameAlg* of the parent
parentName	TPM2B_NAME	Name of the parent at time of creation The size will match digest size associated with *parentNameAlg* unless it is TPM_ALG_NULL, in which case the size will be 4 and *parentName* will be the hierarchy handle.
parentQualifiedName	TPM2B_NAME	Qualified Name of the parent at the time of creation Size is the same as *parentName.*
outsideInfo	TPM2B_DATA	association with additional information added by the key creator This will be the contents of the *outsideInfo* parameter in TPM2_Create() or TPM2_CreatePrimary().

15.2 TPM2B_CREATION_DATA

This structure is created by TPM2_Create() and TPM2_CreatePrimary(). It is never entered into the TPM and never has a size of zero.

Table 223 — Definition of TPM2B_CREATION_DATA Structure <OUT>

Parameter	Type	Description
size=	UINT16	size of the creation data
creationData	TPMS_CREATION_DATA	

 Copyright © TCG 2006-2020

16 Attached Component Structures

16.1 TPM_AT

These constants are used in TPM2_AC_GetCapability() to indicate the first tagged value returned from an attached component.

TPM_AT values of 0x80000000 through 0xFFFFFFFF are reserved for vendor-specific values.

Table 224 — Definition of (UINT32) TPM_AT Constants

Name	Value	Comments
TPM_AT_ANY	0x00000000	in a command, a non-specific request for AC information; in a response, indicates that *outputData* is not meaningful
TPM_AT_ERROR	0x00000001	indicates a TCG defined, device-specific error
TPM_AT_PV1	0x00000002	indicates the most significant 32 bits of a pairing value for the AC
TPM_AT_VEND	0x80000000	value added to a TPM_AT to indicate a vendor-specific tag value

16.2 TPM_AE

These constants are the TCG-defined error values returned by an AC.

Table 225 — Definition of (UINT32) TPM_AE Constants <OUT>

Name	Value	Comments
TPM_AE_NONE	0x00000000	in a command, a non-specific request for AC information; in a response, indicates that *outputData* is not meaningful

16.3 TPMS_AC_OUTPUT

TPMS_AC_OUTPUT is used to return information about an AC. The *tag* structure parameter indicates the type of the *data* value.

Table 226 — Definition of TPMS_AC_OUTPUT Structure <OUT>

Parameter	Type	Description
tag	TPM_AT	tag indicating the contents of *data*
data	UINT32	the data returned from the AC

 Copyright © TCG 2006-2020

16.4 TPML_AC_CAPABILITIES

This list is only used in TPM2_AC_GetCapability().

The values in the list are returned in TPM_AT order (see Table 224) with vendor-specific values returned after TCG defined values.

NOTE MAX_AC_CAPABILITIES = MAX_CAP_DATA / sizeof(TPMS_AC_OUTPUT)

Table 227 — Definition of TPML_AC_CAPABILITIES Structure <OUT>

Parameter	Type	Description
count	UINT32	number of values in the *acCapabilities* list; may be 0
acCapabilities[count] {:MAX_AC_CAPABILITIES}	TPMS_AC_OUTPUT	a list of AC values

 Copyright © TCG 2006-2020

Trusted Platform Module Library

Part 3: Commands

Family "2.0"

Level 00 Revision 01.59

November 8, 2019

Published

Contact: admin@trustedcomputinggroup.org

TCG Published

Copyright © TCG 2006-2020

Licenses and Notices

Copyright Licenses:

- Trusted Computing Group (TCG) grants to the user of the source code in this specification (the "Source Code") a worldwide, irrevocable, nonexclusive, royalty free, copyright license to reproduce, create derivative works, distribute, display and perform the Source Code and derivative works thereof, and to grant others the rights granted herein.
- The TCG grants to the user of the other parts of the specification (other than the Source Code) the rights to reproduce, distribute, display, and perform the specification solely for the purpose of developing products based on such documents.

Source Code Distribution Conditions:

- Redistributions of Source Code must retain the above copyright licenses, this list of conditions and the following disclaimers.
- Redistributions in binary form must reproduce the above copyright licenses, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.

Disclaimers:

- THE COPYRIGHT LICENSES SET FORTH ABOVE DO NOT REPRESENT ANY FORM OF LICENSE OR WAIVER, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, WITH RESPECT TO PATENT RIGHTS HELD BY TCG MEMBERS (OR OTHER THIRD PARTIES) THAT MAY BE NECESSARY TO IMPLEMENT THIS SPECIFICATION OR OTHERWISE. Contact TCG Administration (admin@trustedcomputinggroup.org) for information on specification licensing rights available through TCG membership agreements.
- THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO EXPRESS OR IMPLIED WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, ACCURACY, COMPLETENESS, OR NONINFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.
- Without limitation, TCG and its members and licensors disclaim all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification and to the implementation of this specification, and TCG disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this specification or any information herein.

Any marks and brands contained herein are the property of their respective owners.

CONTENTS

1 Scope ... 1
2 Terms and Definitions ... 1
3 Symbols and abbreviated terms ... 1
4 Notation ... 2
4.1 Introduction ... 2
4.2 Table Decorations ... 2
4.3 Handle and Parameter Demarcation ... 3
4.4 AuthorizationSize and ParameterSize ... 3
4.5 Return Code Alias ... 4
5 Command Processing ... 4
5.1 Introduction ... 4
5.2 Command Header Validation ... 4
5.3 Mode Checks ... 5
5.4 Handle Area Validation ... 5
5.5 Session Area Validation ... 6
5.6 Authorization Checks ... 7
5.7 Parameter Decryption ... 9
5.8 Parameter Unmarshaling ... 9
5.9 Command Post Processing ... 11
6 Response Values ... 12
6.1 Tag ... 12
6.2 Response Codes ... 12
7 Implementation Dependent ... 15
8 Detailed Actions Assumptions ... 16
8.1 Introduction ... 16
8.2 Pre-processing ... 16
8.3 Post Processing ... 16
9 Start-up ... 17
9.1 Introduction ... 17
9.2 _TPM_Init ... 17
9.3 TPM2_Startup ... 19
9.4 TPM2_Shutdown ... 27
10 Testing ... 31
10.1 Introduction ... 31
10.2 TPM2_SelfTest ... 32
10.3 TPM2_IncrementalSelfTest ... 35
10.4 TPM2_GetTestResult ... 38
11 Session Commands ... 41
11.1 TPM2_StartAuthSession ... 41
11.2 TPM2_PolicyRestart ... 47
12 Object Commands ... 50
12.1 TPM2_Create ... 50

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

12.2 TPM2_Load ... 56
12.3 TPM2_LoadExternal ... 60
12.4 TPM2_ReadPublic ... 65
12.5 TPM2_ActivateCredential ... 68
12.6 TPM2_MakeCredential ... 72
12.7 TPM2_Unseal ... 75
12.8 TPM2_ObjectChangeAuth ... 78
12.9 TPM2_CreateLoaded ... 81

13 Duplication Commands ... 86

13.1 TPM2_Duplicate ... 86
13.2 TPM2_Rewrap ... 90
13.3 TPM2_Import ... 94

14 Asymmetric Primitives ... 100

14.1 Introduction ... 100
14.2 TPM2_RSA_Encrypt ... 100
14.3 TPM2_RSA_Decrypt ... 104
14.4 TPM2_ECDH_KeyGen ... 108
14.5 TPM2_ECDH_ZGen ... 111
14.6 TPM2_ECC_Parameters ... 114
14.7 TPM2_ZGen_2Phase ... 117

15 Symmetric Primitives ... 121

15.1 Introduction ... 121
15.2 TPM2_EncryptDecrypt ... 123
15.3 TPM2_EncryptDecrypt2 ... 127
15.4 TPM2_Hash ... 130
15.5 TPM2_HMAC ... 133
15.6 TPM2_MAC ... 137

16 Random Number Generator ... 140

16.1 TPM2_GetRandom ... 140
16.2 TPM2_StirRandom ... 143

17 Hash/HMAC/Event Sequences ... 146

17.1 Introduction ... 146
17.2 TPM2_HMAC_Start ... 146
17.3 TPM2_MAC_Start ... 150
17.4 TPM2_HashSequenceStart ... 153
17.5 TPM2_SequenceUpdate ... 156
17.6 TPM2_SequenceComplete ... 160
17.7 TPM2_EventSequenceComplete ... 164

18 Attestation Commands ... 168

18.1 Introduction ... 168
18.2 TPM2_Certify ... 170
18.3 TPM2_CertifyCreation ... 173
18.4 TPM2_Quote ... 177
18.5 TPM2_GetSessionAuditDigest ... 180
18.6 TPM2_GetCommandAuditDigest ... 183

 Copyright © TCG 2006-2020

18.7 TPM2_GetTime 187
18.8 TPM2_CertifyX509 189

19 Ephemeral EC Keys 197

19.1 Introduction 197
19.2 TPM2_Commit 198
19.3 TPM2_EC_Ephemeral 203

20 Signing and Signature Verification 206

20.1 TPM2_VerifySignature 206
20.2 TPM2_Sign 209

21 Command Audit 213

21.1 Introduction 213
21.2 TPM2_SetCommandCodeAuditStatus 214

22 Integrity Collection (PCR) 217

22.1 Introduction 217
22.2 TPM2_PCR_Extend 218
22.3 TPM2_PCR_Event 221
22.4 TPM2_PCR_Read 224
22.5 TPM2_PCR_Allocate 227
22.6 TPM2_PCR_SetAuthPolicy 230
22.7 TPM2_PCR_SetAuthValue 233
22.8 TPM2_PCR_Reset 236
22.9 _TPM_Hash_Start 239
22.10 _TPM_Hash_Data 241
22.11 _TPM_Hash_End 243

23 Enhanced Authorization (EA) Commands 246

23.1 Introduction 246
23.2 Signed Authorization Actions 247
23.3 TPM2_PolicySigned 251
23.4 TPM2_PolicySecret 257
23.5 TPM2_PolicyTicket 261
23.6 TPM2_PolicyOR 265
23.7 TPM2_PolicyPCR 268
23.8 TPM2_PolicyLocality 273
23.9 TPM2_PolicyNV 277
23.10 TPM2_PolicyCounterTimer 281
23.11 TPM2_PolicyCommandCode 285
23.12 TPM2_PolicyPhysicalPresence 288
23.13 TPM2_PolicyCpHash 291
23.14 TPM2_PolicyNameHash 295
23.15 TPM2_PolicyDuplicationSelect 299
23.16 TPM2_PolicyAuthorize 303
23.17 TPM2_PolicyAuthValue 307
23.18 TPM2_PolicyPassword 310
23.19 TPM2_PolicyGetDigest 313
23.20 TPM2_PolicyNvWritten 316
23.21 TPM2_PolicyTemplate 319

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

23.22 TPM2_PolicyAuthorizeNV 323

24 Hierarchy Commands 327

24.1 TPM2_CreatePrimary 327
24.2 TPM2_HierarchyControl 331
24.3 TPM2_SetPrimaryPolicy 335
24.4 TPM2_ChangePPS 339
24.5 TPM2_ChangeEPS 342
24.6 TPM2_Clear 345
24.7 TPM2_ClearControl 349
24.8 TPM2_HierarchyChangeAuth 352

25 Dictionary Attack Functions 355

25.1 Introduction 355
25.2 TPM2_DictionaryAttackLockReset 355
25.3 TPM2_DictionaryAttackParameters 358

26 Miscellaneous Management Functions 361

26.1 Introduction 361
26.2 TPM2_PP_Commands 361
26.3 TPM2_SetAlgorithmSet 364

27 Field Upgrade 367

27.1 Introduction 367
27.2 TPM2_FieldUpgradeStart 369
27.3 TPM2_FieldUpgradeData 372
27.4 TPM2_FirmwareRead 375

28 Context Management 378

28.1 Introduction 378
28.2 TPM2_ContextSave 378
28.3 TPM2_ContextLoad 383
28.4 TPM2_FlushContext 388
28.5 TPM2_EvictControl 391

29 Clocks and Timers 396

29.1 TPM2_ReadClock 396
29.2 TPM2_ClockSet 399
29.3 TPM2_ClockRateAdjust 402

30 Capability Commands 405

30.1 Introduction 405
30.2 TPM2_GetCapability 405
30.3 TPM2_TestParms 413

31 Non-volatile Storage 416

31.1 Introduction 416
31.2 NV Counters 417
31.3 TPM2_NV_DefineSpace 418
31.4 TPM2_NV_UndefineSpace 424
31.5 TPM2_NV_UndefineSpaceSpecial 427
31.6 TPM2_NV_ReadPublic 430

 Copyright © TCG 2006-2020

31.7 TPM2_NV_Write 433
31.8 TPM2_NV_Increment 437
31.9 TPM2_NV_Extend 441
31.10 TPM2_NV_SetBits 445
31.11 TPM2_NV_WriteLock 448
31.12 TPM2_NV_GlobalWriteLock 451
31.13 TPM2_NV_Read 454
31.14 TPM2_NV_ReadLock 457
31.15 TPM2_NV_ChangeAuth 460
31.16 TPM2_NV_Certify 463

32 Attached Components 467

32.1 Introduction 467
32.2 TPM2_AC_GetCapability 468
32.3 TPM2_AC_Send 471
32.4 TPM2_Policy_AC_SendSelect 475

33 Authenticated Countdown Timer 479

33.1 Introduction 479
33.2 TPM2_ACT_SetTimeout 479

34 Vendor Specific 482

34.1 Introduction 482
34.2 TPM2_Vendor_TCG_Test 482

 Copyright © TCG 2006-2020

Tables

Table 1 — Command Modifiers and Decoration 2
Table 2 — Separators 3
Table 3 — Unmarshaling Errors 10
Table 4 — Command-Independent Response Codes 13
Table 5 — TPM2_Startup Command 22
Table 6 — TPM2_Startup Response 22
Table 7 — TPM2_Shutdown Command 28
Table 8 — TPM2_Shutdown Response 28
Table 9 — TPM2_SelfTest Command 33
Table 10 — TPM2_SelfTest Response 33
Table 11 — TPM2_IncrementalSelfTest Command 36
Table 12 — TPM2_IncrementalSelfTest Response 36
Table 13 — TPM2_GetTestResult Command 39
Table 14 — TPM2_GetTestResult Response 39
Table 15 — TPM2_StartAuthSession Command 43
Table 16 — TPM2_StartAuthSession Response 43
Table 17 — TPM2_PolicyRestart Command 48
Table 18 — TPM2_PolicyRestart Response 48
Table 19 — TPM2_Create Command 53
Table 20 — TPM2_Create Response 53
Table 21 — TPM2_Load Command 57
Table 22 — TPM2_Load Response 57
Table 23 — TPM2_LoadExternal Command 62
Table 24 — TPM2_LoadExternal Response 62
Table 25 — TPM2_ReadPublic Command 66
Table 26 — TPM2_ReadPublic Response 66
Table 27 — TPM2_ActivateCredential Command 69
Table 28 — TPM2_ActivateCredential Response 69
Table 29 — TPM2_MakeCredential Command 73
Table 30 — TPM2_MakeCredential Response 73
Table 31 — TPM2_Unseal Command 76
Table 32 — TPM2_Unseal Response 76
Table 33 — TPM2_ObjectChangeAuth Command 79
Table 34 — TPM2_ObjectChangeAuth Response 79
Table 35 — TPM2_CreateLoaded Command 82
Table 36 — TPM2_CreateLoaded Response 82
Table 37 — TPM2_Duplicate Command 87

 Copyright © TCG 2006-2020

Table 38 — TPM2_Duplicate Response 87
Table 39 — TPM2_Rewrap Command 91
Table 40 — TPM2_Rewrap Response 91
Table 41 — TPM2_Import Command 96
Table 42 — TPM2_Import Response 96
Table 43 — Padding Scheme Selection 100
Table 44 — Message Size Limits Based on Padding 101
Table 45 — TPM2_RSA_Encrypt Command 102
Table 46 — TPM2_RSA_Encrypt Response 102
Table 47 — TPM2_RSA_Decrypt Command 105
Table 48 — TPM2_RSA_Decrypt Response 105
Table 49 — TPM2_ECDH_KeyGen Command 109
Table 50 — TPM2_ECDH_KeyGen Response 109
Table 51 — TPM2_ECDH_ZGen Command 112
Table 52 — TPM2_ECDH_ZGen Response 112
Table 53 — TPM2_ECC_Parameters Command 115
Table 54 — TPM2_ECC_Parameters Response 115
Table 55 — TPM2_ZGen_2Phase Command 118
Table 56 — TPM2_ZGen_2Phase Response 118
Table 57 — Symmetric Chaining Process 122
Table 58 — TPM2_EncryptDecrypt Command 124
Table 59 — TPM2_EncryptDecrypt Response 124
Table 60 — TPM2_EncryptDecrypt2 Command 128
Table 61 — TPM2_EncryptDecrypt2 Response 128
Table 62 — TPM2_Hash Command 131
Table 63 — TPM2_Hash Response 131
Table 64 — TPM2_HMAC Command 134
Table 65 — TPM2_HMAC Response 134
Table 66 — TPM2_MAC Command 138
Table 67 — TPM2_MAC Response 138
Table 68 — TPM2_GetRandom Command 141
Table 69 — TPM2_GetRandom Response 141
Table 70 — TPM2_StirRandom Command 144
Table 71 — TPM2_StirRandom Response 144
Table 72 — Hash Selection Matrix 146
Table 73 — TPM2_HMAC_Start Command 147
Table 74 — TPM2_HMAC_Start Response 147
Table 75 — Algorithm Selection Matrix 150
Table 76 — TPM2_MAC_Start Command 151

 Copyright © TCG 2006-2020

Table 77 — TPM2_MAC_Start Response 151
Table 78 — TPM2_HashSequenceStart Command 154
Table 79 — TPM2_HashSequenceStart Response 154
Table 80 — TPM2_SequenceUpdate Command 157
Table 81 — TPM2_SequenceUpdate Response 157
Table 82 — TPM2_SequenceComplete Command 161
Table 83 — TPM2_SequenceComplete Response 161
Table 84 — TPM2_EventSequenceComplete Command 165
Table 85 — TPM2_EventSequenceComplete Response 165
Table 86 — TPM2_Certify Command 171
Table 87 — TPM2_Certify Response 171
Table 88 — TPM2_CertifyCreation Command 174
Table 89 — TPM2_CertifyCreation Response 174
Table 90 — TPM2_Quote Command 178
Table 91 — TPM2_Quote Response 178
Table 92 — TPM2_GetSessionAuditDigest Command 181
Table 93 — TPM2_GetSessionAuditDigest Response 181
Table 94 — TPM2_GetCommandAuditDigest Command 184
Table 95 — TPM2_GetCommandAuditDigest Response 184
Table 96 — TPM2_GetTime Command 188
Table 97 — TPM2_GetTime Response 188
Table 98 — TPM2_CertifyX509 Command 192
Table 99 — TPM2_CertifyX509 Response 192
Table 100 — TPM2_Commit Command 199
Table 101 — TPM2_Commit Response 199
Table 102 — TPM2_EC_Ephemeral Command 204
Table 103 — TPM2_EC_Ephemeral Response 204
Table 104 — TPM2_VerifySignature Command 207
Table 105 — TPM2_VerifySignature Response 207
Table 106 — TPM2_Sign Command 210
Table 107 — TPM2_Sign Response 210
Table 108 — TPM2_SetCommandCodeAuditStatus Command 215
Table 109 — TPM2_SetCommandCodeAuditStatus Response 215
Table 110 — TPM2_PCR_Extend Command 219
Table 111 — TPM2_PCR_Extend Response 219
Table 112 — TPM2_PCR_Event Command 222
Table 113 — TPM2_PCR_Event Response 222
Table 114 — TPM2_PCR_Read Command 225
Table 115 — TPM2_PCR_Read Response 225

 Copyright © TCG 2006-2020

Table 116 — TPM2_PCR_Allocate Command 228
Table 117 — TPM2_PCR_Allocate Response 228
Table 118 — TPM2_PCR_SetAuthPolicy Command 231
Table 119 — TPM2_PCR_SetAuthPolicy Response 231
Table 120 — TPM2_PCR_SetAuthValue Command 234
Table 121 — TPM2_PCR_SetAuthValue Response 234
Table 122 — TPM2_PCR_Reset Command 237
Table 123 — TPM2_PCR_Reset Response 237
Table 124 — TPM2_PolicySigned Command 253
Table 125 — TPM2_PolicySigned Response 253
Table 126 — TPM2_PolicySecret Command 258
Table 127 — TPM2_PolicySecret Response 258
Table 128 — TPM2_PolicyTicket Command 262
Table 129 — TPM2_PolicyTicket Response 262
Table 130 — TPM2_PolicyOR Command 266
Table 131 — TPM2_PolicyOR Response 266
Table 132 — TPM2_PolicyPCR Command 270
Table 133 — TPM2_PolicyPCR Response 270
Table 134 — TPM2_PolicyLocality Command 274
Table 135 — TPM2_PolicyLocality Response 274
Table 136 — TPM2_PolicyNV Command 278
Table 137 — TPM2_PolicyNV Response 278
Table 138 — TPM2_PolicyCounterTimer Command 282
Table 139 — TPM2_PolicyCounterTimer Response 282
Table 140 — TPM2_PolicyCommandCode Command 286
Table 141 — TPM2_PolicyCommandCode Response 286
Table 142 — TPM2_PolicyPhysicalPresence Command 289
Table 143 — TPM2_PolicyPhysicalPresence Response 289
Table 144 — TPM2_PolicyCpHash Command 292
Table 145 — TPM2_PolicyCpHash Response 292
Table 146 — TPM2_PolicyNameHash Command 296
Table 147 — TPM2_PolicyNameHash Response 296
Table 148 — TPM2_PolicyDuplicationSelect Command 300
Table 149 — TPM2_PolicyDuplicationSelect Response 300
Table 150 — TPM2_PolicyAuthorize Command 304
Table 151 — TPM2_PolicyAuthorize Response 304
Table 152 — TPM2_PolicyAuthValue Command 308
Table 153 — TPM2_PolicyAuthValue Response 308
Table 154 — TPM2_PolicyPassword Command 311

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

Table 155 — TPM2_PolicyPassword Response 311
Table 156 — TPM2_PolicyGetDigest Command 314
Table 157 — TPM2_PolicyGetDigest Response 314
Table 158 — TPM2_PolicyNvWritten Command 317
Table 159 — TPM2_PolicyNvWritten Response 317
Table 160 — TPM2_PolicyTemplate Command 320
Table 161 — TPM2_PolicyTemplate Response 320
Table 162 — TPM2_PolicyAuthorizeNV Command 324
Table 163 — TPM2_PolicyAuthorizeNV Response 324
Table 164 — TPM2_CreatePrimary Command 328
Table 165 — TPM2_CreatePrimary Response 328
Table 166 — TPM2_HierarchyControl Command 332
Table 167 — TPM2_HierarchyControl Response 332
Table 168 — TPM2_SetPrimaryPolicy Command 336
Table 169 — TPM2_SetPrimaryPolicy Response 336
Table 170 — TPM2_ChangePPS Command 340
Table 171 — TPM2_ChangePPS Response 340
Table 172 — TPM2_ChangeEPS Command 343
Table 173 — TPM2_ChangeEPS Response 343
Table 174 — TPM2_Clear Command 346
Table 175 — TPM2_Clear Response 346
Table 176 — TPM2_ClearControl Command 350
Table 177 — TPM2_ClearControl Response 350
Table 178 — TPM2_HierarchyChangeAuth Command 353
Table 179 — TPM2_HierarchyChangeAuth Response 353
Table 180 — TPM2_DictionaryAttackLockReset Command 356
Table 181 — TPM2_DictionaryAttackLockReset Response 356
Table 182 — TPM2_DictionaryAttackParameters Command 359
Table 183 — TPM2_DictionaryAttackParameters Response 359
Table 184 — TPM2_PP_Commands Command 362
Table 185 — TPM2_PP_Commands Response 362
Table 186 — TPM2_SetAlgorithmSet Command 365
Table 187 — TPM2_SetAlgorithmSet Response 365
Table 188 — TPM2_FieldUpgradeStart Command 370
Table 189 — TPM2_FieldUpgradeStart Response 370
Table 190 — TPM2_FieldUpgradeData Command 373
Table 191 — TPM2_FieldUpgradeData Response 373
Table 192 — TPM2_FirmwareRead Command 376
Table 193 — TPM2_FirmwareRead Response 376

 Copyright © TCG 2006-2020

Table 194 — TPM2_ContextSave Command 379
Table 195 — TPM2_ContextSave Response 379
Table 196 — TPM2_ContextLoad Command 384
Table 197 — TPM2_ContextLoad Response 384
Table 198 — TPM2_FlushContext Command 389
Table 199 — TPM2_FlushContext Response 389
Table 200 — TPM2_EvictControl Command 393
Table 201 — TPM2_EvictControl Response 393
Table 202 — TPM2_ReadClock Command 397
Table 203 — TPM2_ReadClock Response 397
Table 204 — TPM2_ClockSet Command 400
Table 205 — TPM2_ClockSet Response 400
Table 206 — TPM2_ClockRateAdjust Command 403
Table 207 — TPM2_ClockRateAdjust Response 403
Table 208 — TPM2_GetCapability Command 409
Table 209 — TPM2_GetCapability Response 409
Table 210 — TPM2_TestParms Command 414
Table 211 — TPM2_TestParms Response 414
Table 212 — TPM2_NV_DefineSpace Command 420
Table 213 — TPM2_NV_DefineSpace Response 420
Table 214 — TPM2_NV_UndefineSpace Command 425
Table 215 — TPM2_NV_UndefineSpace Response 425
Table 216 — TPM2_NV_UndefineSpaceSpecial Command 428
Table 217 — TPM2_NV_UndefineSpaceSpecial Response 428
Table 218 — TPM2_NV_ReadPublic Command 431
Table 219 — TPM2_NV_ReadPublic Response 431
Table 220 — TPM2_NV_Write Command 434
Table 221 — TPM2_NV_Write Response 434
Table 222 — TPM2_NV_Increment Command 438
Table 223 — TPM2_NV_Increment Response 438
Table 224 — TPM2_NV_Extend Command 442
Table 225 — TPM2_NV_Extend Response 442
Table 226 — TPM2_NV_SetBits Command 446
Table 227 — TPM2_NV_SetBits Response 446
Table 228 — TPM2_NV_WriteLock Command 449
Table 229 — TPM2_NV_WriteLock Response 449
Table 230 — TPM2_NV_GlobalWriteLock Command 452
Table 231 — TPM2_NV_GlobalWriteLock Response 452
Table 232 — TPM2_NV_Read Command 455

Table 233 — TPM2_NV_Read Response 455
Table 234 — TPM2_NV_ReadLock Command 458
Table 235 — TPM2_NV_ReadLock Response 458
Table 236 — TPM2_NV_ChangeAuth Command 461
Table 237 — TPM2_NV_ChangeAuth Response 461
Table 238 — TPM2_NV_Certify Command 464
Table 239 — TPM2_NV_Certify Response 464
Table 240 — TPM2_AC_GetCapability Command 469
Table 241 — TPM2_AC_GetCapability Response 469
Table 242 — TPM2_AC_Send Command 472
Table 243 — TPM2_AC_Send Response 472
Table 244 — TPM2_Policy_AC_SendSelect Command 476
Table 245 — TPM2_Policy_AC_SendSelect Response 476
Table 246 — TPM2_ACT_SetTimeout Command 480
Table 247 — TPM2_ACT_SetTimeout Response 480
Table 248 — TPM2_Vendor_TCG_Test Command 483
Table 249 — TPM2_Vendor_TCG_Test Response 483

Trusted Platform Module Library
Part 3: Commands

1 Scope

This TPM 2.0 Part 3 of the *Trusted Platform Module Library* specification contains the definitions of the TPM commands. These commands make use of the constants, flags, structures, and union definitions defined in TPM 2.0 Part 2.

The detailed description of the operation of the commands is written in the C language with extensive comments. The behavior of the C code in this TPM 2.0 Part 3 is normative but does not fully describe the behavior of a TPM. The combination of this TPM 2.0 Part 3 and TPM 2.0 Part 4 is sufficient to fully describe the required behavior of a TPM.

The code in parts 3 and 4 is written to define the behavior of a compliant TPM. In some cases (e.g., firmware update), it is not possible to provide a compliant implementation. In those cases, any implementation provided by the vendor that meets the general description of the function provided in TPM 2.0 Part 3 would be compliant.

The code in parts 3 and 4 is not written to meet any particular level of conformance nor does this specification require that a TPM meet any particular level of conformance.

2 Terms and Definitions

For the purposes of this document, the terms and definitions given in TPM 2.0 Part 1 apply.

3 Symbols and abbreviated terms

For the purposes of this document, the symbols and abbreviated terms given in TPM 2.0 Part 1 apply.

 Copyright © TCG 2006-2020

4 Notation

4.1 Introduction

For the purposes of this document, the notation given in TPM 2.0 Part 1 applies.

Command and response tables use various decorations to indicate the fields of the command and the allowed types. These decorations are described in this clause.

4.2 Table Decorations

The symbols and terms in the Notation column of Table 1 are used in the tables for the command schematics. These values indicate various qualifiers for the parameters or descriptions with which they are associated.

Table 1 — Command Modifiers and Decoration

Notation	Meaning
+	A Type decoration – When appended to a value in the Type column of a command, this symbol indicates that the parameter is allowed to use the “null” value of the data type (see in TPM 2.0 Part 2, *Conditional Types*). The null value is usually TPM_RH_NULL for a handle or TPM_ALG_NULL for an algorithm selector. NOTE This decoration is not appended to response parameters.
@	A Name decoration – When this symbol precedes a handle parameter in the “Name” column, it indicates that an authorization session is required for use of the entity associated with the handle. If a handle does not have this symbol, then an authorization session is not allowed.
+PP	A Description modifier – This modifier may follow TPM_RH_PLATFORM in the “Description” column to indicate that Physical Presence is required when *platformAuth*/*platformPolicy* is provided.
+{PP}	A Description modifier – This modifier may follow TPM_RH_PLATFORM to indicate that Physical Presence may be required when *platformAuth*/*platformPolicy* is provided. The commands with this notation may be in the *setList* or *clearList* of TPM2_PP_Commands().
{NV}	A Description modifier – This modifier may follow the *commandCode* in the “Description” column to indicate that the command may result in an update of NV memory and be subject to rate throttling by the TPM. If the command code does not have this notation, then a write to NV memory does not occur as part of the command actions. NOTE Any command that uses authorization may cause a write to NV if there is an authorization failure. A TPM may use the occasion of command execution to update the NV copy of clock.
{F}	A Description modifier – This modifier indicates that the “flushed” attribute will be SET in the TPMA_CC for the command. The modifier may follow the *commandCode* in the “Description” column to indicate that any transient handle context used by the command will be flushed from the TPM when the command completes. This may be combined with the {NV} modifier but not with the {E} modifier. EXAMPLE 1 {NV F} EXAMPLE 2 TPM2_SequenceComplete() will flush the context associated with the *sequenceHandle*.
{E}	A Description modifier – This modifier indicates that the “extensive” attribute will be SET in the TPMA_CC for the command. This modifier may follow the *commandCode* in the “Description” column to indicate that the command may flush many objects and re-enumeration of the loaded context likely will be required. This may be combined with the {NV} modifier but not with the {F} modifier. EXAMPLE 1 {NV E} EXAMPLE 2 TPM2_Clear() will flush all contexts associated with the Storage hierarchy and the Endorsement hierarchy.

 Copyright © TCG 2006-2020

Notation	Meaning
Auth Index:	A Description modifier – When a handle has a “@” decoration, the “Description” column will contain an “Auth Index:” entry for the handle. This entry indicates the number of the authorization session. The authorization sessions associated with handles will occur in the session area in the order of the handles with the “@” modifier. Sessions used only for encryption/decryption or only for audit will follow the handles used for authorization.
Auth Role:	A Description modifier – This will be in the “Description” column of a handle with the “@” decoration. It may have a value of USER, ADMIN or DUP. If the handle has the Auth Role of USER and the handle is an Object, the type of authorization is determined by the setting of *userWithAuth* in the Object's attributes. If the handle is TPM_RH_OWNER, TPM_RH_ENDORSEMENT, or TPM_RH_PLATFORM, operation is as if *userWithAuth* is SET. If the handle references an NV Index, then the allowed authorizations are determined by the settings of the attributes of the NV Index as described in TPM 2.0 Part 2, "TPMA_NV (NV Index Attributes)." If the Auth Role is ADMIN and the handle is an Object, the type of authorization is determined by the setting of *adminWithPolicy* in the Object's attributes. If the handle is TPM_RH_OWNER, TPM_RH_ENDORSEMENT, or TPM_RH_PLATFORM, operation is as if *adminWithPolicy* is SET. If the handle is an NV index, operation is as if *adminWithPolicy* is SET (see 5.6 e)2)). If the DUP role is selected, authorization may only be with a policy session (DUP role only applies to Objects). When either ADMIN or DUP role is selected, a policy command that selects the command being authorized is required to be part of the policy. EXAMPLE TPM2_Certify requires the ADMIN role for the first handle (*objectHandle*). The policy authorization for objectHandle is required to contain TPM2_PolicyCommandCode(*commandCode* == TPM_CC_Certify). This sets the state of the policy so that it can be used for ADMIN role authorization in TPM2_Certify().

4.3 Handle and Parameter Demarcation

The demarcations between the header, handle, and parameter parts are indicated by:

Table 2 — Separators

Separator	Meaning
(dashed line)	the values immediately following are in the handle area
(double line)	the values immediately following are in the parameter area

4.4 AuthorizationSize and ParameterSize

Authorization sessions are not shown in the command or response schematics. When the tag of a command or response is TPM_ST_SESSIONS, then a 32-bit value will be present in the command/response buffer to indicate the size of the authorization field or the parameter field. This value shall immediately follow the handle area (which may contain no handles). For a command, this value (*authorizationSize*) indicates the size of the Authorization Area and shall have a value of 9 or more. For a response, this value (*parameterSize*) indicates the size of the parameter area and may have a value of zero.

If the *authorizationSize* field is present in the command, *parameterSize* will be present in the response, but only if the *responseCode* is TPM_RC_SUCCESS.

When authorization is required to use the TPM entity associated with a handle, then at least one session will be present. To indicate this, the command *tag* Description field contains TPM_ST_SESSIONS. Addional sessions for audit, encrypt, and decrypt may be present.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

When the command *tag* Description field contains TPM_ST_NO_SESSIONS, then no sessions are allowed and the *authorizationSize* field is not present.

When a command allows use of sessions when not required, the command *tag* Description field will indicate the types of sessions that may be used with the command.

4.5 Return Code Alias

For the RC_FMT1 return codes that may add a parameter, handle, or session number, the prefix TPM_RCS_ is an alias for TPM_RC_.

TPM_RC_n is added, where n is the parameter, handle, or session number. In addition, TPM_RC_H is added for handle, TPM_RC_P for parameter, and TPM_RC_S for session errors.

NOTE TPM_RCS_ is a programming convention. Programmers should only add numbers to TPM_RCS_ return codes, never TPM_RC_ return codes. Only return codes that can have a number added have the TPM_RCS_ alias defined. Attempting to use a TPM_RCS_ return code that does not have the TPM_RCS_ alias will cause a compiler error.

EXAMPLE 1 Since TPM_RC_VALUE can have a number added, TPM_RCS_VALUE is defined. A program can use the construct "TPM_RCS_VALUE + number". Since TPM_RC_SIGNATURE cannot have a number added, TPM_RCS_SIGNATURE is not defined. A program using the construct "TPM_RCS_SIGNATURE + number" will not compile, alerting the programmer that the construct is incorrect.

By convention, the number to be added is of the form RC_CommandName_ParameterName where CommmandName is the name of the command with the TPM2_ prefix removed. The parameter name alone is insufficient because the same parameter name could be in a different position in different commands.

EXAMPLE 2 TPM2_HMAC_Start with parameters that result in TPM_ALG_NULL as the hash algorithm will returns TPM_RC_VALUE plus the parameter number. Since *hashAlg* is the second parameter, This code results:

```
#define RC_HMAC_Start_hashAlg          (TPM_RC_P + TPM_RC_2)

return TPM_RCS_VALUE + RC_HMAC_Start_hashAlg;
```

5 Command Processing

5.1 Introduction

This clause defines the command validations that are required of any implementation and the response code returned if the indicated check fails. Unless stated otherwise, the order of the checks is not normative and different TPM may give different responses when a command has multiple errors.

In the description below, some statements that describe a check may be followed by a response code in parentheses. This is the normative response code should the indicated check fail. A normative response code may also be included in the statement.

5.2 Command Header Validation

Before a TPM may begin the actions associated with a command, a set of command format and consistency checks shall be performed. These checks are listed below and should be performed in the indicated order.

a) The TPM shall successfully unmarshal a TPMI_ST_COMMAND_TAG and verify that it is either TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS (TPM_RC_BAD_TAG).

 Copyright © TCG 2006-2020

b) The TPM shall successfully unmarshal a UINT32 as the *commandSize*. If the TPM has an interface buffer that is loaded by some hardware process, the number of octets in the input buffer for the command reported by the hardware process shall exactly match the value in *commandSize* (TPM_RC_COMMAND_SIZE).

NOTE A TPM may have direct access to system memory and unmarshal directly from that memory.

c) The TPM shall successfully unmarshal a TPM_CC and verify that the command is implemented (TPM_RC_COMMAND_CODE).

5.3 Mode Checks

The following mode checks shall be performed in the order listed:

a) If the TPM is in Failure mode, then the *commandCode* is TPM_CC_GetTestResult or TPM_CC_GetCapability (TPM_RC_FAILURE) and the command *tag* is TPM_ST_NO_SESSIONS (TPM_RC_FAILURE).

NOTE 1 In Failure mode, the TPM has no cryptographic capability and processing of sessions is not supported.

b) The TPM is in Field Upgrade mode (FUM), the *commandCode* is TPM_CC_FieldUpgradeData (TPM_RC_UPGRADE).

c) If the TPM has not been initialized (TPM2_Startup()), then the *commandCode* is TPM_CC_Startup (TPM_RC_INITIALIZE).

NOTE 2 The TPM may enter Failure mode during _TPM_Init processing, before TPM2_Startup(). Since the platform firmware cannot know that the TPM is in Failure mode without accessing it, and since the first command is required to be TPM2_Startup(), the expected sequence will be that platform firmware (the CRTM) will issue TPM2_Startup() and receive TPM_RC_FAILURE indicating that the TPM is in Failure mode.

There may be failures where a TPM cannot record that it received TPM2_Startup(). In those cases, a TPM in failure mode may process TPM2_GetTestResult(), TPM2_GetCapability(), or the field upgrade commands. As a side effect, that TPM may process TPM2_GetTestResult(), TPM2_GetCapability() or the field upgrade commands before TPM2_Startup().

This is a corner case exception to the rule that TPM2_Startup() must be the first command.

The mode checks may be performed before or after the command header validation.

5.4 Handle Area Validation

After successfully unmarshaling and validating the command header, the TPM shall perform the following checks on the handles and sessions. These checks may be performed in any order.

NOTE 1 A TPM is required to perform the handle area validation before the authorization checks because an authorization cannot be performed unless the authorization values and attributes for the referenced entity are known by the TPM. For them to be known, the referenced entity must be in the TPM and accessible.

a) The TPM shall successfully unmarshal the number of handles required by the command and validate that the value of the handle is consistent with the command syntax. If not, the TPM shall return TPM_RC_VALUE.

NOTE 2 The TPM may unmarshal a handle and validate that it references an entity on the TPM before unmarshaling a subsequent handle.

NOTE 3 If the submitted command contains fewer handles than required by the syntax of the command, the TPM may continue to read into the next area and attempt to interpret the data as a handle.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

b) For all handles in the handle area of the command, the TPM will validate that the referenced entity is present in the TPM.

1) If the handle references a transient object, the handle shall reference a loaded object (TPM_RC_REFERENCE_H0 + N where N is the number of the handle in the command).

NOTE 4 If the hierarchy for a transient object is disabled, then the transient objects will be flushed so this check will fail.

2) If the handle references a persistent object, then

i) the hierarchy associated with the object (platform or storage, based on the handle value) is enabled (TPM_RC_HANDLE);

ii) the handle shall reference a persistent object that is currently in TPM non-volatile memory (TPM_RC_HANDLE);

iii) if the handle references a persistent object that is associated with the endorsement hierarchy, that the endorsement hierarchy is not disabled (TPM_RC_HANDLE); and

NOTE 5 The reference implementation keeps an internal attribute, passed down from a primary key to its descendents, indicating the object's hierarchy.

iv) if the TPM implementation moves a persistent object to RAM for command processing then sufficient RAM space is available (TPM_RC_OBJECT_MEMORY).

3) If the handle references an NV Index, then

i) an Index exists that corresponds to the handle (TPM_RC_HANDLE); and

ii) the hierarchy associated with the existing NV Index is not disabled (TPM_RC_HANDLE).

iii) If the command requires write access to the index data then TPMA_NV_WRITELOCKED is not SET (TPM_RC_NV_LOCKED)

iv) If the command requires read access to the index data then TPMA_NV_READLOCKED is not SET (TPM_RC_NV_LOCKED)

4) If the handle references a session, then the session context shall be present in TPM memory (TPM_RC_REFERENCE_H0 + N).

5) If the handle references a primary seed for a hierarchy (TPM_RH_ENDORSEMENT, TPM_RH_OWNER, or TPM_RH_PLATFORM) then the enable for the hierarchy is SET (TPM_RC_HIERARCHY).

6) If the handle references a PCR, then the value is within the range of PCR supported by the TPM (TPM_RC_VALUE)

NOTE 6 In the reference implementation, this TPM_RC_VALUE is returned by the unmarshaling code for a TPMI_DH_PCR.

5.5 Session Area Validation

a) If the tag is TPM_ST_SESSIONS and the command requires TPM_ST_NO_SESSIONS, the TPM will return TPM_RC_AUTH_CONTEXT.

b) If the tag is TPM_ST_NO_SESSIONS and the command requires TPM_ST_SESSIONS, the TPM will return TPM_RC_AUTH_MISSING.

c) If the tag is TPM_ST_SESSIONS, the TPM will attempt to unmarshal an *authorizationSize* and return TPM_RC_AUTHSIZE if the value is not within an acceptable range.

1) The minimum value is (sizeof(TPM_HANDLE) + sizeof(UINT16) + sizeof(TPMA_SESSION) + sizeof(UINT16)).

 Copyright © TCG 2006-2020

2) The maximum value of authorizationSize is equal to commandSize – (sizeof(TPM_ST) + sizeof(UINT32) + sizeof(TPM_CC) + (N * sizeof(TPM_HANDLE)) + sizeof(UINT32)) where N is the number of handles associated with the *commandCode* and may be zero.

NOTE 1 (sizeof(TPM_ST) + sizeof(UINT32) + sizeof(TPM_CC)) is the size of a command header. The last UINT32 contains the authorizationSize octets, which are not counted as being in the authorization session area.

d) The TPM will unmarshal the authorization sessions and perform the following validations:

1) If the session handle is not a handle for an HMAC session, a handle for a policy session, or, TPM_RS_PW then the TPM shall return TPM_RC_HANDLE.

2) If the session is not loaded, the TPM will return the warning TPM_RC_REFERENCE_S0 + N where N is the number of the session. The first session is session zero, N = 0.

NOTE 2 If the HMAC and policy session contexts use the same memory, the type of the context must match the type of the handle.

3) If the maximum allowed number of sessions have been unmarshaled and fewer octets than indicated in *authorizationSize* were unmarshaled (that is, *authorizationSize* is too large), the TPM shall return TPM_RC_AUTHSIZE.

4) The consistency of the authorization session attributes is checked.

i) Only one session is allowed for:

(a) session auditing (TPM_RC_ATTRIBUTES) – this session may be used for encrypt or decrypt but may not be a session that is also used for authorization;

(b) decrypting a command parameter (TPM_RC_ATTRIBUTES) – this may be any of the authorization sessions, or the audit session, or a session may be added for the single purpose of decrypting a command parameter, as long as the total number of sessions does not exceed three; and

(c) encrypting a response parameter (TPM_RC_ATTRIBUTES) – this may be any of the authorization sessions, or the audit session if present, ora session may be added for the single purpose of encrypting a response parameter, as long as the total number of sessions does not exceed three.

NOTE 3 A session used for decrypting a command parameter may also be used for encrypting a response parameter.

ii) If a session is not being used for authorization, at least one of decrypt, encrypt, or audit must be SET. (TPM_RC_ATTRIBUTES).

5) An authorization session is present for each of the handles with the “@” decoration (TPM_RC_AUTH_MISSING).

5.6 Authorization Checks

After unmarshaling and validating the handles and the consistency of the authorization sessions, the authorizations shall be checked. Authorization checks only apply to handles if the handle in the command schematic has the “@” decoration. Authorization checks must be performed in this order.

a) The public and sensitive portions of the object shall be present on the TPM (TPM_RC_AUTH_UNAVAILABLE).

b) If the associated handle is TPM_RH_PLATFORM, and the command requires confirmation with physical presence, then physical presence is asserted (TPM_RC_PP).

c) If the object or NV Index is subject to DA protection, and the authorization is with an HMAC or password, then the TPM is not in lockout (TPM_RC_LOCKOUT).

 Copyright © TCG 2006-2020

NOTE 1 An object is subject to DA protection if its *noDA* attribute is CLEAR. An NV Index is subject to DA protection if its TPMA_NV_NO_DA attribute is CLEAR.

NOTE 2 An HMAC or password is required in a policy session when the policy contains TPM2_PolicyAuthValue() or TPM2_PolicyPassword().

d) If the command requires a handle to have DUP role authorization, then the associated authorization session is a policy session (TPM_RC_AUTH_TYPE).

e) If the command requires a handle to have ADMIN role authorization:

1) If the entity being authorized is an object and its *adminWithPolicy* attribute is SET, or a hierarchy, then the authorization session is a policy session (TPM_RC_AUTH_TYPE).

NOTE 3 If adminWithPolicy is CLEAR, then any type of authorization session is allowed.

2) If the entity being authorized is an NV Index, then the associated authorization session is a policy session.

NOTE 4 The only commands that are currently defined that require use of ADMIN role authorization are commands that operate on objects and NV Indices.

f) If the command requires a handle to have USER role authorization:

1) If the entity being authorized is an object and its *userWithAuth* attribute is CLEAR, then the associated authorization session is a policy session (TPM_RC_POLICY_FAIL).

NOTE 5 There is no check for a hierarchy, because a hierarchy operates as if userWithAuth is SET.

2) If the entity being authorized is an NV Index;

i) if the authorization session is a policy session;

(a) the TPMA_NV_POLICYWRITE attribute of the NV Index is SET if the command modifies the NV Index data (TPM_RC_AUTH_UNAVAILABLE);

(b) the TPMA_NV_POLICYREAD attribute of the NV Index is SET if the command reads the NV Index data (TPM_RC_AUTH_UNAVAILABLE);

ii) if the authorization is an HMAC session or a password;

(a) the TPMA_NV_AUTHWRITE attribute of the NV Index is SET if the command modifies the NV Index data (TPM_RC_AUTH_UNAVAILABLE);

(b) the TPMA_NV_AUTHREAD attribute of the NV Index is SET if the command reads the NV Index data (TPM_RC_AUTH_UNAVAILABLE).

g) If the authorization is provided by a policy session, then:

1) if *policySession→timeOut* has been set, the session shall not have expired (TPM_RC_EXPIRED);

2) if *policySession→cpHash* has been set, it shall match the *cpHash* of the command (TPM_RC_POLICY_FAIL);

3) if *policySession→commandCode* has been set, then *commandCode* of the command shall match (TPM_RC_POLICY_CC);

4) *policySession→policyDigest* shall match the *authPolicy* associated with the handle (TPM_RC_POLICY_FAIL);

5) if *policySession→pcrUpdateCounter* has been set, then it shall match the value of *pcrUpdateCounter* (TPM_RC_PCR_CHANGED);

6) if *policySession→commandLocality* has been set, it shall match the locality of the command (TPM_RC_LOCALITY),

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

7) if *policySession→cpHash* contains a template, and the command is TPM2_Create(), TPM2_CreatePrimary(), or TPM2_CreateLoaded(), then the *inPublic* parmeter matches the contents of *policySession→cpHash;* and

8) if the policy requires that an authValue be provided in order to satisfy the policy, then *session.hmac* is not an Empty Buffer.

h) If the authorization uses an HMAC, then the HMAC is properly constructed using the *authValue* associated with the handle and/or the session secret (TPM_RC_AUTH_FAIL or TPM_RC_BAD_AUTH).

NOTE 6 A policy session may require proof of knowledge of the authValue of the object being authorized.

i) If the authorization uses a password, then the password matches the *authValue* associated with the handle (TPM_RC_AUTH_FAIL or TPM_RC_BAD_AUTH).

If the TPM returns an error other than TPM_RC_AUTH_FAIL then the TPM shall not alter any TPM state. If the TPM return TPM_RC_AUTH_FAIL, then the TPM shall not alter any TPM state other than *lockoutCount*.

NOTE 7 The TPM may decrease failedTries regardless of any other processing performed by the TPM. That is, the TPM may exit Lockout mode, regardless of the return code.

5.7 Parameter Decryption

If an authorization session has the TPMA_SESSION.*decrypt* attribute SET, and the command does not allow a command parameter to be encrypted, then the TPM will return TPM_RC_ATTRIBUTES. Otherwise, the TPM will decrypt the parameter using the values associated with the session before parsing parameters.

NOTE The size of the parameter to be encrypted can be zero.

5.8 Parameter Unmarshaling

5.8.1 Introduction

The detailed actions for each command assume that the input parameters of the command have been unmarshaled into a command-specific structure with the structure defined by the command schematic. Additionally, a response-specific output structure is assumed which will receive the values produced by the detailed actions.

NOTE An implementation is not required to process parameters in this manner or to separate the parameter parsing from the command actions. This method was chosen for the specification so that the normative behavior described by the detailed actions would be clear and unencumbered.

Unmarshaling is the process of processing the parameters in the input buffer and preparing the parameters for use by the command-specific action code. No data movement need take place but it is required that the TPM validate that the parameters meet the requirements of the expected data type as defined in TPM 2.0 Part 2.

 Copyright © TCG 2006-2020

5.8.2 Unmarshaling Errors

When an error is encountered while unmarshaling a command parameter, an error response code is returned and no command processing occurs. A table defining a data type may have response codes embedded in the table to indicate the error returned when the input value does not match the parameters of the table.

NOTE In the reference implementation, a parameter number is added to the response code so that the offending parameter can be isolated. This is optional.

In many cases, the table contains no specific response code value and the return code will be determined as defined in Table 3.

Table 3 — Unmarshaling Errors

Response Code	Meaning
TPM_RC_ASYMMETRIC	a parameter that should be an asymmetric algorithm selection does not have a value that is supported by the TPM
TPM_RC_BAD_TAG	a parameter that should be a command tag selection has a value that is not supported by the TPM
TPM_RC_COMMAND_CODE	a parameter that should be a command code does not have a value that is supported by the TPM
TPM_RC_HASH	a parameter that should be a hash algorithm selection does not have a value that is supported by the TPM
TPM_RC_INSUFFICIENT	the input buffer did not contain enough octets to allow unmarshaling of the expected data type;
TPM_RC_KDF	a parameter that should be a key derivation scheme (KDF) selection does not have a value that is supported by the TPM
TPM_RC_KEY_SIZE	a parameter that is a key size has a value that is not supported by the TPM
TPM_RC_MODE	a parameter that should be a symmetric encryption mode selection does not have a value that is supported by the TPM
TPM_RC_RESERVED	a non-zero value was found in a reserved field of an attribute structure (TPMA_)
TPM_RC_SCHEME	a parameter that should be signing or encryption scheme selection does not have a value that is supported by the TPM
TPM_RC_SIZE	the value of a size parameter is larger or smaller than allowed
TPM_RC_SYMMETRIC	a parameter that should be a symmetric algorithm selection does not have a value that is supported by the TPM
TPM_RC_TAG	a parameter that should be a structure tag has a value that is not supported by the TPM
TPM_RC_TYPE	The type parameter of a TPMT_PUBLIC or TPMT_SENSITIVE has a value that is not supported by the TPM
TPM_RC_VALUE	a parameter does not have one of its allowed values

In some commands, a parameter may not be used because of various options of that command. However, the unmarshaling code is required to validate that all parameters have values that are allowed by the TPM 2.0 Part 2 definition of the parameter type even if that parameter is not used in the command actions.

 Copyright © TCG 2006-2020

5.9 Command Post Processing

When the code that implements the detailed actions of the command completes, it returns a response code. If that code is not TPM_RC_SUCCESS, the post processing code will not update any session or audit data and will return a 10-octet response packet.

If the command completes successfully, the tag of the command determines if any authorization sessions will be in the response. If so, the TPM will encrypt the first parameter of the response if indicated by the authorization attributes. The TPM will then generate a new nonce value for each session and, if appropriate, generate an HMAC.

If authorization HMAC computations are performed on the response, the HMAC keys used in the response will be the same as the HMAC keys used in processing the HMAC in the command.

NOTE 1 This primarily affects authorizations associated with a first write to an NV Index using a bound session. The computation of the HMAC in the response is performed as if the Name of the Index did not change as a consequence of the command actions. The session binding to the NV Index will not persist to any subsequent command.

NOTE 2 The authorization attributes were validated during the session area validation to ensure that only one session was used for parameter encryption of the response and that the command allowed encryption in the response.

NOTE 3 No session nonce value is used for a password authorization but the session data is present.

Additionally, if the command is being audited by Command Audit, the audit digest is updated with the *cpHash* of the command and *rpHash* of the response.

 Copyright © TCG 2006-2020

6 Response Values

6.1 Tag

When a command completes successfully, the *tag* parameter in the response shall have the same value as the *tag* parameter in the command (TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS). When a command fails (the responseCode is not TPM_RC_SUCCESS), then the *tag* parameter in the response shall be TPM_ST_NO_SESSIONS.

A special case exists when the command *tag* parameter is not an allowed value (TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS). For this case, it is assumed that the system software is attempting to send a command formatted for a TPM 1.2 but the TPM is not capable of executing TPM 1.2 commands. So that the TPM 1.2 compatible software will have a recognizable response, the TPM sets *tag* to TPM_ST_RSP_COMMAND, *responseSize* to 00 00 00 $0A_{16}$ and *responseCode* to TPM_RC_BAD_TAG. This is the same response as the TPM 1.2 fatal error for TPM_BADTAG.

6.2 Response Codes

The normal response for any command is TPM_RC_SUCCESS. Any other value indicates that the command did not complete and the state of the TPM is unchanged. An exception to this general rule is that the logic associated with dictionary attack protection is allowed to be modified when an authorization failure occurs.

Commands have response codes that are specific to that command, and those response codes are enumerated in the detailed actions of each command. The codes associated with the unmarshaling of parameters are documented Table 3. Another set of response code values are not command specific and indicate a problem that is not specific to the command. That is, if the indicated problem is remedied, the same command could be resubmitted and may complete normally.

The response codes that are not command specific are listed and described in

Table 4.

The reference code for the command actions may have code that generates specific response codes associated with a specific check but the listing of responses may not have that response code listed.

 Copyright © TCG 2006-2020

Table 4 — Command-Independent Response Codes

Response Code	Meaning
TPM_RC_CANCELED	This response code may be returned by a TPM that supports command cancel. When the TPM receives an indication that the current command should be cancelled, the TPM may complete the command or return this code. If this code is returned, then the TPM state is not changed and the same command may be retried.
TPM_RC_CONTEXT_GAP	This response code can be returned for commands that manage session contexts. It indicates that the gap between the lowest numbered active session and the highest numbered session is at the limits of the session tracking logic. The remedy is to load the session context with the lowest number so that its tracking number can be updated.
TPM_RC_LOCKOUT	This response indicates that authorizations for objects subject to DA protection are not allowed at this time because the TPM is in DA lockout mode. The remedy is to wait or to exeucte TPM2_DictionaryAttackLockoutReset().
TPM_RC_MEMORY	A TPM may use a common pool of memory for objects, sessions, and other purposes. When the TPM does not have enough memory available to perform the actions of the command, it may return TPM_RC_MEMORY. This indicates that the TPM resource manager may flush either sessions or objects in order to make memory available for the command execution. A TPM may choose to return TPM_RC_OBJECT_MEMORY or TPM_RC_SESSION_MEMORY if it needs contexts of a particular type to be flushed.
TPM_RC_NV_RATE	This response code indicates that the TPM is rate-limiting writes to the NV memory in order to prevent wearout. This response is possible for any command that explicity writes to NV or commands that incidentally use NV such as a command that uses authorization session that may need to update the dictionary attack logic.
TPM_RC_NV_UNAVAILABLE	This response code is similar to TPM_RC_NV_RATE but indicates that access to NV memory is currently not available and the command is not allowed to proceed until it is. This would occur in a system where the NV memory used by the TPM is not exclusive to the TPM and is a shared system resource.
TPM_RC_OBJECT_HANDLES	This response code indicates that the TPM has exhausted its handle space and no new objects can be loaded unless the TPM is rebooted. This does not occur in the reference implementation because of the way that object handles are allocated. However, other implementations are allowed to assign each object a unique handle each time the object is loaded. A TPM using this implementation would be able to load 2^{24} objects before the object space is exhausted.
TPM_RC_OBJECT_MEMORY	This response code can be returned by any command that causes the TPM to need an object 'slot'. The most common case where this might be returned is when an object is loaded (TPM2_Load, TPM2_CreatePrimary(), or TPM2_ContextLoad()). However, the TPM implementation is allowed to use object slots for other reasons. In the reference implementation, the TPM copies a referenced persistent object into RAM for the duration of the commannd. If all the slots are previously occupied, the TPM may return this value. A TPM is allowed to use object slots for other purposes and return this value. The remedy when this response is returned is for the TPM resource manager to flush a transient object.
TPM_RC_REFERENCE_Hx	This response code indicates that a handle in the handle area of the command is not associated with a loaded object. The value of 'x' is in the range 0 to 6 with a value of 0 indicating the 1st handle and 6 representing the 7th. Upper values are provided for future use. The TPM resource manager needs to find the correct object and load it. It may then adjust the handle and retry the command. NOTE Usually, this error indicates that the TPM resource manager has a corrupted database.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

Response Code	**Meaning**
TPM_RC_REFERENCE_Sx	This response code indicates that a handle in the session area of the command is not associated with a loaded session. The value of 'x' is in the range 0 to 6 with a value of 0 indicating the 1st session handle and 6 representing the 7th. Upper values are provided for future use. The TPM resource manager needs to find the correct session and load it. It may then retry the command. NOTE Usually, this error indicates that the TPM resource manager has a corrupted database.
TPM_RC_RETRY	the TPM was not able to start the command
TPM_RC_SESSION_HANDLES	This response code indicates that the TPM does not have a handle to assign to a new session. This respose is only returned by TPM2_StartAuthSession(). It is listed here because the command is not in error and the TPM resource manager can remedy the situation by flushing a session (TPM2_FlushContext().
TPM_RC_SESSION_MEMORY	This response code can be returned by any command that causes the TPM to need a session 'slot'. The most common case where this might be returned is when a session is loaded (TPM2_StartAuthSession() or TPM2_ContextLoad()). However, the TPM implementation is allowed to use object slots for other purposes. The remedy when this response is returned is for the TPM resource manager to flush a transient object.
TPM_RC_SUCCESS	Normal completion for any command. If the responseCode is TPM_RC_SUCCESS, then the rest of the response has the format indicated in the response schematic. Otherwise, the response is a 10 octet value indicating an error.
TPM_RC_TESTING	This response code indicates that the TPM is performing tests and cannot respond to the request at this time. The command may be retried.
TPM_RC_YIELDED	the TPM has suspended operation on the command; forward progress was made and the command may be retried. See TPM 2.0 Part 1, “Multi-tasking.” NOTE This cannot occur on the reference implementation.

 Copyright © TCG 2006-2020

7 Implementation Dependent

The actions code for each command makes assumptions about the behavior of various sub-systems. There are many possible implementations of the subsystems that would achieve equivalent results. The actions code is not written to anticipate all possible implementations of the sub-systems. Therefore, it is the responsibility of the implementer to ensure that the necessary changes are made to the actions code when the sub-system behavior changes.

 Copyright © TCG 2006-2020

8 Detailed Actions Assumptions

8.1 Introduction

The C code in the Detailed Actions for each command is written with a set of assumptions about the processing performed before the action code is called and the processing that will be done after the action code completes.

8.2 Pre-processing

Before calling the command actions code, the following actions have occurred.

- Verification that the handles in the handle area reference entities that are resident on the TPM.
- NOTE If a handle is in the parameter portion of the command, the associated entity does not have to be loaded, but the handle is required to be the correct type.
- If use of a handle requires authorization, the Password, HMAC, or Policy session associated with the handle has been verified.
- If a command parameter was encrypted using parameter encryption, it was decrypted before being unmarshaled.
- If the command uses handles or parameters, the calling stack contains a pointer to a data structure (*in*) that holds the unmarshaled values for the handles and command parameters. If the response has handles or parameters, the calling stack contains a pointer to a data structure (*out*) to hold the handles and response parameters generated by the command.
- All parameters of the *in* structure have been validated and meet the requirements of the parameter type as defined in TPM 2.0 Part 2.
- Space set aside for the out structure is sufficient to hold the largest *out* structure that could be produced by the command

8.3 Post Processing

When the function implementing the command actions completes,

- response parameters that require parameter encryption will be encrypted after the command actions complete;
- audit and session contexts will be updated if the command response is TPM_RC_SUCCESS; and
- the command header and command response parameters will be marshaled to the response buffer.

 Copyright © TCG 2006-2020

9 Start-up

9.1 Introduction

This clause contains the commands used to manage the startup and restart state of a TPM.

9.2 _TPM_Init

9.2.1 General Description

_TPM_Init initializes a TPM.

Initialization actions include testing code required to execute the next expected command. If the TPM is in FUM, the next expected command is TPM2_FieldUpgradeData(); otherwise, the next expected command is TPM2_Startup().

NOTE 1 If the TPM performs self-tests after receiving _TPM_Init() and the TPM enters Failure mode before receiving TPM2_Startup() or TPM2_FieldUpgradeData(), then the TPM may be able to accept TPM2_GetTestResult() or TPM2_GetCapability().

The means of signaling _TPM_Init shall be defined in the platform-specific specifications that define the physical interface to the TPM. The platform shall send this indication whenever the platform starts its boot process and only when the platform starts its boot process.

There shall be no software method of generating this indication that does not also reset the platform and begin execution of the CRTM.

NOTE 2 In the reference implementation, this signal causes an internal flag (*s_initialized*) to be CLEAR. While this flag is CLEAR, the TPM will only accept the next expected command described above.

 Copyright © TCG 2006-2020

9.2.2 Detailed Actions

```
1  #include "Tpm.h"
2  #include "_TPM_Init_fp.h"
3   // This function is used to process a _TPM_Init indication.
4  LIB_EXPORT void
5  _TPM_Init(
6      void
7      )
8  {
9      g_powerWasLost = g_powerWasLost | _plat__WasPowerLost();
10
11 #if SIMULATION && DEBUG
12     // If power was lost and this was a simulation, put canary in RAM used by NV
13     // so that uninitialized memory can be detected more easily
14     if(g_powerWasLost)
15     {
16         memset(&gc, 0xbb, sizeof(gc));
17         memset(&gr, 0xbb, sizeof(gr));
18         memset(&gp, 0xbb, sizeof(gp));
19         memset(&go, 0xbb, sizeof(go));
20     }
21 #endif
22
23 #if SIMULATION
24     // Clear the flag that forces failure on self-test
25     g_forceFailureMode = FALSE;
26 #endif
27
28     // Disable the tick processing
29     _plat__ACT_EnableTicks(FALSE);
30
31     // Set initialization state
32     TPMInit();
33
34     // Set g_DRTMHandle as unassigned
35     g_DRTMHandle = TPM_RH_UNASSIGNED;
36
37     // No H-CRTM, yet.
38     g_DrtmPreStartup = FALSE;
39
40     // Initialize the NvEnvironment.
41     g_nvOk = NvPowerOn();
42
43     // Initialize cryptographic functions
44     g_inFailureMode = (CryptInit() == FALSE);
45     if(!g_inFailureMode)
46     {
47         // Load the persistent data
48         NvReadPersistent();
49
50         // Load the orderly data (clock and DRBG state).
51         // If this is not done here, things break
52         NvRead(&go, NV_ORDERLY_DATA, sizeof(go));
53
54         // Start clock. Need to do this after NV has been restored.
55         TimePowerOn();
56     }
57     return;
58 }
```

9.3 TPM2_Startup

9.3.1 General Description

TPM2_Startup() is always preceded by _TPM_Init, which is the physical indication that TPM initialization is necessary because of a system-wide reset. TPM2_Startup() is only valid after _TPM_Init. Additional TPM2_Startup() commands are not allowed after it has completed successfully. If a TPM requires TPM2_Startup() and another command is received, or if the TPM receives TPM2_Startup() when it is not required, the TPM shall return TPM_RC_INITIALIZE.

NOTE 1 See 9.2.1 for other command options for a TPM supporting field upgrade mode.

NOTE 2 _TPM_Hash_Start, _TPM_Hash_Data, and _TPM_Hash_End are not commands and a platform-specific specification may allow these indications between _TPM_Init and TPM2_Startup().

If in Failure mode, the TPM shall accept TPM2_GetTestResult() and TPM2_GetCapability() even if TPM2_Startup() is not completed successfully or processed at all.

A platform-specific specification may restrict the localities at which TPM2_Startup() may be received.

A Shutdown/Startup sequence determines the way in which the TPM will operate in response to TPM2_Startup(). The three sequences are:

1) TPM Reset – This is a Startup(CLEAR) preceded by either Shutdown(CLEAR) or no TPM2_Shutdown(). On TPM Reset, all variables go back to their default initialization state.

   NOTE 3 Only those values that are specified as having a default initialization state are changed by TPM Reset. Persistent values that have no default initialization state are not changed by this command. Values such as seeds have no default initialization state and only change due to specific commands.

2) TPM Restart – This is a Startup(CLEAR) preceded by Shutdown(STATE). This preserves much of the previous state of the TPM except that PCR and the controls associated with the Platform hierarchy are all returned to their default initialization state;

3) TPM Resume – This is a Startup(STATE) preceded by Shutdown(STATE). This preserves the previous state of the TPM including the static Root of Trust for Measurement (S-RTM) PCR and the platform controls other than the *phEnable*.

If a TPM receives Startup(STATE) and that was not preceded by Shutdown(STATE), the TPM shall return TPM_RC_VALUE.

If, during TPM Restart or TPM Resume, the TPM fails to restore the state saved at the last Shutdown(STATE), the TPM shall enter Failure Mode and return TPM_RC_FAILURE.

On any TPM2_Startup(),

- *phEnable* shall be SET;
- all transient contexts (objects, sessions, and sequences) shall be flushed from TPM memory;

NOTE 4 See Part 1 Time for a description of the TPMS_TIME_INFO.*time* behaviour.

- use of *lockoutAuth* shall be enabled if *lockoutRecovery* is zero.

Additional actions are performed based on the Shutdown/Startup sequence.

 Copyright © TCG 2006-2020

On TPM Reset:

- *platformAuth* and *platformPolicy* shall be set to the Empty Buffer,
- For each NV Index with TPMA_NV_WRITEDEFINE CLEAR or TPMA_NV_WRITTEN CLEAR, TPMA_NV_WRITELOCKED shall be CLEAR,
- For each NV Index with TPMA_NV_ORDERLY SET, TPMA_NV_WRITTEN shall be CLEAR unless the type is TPM_NT_COUNTER,
- On a disorderly reset, advance the orderly counters,
- For each NV Index with TPMA_NV_CLEAR_STCLEAR SET, TPMA_NV_WRITTEN shall be CLEAR,
- tracking data for saved session contexts shall be set to its initial value,
- the object context sequence number is reset to zero,
- a new context encryption key shall be generated,
- TPMS_CLOCK_INFO.*restartCount* shall be reset to zero,
- TPMS_CLOCK_INFO.*resetCount* shall be incremented,
- the PCR Update Counter shall be clear to zero,

  NOTE 5 Because the PCR update counter may be incremented when a PCR is reset, the PCR resets performed as part of this command can result in the PCR update counter being non-zero at the end of this command.

- *phEnableNV, shEnable* and *ehEnable* shall be SET, and
- PCR in all banks are reset to their default initial conditions as determined by the relevant platform-specific specification and the H-CRTM state (for exceptions, see TPM 2.0 Part 1*, H-CRTM before TPM2_Startup() and TPM2_Startup without H-CRTM*),
- For each ACT the timeout is reset to zero, the *signaled* attribute is set to CLEAR (if *preserveSignaled* is CLEAR), and the *authPolicy* is set to the Empty Buffer and its hashAlg is set to TPM_ALG_NULL.

  NOTE 6 PCR may be initialized any time between _TPM_Init and the end of TPM2_Startup(). PCR that are preserved by TPM Resume will need to be restored during TPM2_Startup().

  NOTE 7 See "Initializing PCR" in TPM 2.0 Part 1 for a description of the default initial conditions for a PCR.

 Copyright © TCG 2006-2020

On TPM Restart:

- TPMS_CLOCK_INFO.*restartCount* shall be incremented,
- *phEnableNV, shEnable* and *ehEnable* shall be SET,
- *platformAuth* and *platformPolicy* shall be set to the Empty Buffer,
- For each NV index with TPMA_NV_WRITEDEFINE CLEAR or TPMA_NV_WRITTEN CLEAR, TPMA_NV_WRITELOCKED shall be CLEAR,
- For each NV index with TPMA_NV_CLEAR_STCLEAR SET, TPMA_NV_WRITTEN shall be CLEAR, and
- PCR in all banks are reset to their default initial conditions.
- If an H-CRTM Event Sequence is active, extend the PCR designated by the platform-specific specification.
- For each ACT the timeout is reset to zero, the *signaled* attribute is set to CLEAR (if *preserveSignaled* is CLEAR), and the *authPolicy* is set to the Empty Buffer and its hashAlg is set to TPM_ALG_NULL.

On TPM Resume:

- the H-CRTM startup method is the same for this TPM2_Startup() as for the previous TPM2_Startup(); (TPM_RC_LOCALITY)
- TPMS_CLOCK_INFO.*restartCount* shall be incremented; and
- PCR that are specified in a platform-specific specification to be preserved on TPM Resume are restored to their saved state and other PCR are set to their initial value as determined by a platform-specific specification. For constraints, see TPM 2.0 Part 1, *H-CRTM before TPM2_Startup() and TPM2_Startup without H-CRTM.*
- The ACT timeout, the ACT *signaled* attribute and the ACT specific *authPolicy* values are preserved.

Other TPM state may change as required to meet the needs of the implementation.

If the *startupType* is TPM_SU_STATE and the TPM requires TPM_SU_CLEAR, then the TPM shall return TPM_RC_VALUE.

NOTE 8 The TPM will require TPM_SU_CLEAR when no shutdown was performed or after Shutdown(CLEAR).

NOTE 9 If *startupType* is neither TPM_SU_STATE nor TPM_SU_CLEAR, then the unmarshaling code returns TPM_RC_VALUE.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

9.3.2 Command and Response

Table 5 — TPM2_Startup Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Startup {NV}
TPM_SU	startupType	TPM_SU_CLEAR or TPM_SU_STATE

Table 6 — TPM2_Startup Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

9.3.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "Startup_fp.h"
3  #if CC_Startup  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_LOCALITY	a Startup(STATE) does not have the same H-CRTM state as the previous Startup() or the locality of the startup is not 0 pr 3
TPM_RC_NV_UNINITIALIZED	the saved state cannot be recovered and a Startup(CLEAR) is required.
TPM_RC_VALUE	start up type is not compatible with previous shutdown sequence

```
4  TPM_RC
5  TPM2_Startup(
6      Startup_In      *in              // IN: input parameter list
7      )
8  {
9      STARTUP_TYPE         startup;
10     BYTE                 locality = _plat__LocalityGet();
11     BOOL                 OK = TRUE;
12 //
13     // The command needs NV update.
14     RETURN_IF_NV_IS_NOT_AVAILABLE;
15
16     // Get the flags for the current startup locality and the H-CRTM.
17     // Rather than generalizing the locality setting, this code takes advantage
18     // of the fact that the PC Client specification only allows Startup()
19     // from locality 0 and 3. To generalize this probably would require a
20     // redo of the NV space and since this is a feature that is hardly ever used
21     // outside of the PC Client, this code just support the PC Client needs.
22
23 // Input Validation
24     // Check that the locality is a supported value
25     if(locality != 0 && locality != 3)
26         return TPM_RC_LOCALITY;
27     // If there was a H-CRTM, then treat the locality as being 3
28     // regardless of what the Startup() was. This is done to preserve the
29     // H-CRTM PCR so that they don't get overwritten with the normal
30     // PCR startup initialization. This basically means that g_StartupLocality3
31     // and g_DrtmPreStartup can't both be SET at the same time.
32     if(g_DrtmPreStartup)
33         locality = 0;
34     g_StartupLocality3 = (locality == 3);
35
36 #if USE_DA_USED
37     // If there was no orderly shutdown, then their might have been a write to
38     // failedTries that didn't get recorded but only if g_daUsed was SET in the
39     // shutdown state
40     g_daUsed = (gp.orderlyState == SU_DA_USED_VALUE);
41     if(g_daUsed)
42         gp.orderlyState = SU_NONE_VALUE;
43 #endif
44
45     g_prevOrderlyState = gp.orderlyState;
46
47     // If there was a proper shutdown, then the startup modifiers are in the
48     // orderlyState. Turn them off in the copy.
49     if(IS_ORDERLY(g_prevOrderlyState))
50         g_prevOrderlyState &=  ~(PRE_STARTUP_FLAG | STARTUP_LOCALITY_3);
```


Level 00 Revision 01.59 **Copyright © TCG 2006-2020** November 8, 2019

```
51      // If this is a Resume,
52      if(in->startupType == TPM_SU_STATE)
53      {
54          // then there must have been a prior TPM2_ShutdownState(STATE)
55          if(g_prevOrderlyState != TPM_SU_STATE)
56              return TPM_RCS_VALUE + RC_Startup_startupType;
57          // and the part of NV used for state save must have been recovered
58          // correctly.
59          // NOTE: if this fails, then the caller will need to do Startup(CLEAR). The
60          // code for Startup(Clear) cannot fail if the NV can't be read correctly
61          // because that would prevent the TPM from ever getting unstuck.
62          if(g_nvOk == FALSE)
63              return TPM_RC_NV_UNINITIALIZED;
64          // For Resume, the H-CRTM has to be the same as the previous boot
65          if(g_DrtmPreStartup != ((gp.orderlyState & PRE_STARTUP_FLAG) != 0))
66              return TPM_RCS_VALUE + RC_Startup_startupType;
67          if(g_StartupLocality3 != ((gp.orderlyState & STARTUP_LOCALITY_3) != 0))
68              return TPM_RC_LOCALITY;
69      }
70      // Clean up the gp state
71      gp.orderlyState = g_prevOrderlyState;
72
73  // Internal Date Update
74      if((gp.orderlyState == TPM_SU_STATE) && (g_nvOk == TRUE))
75      {
76          // Always read the data that is only cleared on a Reset because this is not
77          // a reset
78          NvRead(&gr, NV_STATE_RESET_DATA, sizeof(gr));
79          if(in->startupType == TPM_SU_STATE)
80          {
81              // If this is a startup STATE (a Resume) need to read the data
82              // that is cleared on a startup CLEAR because this is not a Reset
83              // or Restart.
84              NvRead(&gc, NV_STATE_CLEAR_DATA, sizeof(gc));
85              startup = SU_RESUME;
86          }
87          else
88              startup = SU_RESTART;
89      }
90      else
91          // Will do a TPM reset if Shutdown(CLEAR) and Startup(CLEAR) or no shutdown
92          // or there was a failure reading the NV data.
93          startup = SU_RESET;
94      // Startup for cryptographic library. Don't do this until after the orderly
95      // state has been read in from NV.
96      OK = OK && CryptStartup(startup);
97
98      // When the cryptographic library has been started, indicate that a TPM2_Startup
99      // command has been received.
100     OK = OK && TPMRegisterStartup();
101
102     // Read the platform unique value that is used as VENDOR_PERMANENT
103     // authorization value
104     g_platformUniqueDetails.t.size
105         = (UINT16)_plat__GetUnique(1, sizeof(g_platformUniqueDetails.t.buffer),
106                                    g_platformUniqueDetails.t.buffer);
107
108 // Start up subsystems
109     // Start set the safe flag
110     OK = OK && TimeStartup(startup);
111
112     // Start dictionary attack subsystem
113     OK = OK && DAStartup(startup);
114
115     // Enable hierarchies
116     OK = OK && HierarchyStartup(startup);
```

```
117
118     // Restore/Initialize PCR
119     OK = OK && PCRStartup(startup, locality);
120
121     // Restore/Initialize command audit information
122     OK = OK && CommandAuditStartup(startup);
123
124     // Restore the ACT
125     OK = OK && ActStartup(startup);
126
127 //// The following code was moved from Time.c where it made no sense
128     if(OK)
129     {
130         switch(startup)
131         {
132             case SU_RESUME:
133                 // Resume sequence
134                 gr.restartCount++;
135                 break;
136             case SU_RESTART:
137                 // Hibernate sequence
138                 gr.clearCount++;
139                 gr.restartCount++;
140                 break;
141             default:
142                 // Reset object context ID to 0
143                 gr.objectContextID = 0;
144                 // Reset clearCount to 0
145                 gr.clearCount = 0;
146
147                 // Reset sequence
148                 // Increase resetCount
149                 gp.resetCount++;
150
151                 // Write resetCount to NV
152                 NV_SYNC_PERSISTENT(resetCount);
153
154                 gp.totalResetCount++;
155                 // We do not expect the total reset counter overflow during the life
156                 // time of TPM.  if it ever happens, TPM will be put to failure mode
157                 // and there is no way to recover it.
158                 // The reason that there is no recovery is that we don't increment
159                 // the NV totalResetCount when incrementing would make it 0. When the
160                 // TPM starts up again, the old value of totalResetCount will be read
161                 // and we will get right back to here with the increment failing.
162                 if(gp.totalResetCount == 0)
163                     FAIL(FATAL_ERROR_INTERNAL);
164
165                 // Write total reset counter to NV
166                 NV_SYNC_PERSISTENT(totalResetCount);
167
168                 // Reset restartCount
169                 gr.restartCount = 0;
170
171                 break;
172         }
173     }
174     // Initialize session table
175     OK = OK && SessionStartup(startup);
176
177     // Initialize object table
178     OK = OK && ObjectStartup();
179
180     // Initialize index/evict data.  This function clears read/write locks
181     // in NV index
182     OK = OK && NvEntityStartup(startup);
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

```
183
184     // Initialize the orderly shut down flag for this cycle to SU_NONE_VALUE.
185     gp.orderlyState = SU_NONE_VALUE;
186
187     OK = OK && NV_SYNC_PERSISTENT(orderlyState);
188
189     // This can be reset after the first completion of a TPM2_Startup() after
190     // a power loss. It can probably be reset earlier but this is an OK place.
191     if(OK)
192         g_powerWasLost = FALSE;
193
194     return (OK) ? TPM_RC_SUCCESS : TPM_RC_FAILURE;
195 }
196 #endif // CC_Startup
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

9.4 TPM2_Shutdown

9.4.1 General Description

This command is used to prepare the TPM for a power cycle. The *shutdownType* parameter indicates how the subsequent TPM2_Startup() will be processed.

For a *shutdownType* of any type, the volatile portion of Clock is saved to NV memory and the orderly shutdown indication is SET. NV Indexes with the TPMA_NV_ORDERLY attribute will be updated.

For a *shutdownType* of TPM_SU_STATE, the following additional items are saved:

- tracking information for saved session contexts;
- the session context counter;
- PCR that are designated as being preserved by TPM2_Shutdown(TPM_SU_STATE);
- the PCR Update Counter;
- flags associated with supporting the TPMA_NV_WRITESTCLEAR and TPMA_NV_READSTCLEAR attributes;
- the counter value and authPolicy for each ACT; and

  NOTE If a counter has not been updated since the last TPM2_Startup(), then the saved value will be one half of the current counter value.

- the command audit digest and count.

The following items shall not be saved and will not be in TPM memory after the next TPM2_Startup:

- TPM-memory-resident session contexts;
- TPM-memory-resident transient objects; or
- TPM-memory-resident hash contexts created by TPM2_HashSequenceStart().

Some values may be either derived from other values or saved to NV memory.

This command saves TPM state but does not change the state other than the internal indication that the context has been saved. The TPM shall continue to accept commands. If a subsequent command changes TPM state saved by this command, then the effect of this command is nullified. The TPM MAY nullify this command for any subsequent command rather than check whether the command changed state saved by this command. If this command is nullified. and if no TPM2_Shutdown() occurs before the next TPM2_Startup(), then the next TPM2_Startup() shall be TPM2_Startup(CLEAR).

 Copyright © TCG 2006-2020

9.4.2 Command and Response

Table 7 — TPM2_Shutdown Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Shutdown {NV}
TPM_SU	shutdownType	TPM_SU_CLEAR or TPM_SU_STATE

Table 8 — TPM2_Shutdown Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

9.4.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "Shutdown_fp.h"
3  #if CC_Shutdown  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_TYPE	if PCR bank has been re-configured, a CLEAR StateSave() is required

```
4  TPM_RC
5  TPM2_Shutdown(
6      Shutdown_In     *in              // IN: input parameter list
7      )
8  {
9      // The command needs NV update.  Check if NV is available.
10     // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
11     // this point
12     RETURN_IF_NV_IS_NOT_AVAILABLE;
13
14 // Input Validation
15
16     // If PCR bank has been reconfigured, a CLEAR state save is required
17     if(g_pcrReConfig && in->shutdownType == TPM_SU_STATE)
18         return TPM_RCS_TYPE + RC_Shutdown_shutdownType;
19
20 // Internal Data Update
21
22     gp.orderlyState = in->shutdownType;
23
24     // PCR private date state save
25     PCRStateSave(in->shutdownType);
26
27     // Save the ACT state
28     ActShutdown(in->shutdownType);
29
30     // Save RAM backed NV index data
31     NvUpdateIndexOrderlyData();
32
33 #if ACCUMULATE_SELF_HEAL_TIMER
34     // Save the current time value
35     go.time = g_time;
36 #endif
37
38     // Save all orderly data
39     NvWrite(NV_ORDERLY_DATA, sizeof(ORDERLY_DATA), &go);
40
41     if(in->shutdownType == TPM_SU_STATE)
42     {
43         // Save STATE_RESET and STATE_CLEAR data
44         NvWrite(NV_STATE_CLEAR_DATA, sizeof(STATE_CLEAR_DATA), &gc);
45         NvWrite(NV_STATE_RESET_DATA, sizeof(STATE_RESET_DATA), &gr);
46
47         // Save the startup flags for resume
48         if(g_DrtmPreStartup)
49             gp.orderlyState = TPM_SU_STATE | PRE_STARTUP_FLAG;
50         else if(g_StartupLocality3)
51             gp.orderlyState = TPM_SU_STATE | STARTUP_LOCALITY_3;
52     }
53     // only two shutdown options.
54     else if(in->shutdownType != TPM_SU_CLEAR)
55         return TPM_RCS_VALUE + RC_Shutdown_shutdownType;
```

 Copyright © TCG 2006-2020

```
56
57      NV_SYNC_PERSISTENT(orderlyState);
58
59      return TPM_RC_SUCCESS;
60  }
61  #endif // CC_Shutdown
```

 Copyright © TCG 2006-2020

10 Testing

10.1 Introduction

Compliance to standards for hardware security modules may require that the TPM test its functions before the results that depend on those functions may be returned. The TPM may perform operations using testable functions before those functions have been tested as long as the TPM returns no value that depends on the correctness of the testable function.

EXAMPLE TPM2_PCR_Extend() may be executed before the hash algorithms have been tested. However, until the hash algorithms have been tested, the contents of a PCR may not be used in any command if that command may result in a value being returned to the TPM user. This means that TPM2_PCR_Read() or TPM2_PolicyPCR() could not complete until the hashes have been checked but other TPM2_PCR_Extend() commands may be executed even though the operation uses previous PCR values.

If a command is received that requires return of a value that depends on untested functions, the TPM shall test the required functions before completing the command.

Once the TPM has received TPM2_SelfTest() and before completion of all tests, the TPM is required to return TPM_RC_TESTING for any command that uses a function that requires a test.

If a self-test fails at any time, the TPM will enter Failure mode. While in Failure mode, the TPM will return TPM_RC_FAILURE for any command other than TPM2_GetTestResult() and TPM2_GetCapability(). The TPM will remain in Failure mode until the next _TPM_Init.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

10.2 TPM2_SelfTest

10.2.1 General Description

This command causes the TPM to perform a test of its capabilities. If the *fullTest* is YES, the TPM will test all functions. If *fullTest* = NO, the TPM will only test those functions that have not previously been tested.

If any tests are required, the TPM shall either

- return TPM_RC_TESTING and begin self-test of the required functions, or

  NOTE 1 If *fullTest* is NO, and all functions have been tested, the TPM shall return TPM_RC_SUCCESS.

- perform the tests and return the test result when complete. On failure, the TPM shall return TPM_RC_FAILURE.

If the TPM uses option a), the TPM shall return TPM_RC_TESTING for any command that requires use of a testable function, even if the functions required for completion of the command have already been tested.

NOTE 2 This command may cause the TPM to continue processing after it has returned the response. So that software can be notified of the completion of the testing, the interface may include controls that would allow the TPM to generate an interrupt when the “background” processing is complete. This would be in addition to the interrupt that may be available for signaling normal command completion. It is not necessary that there be two interrupts, but the interface should provide a way to indicate the nature of the interrupt (normal command or deferred command).

NOTE 3 The PC Client platform specific TPM, in response to *fullTest* YES, will not return TPM_RC_TESTING. It will block until all tests are complete.

 Copyright © TCG 2006-2020

10.2.2 Command and Response

Table 9 — TPM2_SelfTest Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_SelfTest {NV}
TPMI_YES_NO	fullTest	YES if full test to be performed NO if only test of untested functions required

Table 10 — TPM2_SelfTest Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

10.2.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "SelfTest_fp.h"
3  #if CC_SelfTest  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_CANCELED	the command was canceled (some incremental process may have been made)
TPM_RC_TESTING	self test in process

```
4  TPM_RC
5  TPM2_SelfTest(
6      SelfTest_In     *in             // IN: input parameter list
7      )
8  {
9  // Command Output
10
11     // Call self test function in crypt module
12     return CryptSelfTest(in->fullTest);
13 }
14 #endif // CC_SelfTest
```

 Copyright © TCG 2006-2020

10.3 TPM2_IncrementalSelfTest

10.3.1 General Description

This command causes the TPM to perform a test of the selected algorithms.

NOTE 1 The *toTest* list indicates the algorithms that software would like the TPM to test in anticipation of future use. This allows tests to be done so that a future commands will not be delayed due to testing.

The implementation may treat algorithms on the *toTest* list as either 'test each completely' or 'test this combination.'

EXAMPLE If the *toTest* list includes AES and CTR mode, it may be interpreted as a request to test only AES in CTR mode. Alternatively, it may be interpreted as a request to test AES in all modes and CTR mode for all symmetric algorithms.

If *toTest* contains an algorithm that has already been tested, it will not be tested again.

NOTE 2 The only way to force retesting of an algorithm is with TPM2_SelfTest(*fullTest* = YES).

The TPM will return in *toDoList* a list of algorithms that are yet to be tested. This list is not the list of algorithms that are scheduled to be tested but the algorithms/functions that have not been tested. Only the algorithms on the *toTest* list are scheduled to be tested by this command.

NOTE 3 An algorithm remains on the *toDoList* while any part of it remains untested.

EXAMPLE A symmetric algorithm remains untested until it is tested with all its modes.

Making *toTest* an empty list allows the determination of the algorithms that remain untested without triggering any testing.

If *toTest* is not an empty list, the TPM shall return TPM_RC_SUCCESS for this command and then return TPM_RC_TESTING for any subsequent command (including TPM2_IncrementalSelfTest()) until the requested testing is complete.

NOTE 4 If *toDoList* is empty, then no additional tests are required and TPM_RC_TESTING will not be returned in subsequent commands and no additional delay will occur in a command due to testing.

NOTE 5 If none of the algorithms listed in *toTest* is in the *toDoList*, then no tests will be performed.

NOTE 6 The TPM cannot return TPM_RC_TESTING for the first call to this command even when testing is not complete, because response parameters can only returned with the TPM_RC_SUCCESS return code.

If all the parameters in this command are valid, the TPM returns TPM_RC_SUCCESS and the *toDoList* (which may be empty).

NOTE 7 An implementation may perform all requested tests before returning TPM_RC_SUCCESS, or it may return TPM_RC_SUCCESS for this command and then return TPM_RC_TESTING for all subsequence commands (including TPM2_IncrementatSelfTest()) until the requested tests are complete.

 Copyright © TCG 2006-2020

10.3.2 Command and Response

Table 11 — TPM2_IncrementalSelfTest Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_IncrementalSelfTest {NV}
TPML_ALG	toTest	list of algorithms that should be tested

Table 12 — TPM2_IncrementalSelfTest Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPML_ALG	toDoList	list of algorithms that need testing

10.3.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "IncrementalSelfTest_fp.h"
3   #if CC_IncrementalSelfTest  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_CANCELED	the command was canceled (some tests may have completed)
TPM_RC_VALUE	an algorithm in the *toTest* list is not implemented

```
4   TPM_RC
5   TPM2_IncrementalSelfTest(
6       IncrementalSelfTest_In      *in,            // IN: input parameter list
7       IncrementalSelfTest_Out     *out            // OUT: output parameter list
8       )
9   {
10      TPM_RC                       result;
11  // Command Output
12
13      // Call incremental self test function in crypt module. If this function
14      // returns TPM_RC_VALUE, it means that an algorithm on the 'toTest' list is
15      // not implemented.
16      result = CryptIncrementalSelfTest(&in->toTest, &out->toDoList);
17      if(result == TPM_RC_VALUE)
18          return TPM_RCS_VALUE + RC_IncrementalSelfTest_toTest;
19      return result;
20  }
21  #endif // CC_IncrementalSelfTest
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

10.4 TPM2_GetTestResult

10.4.1 General Description

This command returns manufacturer-specific information regarding the results of a self-test and an indication of the test status.

If TPM2_SelfTest() has not been executed and a testable function has not been tested, *testResult* will be TPM_RC_NEEDS_TEST. If TPM2_SelfTest() has been received and the tests are not complete, *testResult* will be TPM_RC_TESTING.

If testing of all functions is complete without functional failures, *testResult* will be TPM_RC_SUCCESS. If any test failed, *testResult* will be TPM_RC_FAILURE.

This command will operate when the TPM is in Failure mode so that software can determine the test status of the TPM and so that diagnostic information can be obtained for use in failure analysis. If the TPM is in Failure mode, then *tag* is required to be TPM_ST_NO_SESSIONS or the TPM shall return TPM_RC_FAILURE.

NOTE The reference implementation may return a 32-bit value *s_failFunction*. This simply gives a unique value to each of the possible places where a failure could occur. It is not intended to provide a pointer to the function. __func__ is a pointer to a character string but the failure mode code can only return 32-bit values. It is expected that the manufacturer can disambiguate this value if a customer's TPM goes into failure mode.

 Copyright © TCG 2006-2020

10.4.2 Command and Response

Table 13 — TPM2_GetTestResult Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_GetTestResult

Table 14 — TPM2_GetTestResult Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_MAX_BUFFER	outData	test result data contains manufacturer-specific information
TPM_RC	testResult	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

10.4.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "GetTestResult_fp.h"
3   #if CC_GetTestResult  // Conditional expansion of this file
```

In the reference implementation, this function is only reachable if the TPM is not in failure mode meaning that all tests that have been run have completed successfully. There is not test data and the test result is TPM_RC_SUCCESS.

```
4   TPM_RC
5   TPM2_GetTestResult(
6       GetTestResult_Out   *out            // OUT: output parameter list
7       )
8   {
9   // Command Output
10
11      // Call incremental self test function in crypt module
12      out->testResult = CryptGetTestResult(&out->outData);
13
14      return TPM_RC_SUCCESS;
15  }
16  #endif // CC_GetTestResult
```

11 Session Commands

11.1 TPM2_StartAuthSession

11.1.1 General Description

This command is used to start an authorization session using alternative methods of establishing the session key (*sessionKey*). The session key is then used to derive values used for authorization and for encrypting parameters.

This command allows injection of a secret into the TPM using either asymmetric or symmetric encryption. The type of *tpmKey* determines how the value in *encryptedSalt* is encrypted. The decrypted secret value is used to compute the *sessionKey*.

NOTE 1 If *tpmKey* Is TPM_RH_NULL, then *encryptedSalt* is required to be an Empty Buffer.

The label value of "SECRET" (see "Terms and Definitions" in TPM 2.0 Part 1) is used in the recovery of the secret value.

The TPM generates the *sessionKey* from the recovered secret value.

No authorization is required for *tpmKey* or *bind*.

NOTE 2 The justification for using *tpmKey* without providing authorization is that the result of using the key is not available to the caller, except indirectly through the *sessionKey*. This does not represent a point of attack on the value of the key. If the caller attempts to use the session without knowing the *sessionKey* value, it is an authorization failure that will trigger the dictionary attack logic.

The entity referenced with the *bind* parameter contributes an authorization value to the *sessionKey* generation process.

If both *tpmKey* and *bind* are TPM_RH_NULL, then *sessionKey* is set to the Empty Buffer. If *tpmKey* is not TPM_RH_NULL, then *encryptedSalt* is used in the computation of *sessionKey*. If *bind* is not TPM_RH_NULL, the *authValue* of *bind* is used in the *sessionKey* computation.

If *symmetric* specifies a block cipher, then TPM_ALG_CFB is the only allowed value for the *mode* field in the *symmetric* parameter (TPM_RC_MODE).

This command starts an authorization session and returns the session handle along with an initial *nonceTPM* in the response.

If the TPM does not have a free slot for an authorization session, it shall return TPM_RC_SESSION_HANDLES.

If the TPM implements a "gap" scheme for assigning *contextID* values, then the TPM shall return TPM_RC_CONTEXT_GAP if creating the session would prevent recycling of old saved contexts (See "Context Management" in TPM 2.0 Part 1).

If *tpmKey* is not TPM_ALG_NULL then *encryptedSalt* shall be a TPM2B_ENCRYPTED_SECRET of the proper type for *tpmKey*. The TPM shall return TPM_RC_HANDLE if the sensitive portion of *tpmKey* is not loaded. The TPM shall return TPM_RC_VALUE if:

a) *tpmKey* references an RSA key and
   1) the size of *encryptedSalt* is not the same as the size of the public modulus of *tpmKey*,
   2) *encryptedSalt* has a value that is greater than the public modulus of *tpmKey*,
   3) *encryptedSalt* is not a properly encoded OAEP value, or
   4) the decrypted *salt* value is larger than the size of the digest produced by the *nameAlg* of *tpmKey*; or

 Copyright © TCG 2006-2020

NOTE 3 The asymScheme of the key object is ignored in this case and TPM_ALG_OAEP is used, even if asymScheme is set to TPM_ALG_NULL.

b) *tpmKey* references an ECC key and *encryptedSalt*

1) does not contain a TPMS_ECC_POINT or

2) is not a point on the curve of *tpmKey*;

NOTE 4 When ECC is used, the point multiply process produces a value (Z) that is used in a KDF to produce the final secret value. The size of the secret value is an input parameter to the KDF and the result will be set to be the size of the digest produced by the *nameAlg* of *tpmKey*.

The TPM shall return TPM_RC_KEY if *tpmkey* does not reference an asymmetric key. The TPM shall return TPM_RC_VALUE if the scheme of the key is not TPM_ALG_OAEP or TPM_ALG_NULL. The TPM shall return TPM_RC_ATTRIBUTES if tpmKey does not have the *decrypt* attribute SET.

NOTE While TPM_RC_VALUE is preferred, TPM_RC_SCHEME is acceptable.

If *bind* references a transient object, then the TPM shall return TPM_RC_HANDLE if the sensitive portion of the object is not loaded.

For all session types, this command will cause initialization of the *sessionKey* and may establish binding between the session and an object (the *bind* object). If *sessionType* is TPM_SE_POLICY or TPM_SE_TRIAL, the additional session initialization is:

- set *policySession→policyDigest* to a Zero Digest (the digest size for *policySession→policyDigest* is the size of the digest produced by *authHash*);
- authorization may be given at any locality;
- authorization may apply to any command code;
- authorization may apply to any command parameters or handles;
- the authorization has no time limit;
- an authValue is not needed when the authorization is used;
- the session is not bound;
- the session is not an audit session; and
- the time at which the policy session was created is recorded.

Additionally, if *sessionType* is TPM_SE_TRIAL, the session will not be usable for authorization but can be used to compute the *authPolicy* for an object.

NOTE 5 Although this command changes the session allocation information in the TPM, it does not invalidate a saved context. That is, TPM2_Shutdown() is not required after this command in order to re-establish the orderly state of the TPM. This is because the created context will occupy an available slot in the TPM and sessions in the TPM do not survive any TPM2_Startup(). However, if a created session is context saved, the orderly state does change.

The TPM shall return TPM_RC_SIZE if *nonceCaller* is less than 16 octets or is greater than the size of the digest produced by *authHash*.

 Copyright © TCG 2006-2020

11.1.2 Command and Response

Table 15 — TPM2_StartAuthSession Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit, decrypt, or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_StartAuthSession
TPMI_DH_OBJECT+	tpmKey	handle of a loaded decrypt key used to encrypt *salt* may be TPM_RH_NULL Auth Index: None
TPMI_DH_ENTITY+	bind	entity providing the *authValue* may be TPM_RH_NULL Auth Index: None
TPM2B_NONCE	nonceCaller	initial *nonceCaller,* sets nonceTPM size for the session shall be at least 16 octets
TPM2B_ENCRYPTED_SECRET	encryptedSalt	value encrypted according to the type of *tpmKey* If *tpmKey* is TPM_RH_NULL, this shall be the Empty Buffer.
TPM_SE	sessionType	indicates the type of the session; simple HMAC or policy (including a trial policy)
TPMT_SYM_DEF+	symmetric	the algorithm and key size for parameter encryption may select TPM_ALG_NULL
TPMI_ALG_HASH	authHash	hash algorithm to use for the session Shall be a hash algorithm supported by the TPM and not TPM_ALG_NULL

Table 16 — TPM2_StartAuthSession Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_SH_AUTH_SESSION	sessionHandle	handle for the newly created session
TPM2B_NONCE	nonceTPM	the initial nonce from the TPM, used in the computation of the *sessionKey*

 Copyright © TCG 2006-2020

11.1.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "StartAuthSession_fp.h"
3  #if CC_StartAuthSession  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*tpmKey* does not reference a decrypt key
TPM_RC_CONTEXT_GAP	the difference between the most recently created active context and the oldest active context is at the limits of the TPM
TPM_RC_HANDLE	input decrypt key handle only has public portion loaded
TPM_RC_MODE	*symmetric* specifies a block cipher but the mode is not TPM_ALG_CFB.
TPM_RC_SESSION_HANDLES	no session handle is available
TPM_RC_SESSION_MEMORY	no more slots for loading a session
TPM_RC_SIZE	nonce less than 16 octets or greater than the size of the digest produced by *authHash*
TPM_RC_VALUE	secret size does not match decrypt key type; or the recovered secret is larger than the digest size of the *nameAlg* of *tpmKey*; or, for an RSA decrypt key, if *encryptedSecret* is greater than the public modulus of *tpmKey*.

```
4  TPM_RC
5  TPM2_StartAuthSession(
6      StartAuthSession_In      *in,            // IN: input parameter buffer
7      StartAuthSession_Out     *out            // OUT: output parameter buffer
8      )
9  {
10     TPM_RC                    result = TPM_RC_SUCCESS;
11     OBJECT                   *tpmKey;                // TPM key for decrypt salt
12     TPM2B_DATA                salt;
13
14 // Input Validation
15
16     // Check input nonce size.  IT should be at least 16 bytes but not larger
17     // than the digest size of session hash.
18     if(in->nonceCaller.t.size < 16
19        || in->nonceCaller.t.size > CryptHashGetDigestSize(in->authHash))
20         return TPM_RCS_SIZE + RC_StartAuthSession_nonceCaller;
21
22     // If an decrypt key is passed in, check its validation
23     if(in->tpmKey != TPM_RH_NULL)
24     {
25         // Get pointer to loaded decrypt key
26         tpmKey = HandleToObject(in->tpmKey);
27
28         // key must be asymmetric with its sensitive area loaded. Since this
29         // command does not require authorization, the presence of the sensitive
30         // area was not already checked as it is with most other commands that
31         // use the sensitive are so check it here
32         if(!CryptIsAsymAlgorithm(tpmKey->publicArea.type))
33             return TPM_RCS_KEY + RC_StartAuthSession_tpmKey;
34         // secret size cannot be 0
35         if(in->encryptedSalt.t.size == 0)
36             return TPM_RCS_VALUE + RC_StartAuthSession_encryptedSalt;
37         // Decrypting salt requires accessing the private portion of a key.
38         // Therefore, tmpKey can not be a key with only public portion loaded
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
39          if(tpmKey->attributes.publicOnly)
40              return TPM_RCS_HANDLE + RC_StartAuthSession_tpmKey;
41          // HMAC session input handle check.
42          // tpmKey should be a decryption key
43          if(!IS_ATTRIBUTE(tpmKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
44              return TPM_RCS_ATTRIBUTES + RC_StartAuthSession_tpmKey;
45          // Secret Decryption.  A TPM_RC_VALUE, TPM_RC_KEY or Unmarshal errors
46          // may be returned at this point
47          result = CryptSecretDecrypt(tpmKey, &in->nonceCaller, SECRET_KEY,
48                                      &in->encryptedSalt, &salt);
49          if(result != TPM_RC_SUCCESS)
50              return TPM_RCS_VALUE + RC_StartAuthSession_encryptedSalt;
51      }
52      else
53      {
54          // secret size must be 0
55          if(in->encryptedSalt.t.size != 0)
56              return TPM_RCS_VALUE + RC_StartAuthSession_encryptedSalt;
57          salt.t.size = 0;
58      }
59      switch(HandleGetType(in->bind))
60      {
61          case TPM_HT_TRANSIENT:
62          {
63              OBJECT      *object = HandleToObject(in->bind);
64              // If the bind handle references a transient object, make sure that we
65              // can get to the authorization value. Also, make sure that the object
66              // has a proper Name (nameAlg != TPM_ALG_NULL). If it doesn't, then
67              // it might be possible to bind to an object where the authValue is
68              // known. This does not create a real issue in that, if you know the
69              // authorization value, you can actually bind to the object. However,
70              // there is a potential
71              if(object->attributes.publicOnly == SET)
72                  return TPM_RCS_HANDLE + RC_StartAuthSession_bind;
73              break;
74          }
75          case TPM_HT_NV_INDEX:
76          // a PIN index can't be a bind object
77          {
78              NV_INDEX       *nvIndex = NvGetIndexInfo(in->bind, NULL);
79              if(IsNvPinPassIndex(nvIndex->publicArea.attributes)
80                 || IsNvPinFailIndex(nvIndex->publicArea.attributes))
81                  return TPM_RCS_HANDLE + RC_StartAuthSession_bind;
82              break;
83          }
84          default:
85              break;
86      }
87      // If 'symmetric' is a symmetric block cipher (not TPM_ALG_NULL or TPM_ALG_XOR)
88      // then the mode must be CFB.
89      if(in->symmetric.algorithm != TPM_ALG_NULL
90         && in->symmetric.algorithm != TPM_ALG_XOR
91         && in->symmetric.mode.sym != TPM_ALG_CFB)
92          return TPM_RCS_MODE + RC_StartAuthSession_symmetric;
93
94  // Internal Data Update and command output
95
96      // Create internal session structure.  TPM_RC_CONTEXT_GAP, TPM_RC_NO_HANDLES
97      // or TPM_RC_SESSION_MEMORY errors may be returned at this point.
98      //
99      // The detailed actions for creating the session context are not shown here
100     // as the details are implementation dependent
101     // SessionCreate sets the output handle and nonceTPM
102     result = SessionCreate(in->sessionType, in->authHash, &in->nonceCaller,
103                            &in->symmetric, in->bind, &salt, &out->sessionHandle,
104                            &out->nonceTPM);
```


 Copyright © TCG 2006-2020

```
105     return result;
106 }
107 #endif // CC_StartAuthSession
```

11.2 TPM2_PolicyRestart

11.2.1 General Description

This command allows a policy authorization session to be returned to its initial state. This command is used after the TPM returns TPM_RC_PCR_CHANGED. That response code indicates that a policy will fail because the PCR have changed after TPM2_PolicyPCR() was executed. Restarting the session allows the authorizations to be replayed because the session restarts with the same *nonceTPM*. If the PCR are valid for the policy, the policy may then succeed.

This command does not reset the policy ID or the policy start time.

 Copyright © TCG 2006-2020

11.2.2 Command and Response

Table 17 — TPM2_PolicyRestart Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyRestart
TPMI_SH_POLICY	sessionHandle	the handle for the policy session

Table 18 — TPM2_PolicyRestart Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

11.2.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PolicyRestart_fp.h"
3  #if CC_PolicyRestart  // Conditional expansion of this file
4  TPM_RC
5  TPM2_PolicyRestart(
6      PolicyRestart_In    *in             // IN: input parameter list
7      )
8  {
9      // Initialize policy session data
10     SessionResetPolicyData(SessionGet(in->sessionHandle));
11
12     return TPM_RC_SUCCESS;
13 }
14 #endif // CC_PolicyRestart
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

12 Object Commands

12.1 TPM2_Create

12.1.1 General Description

This command is used to create an object that can be loaded into a TPM using TPM2_Load(). If the command completes successfully, the TPM will create the new object and return the object's creation data (*creationData*), its public area (*outPublic*), and its encrypted sensitive area (*outPrivate*). Preservation of the returned data is the responsibility of the caller. The object will need to be loaded (TPM2_Load()) before it may be used. The only difference between the *inPublic* TPMT_PUBLIC template and the *outPublic* TPMT_PUBLIC object is in the *unique* field.

NOTE 1 This command may require temporary use of a transient resource, even though the object does not remain loaded after the command. See Part 1 Transient Resources.

TPM2B_PUBLIC template (*inPublic*) contains all of the fields necessary to define the properties of the new object. The setting for these fields is defined in "Public Area Template" in Part 1 of this specification and in "TPMA_OBJECT" in Part 2 of this specification. The size of the *unique* field shall not be checked for consistency with the other object parameters.

NOTE 2 For interoperability, the *unique* field should not be set to a value that is larger than allowed by object parameters, so that the unmarshaling will not fail. A size of zero is recommended. After unmarshaling, the TPM does not use the input *unique* field. It is, however, used in TPM2_CreatePrimary() and TPM2_CreateLoaded.

EXAMPLE 1 A TPM_ALG_RSA object with a *keyBits* of 2048 in the object's parameters should have a *unique* field that is no larger than 256 bytes.

EXAMPLE 2 TPM_ALG_KEYEDHASH or a TPM_ALG_SYMCIPHER object should have a *unique* field that is no larger than the digest produced by the object's *nameAlg*.

The *parentHandle* parameter shall reference a loaded decryption key that has both the public and sensitive area loaded.

When defining the object, the caller provides a template structure for the object in a TPM2B_PUBLIC structure (*inPublic*), an initial value for the object's *authValue (inSensitive.userAuth),* and, if the object is a symmetric object, an optional initial data value (*inSensitive.data*). The TPM shall validate the consistency of the attributes of *inPublic* according to the Creation rules in "TPMA_OBJECT" in TPM 2.0 Part 2.

The *inSensitive* parameter may be encrypted using parameter encryption.

The methods in this clause are used by both TPM2_Create() and TPM2_CreatePrimary(). When a value is indicated as being TPM-generated, the value is filled in by bits from the RNG if the command is TPM2_Create() and with values from **KDFa**() if the command is TPM2_CreatePrimary(). The parameters of each creation value are specified in TPM 2.0 Part 1.

The *sensitiveDataOrigin* attribute of *inPublic* shall be SET if *inSensitive.data* is an Empty Buffer and CLEAR if *inSensitive.data* is not an Empty Buffer or the TPM shall return TPM_RC_ATTRIBUTES.

If the Object is a not a *keyedHash* object, and the *sign* and *encrypt* attributes are CLEAR, the TPM shall return TPM_RC_ATTRIBUTES.

The TPM will create new data for the sensitive area and compute a TPMT_PUBLIC.*unique* from the sensitive area based on the object type:

a) For a symmetric key:

   1) If *inSensitive*.sensitive.*data* is the Empty Buffer, a TPM-generated key value is placed in the new object's TPMT_SENSITIVE.*sensitive.sym*. The size of the key will be determined by *inPublic.publicArea.parameters*.

 Copyright © TCG 2006-2020

2) If *inSensitive*.sensitive.*data* is not the Empty Buffer, the TPM will validate that the size of *inSensitive.data* is no larger than the key size indicated in the *inPublic template* (TPM_RC_SIZE) and copy the *inSensitive.data* to TPMT_SENSITIVE.*sensitive.sym* of the new object.

3) A TPM-generated obfuscation value is placed in TPMT_SENSITIVE.*sensitive.seedValue*. The size of the obfuscation value is the size of the digest produced by the nameAlg in *inPublic*. This value prevents the public *unique* value from leaking information about the *sensitive* area.

4) The TPMT_PUBLIC.*unique.sym* value for the new object is then generated, as shown in equation (1) below, by hashing the key and obfuscation values in the TPMT_SENSITIVE with the *nameAlg* of the object.

$$unique := \mathbf{H}_{nameAlg}(sensitive.seedValue.buffer \mid\mid sensitive.any.buffer) \quad (1)$$

b) If the Object is an asymmetric key:

1) If *inSensitive.sensitive.data* is not the Empty Buffer, then the TPM shall return TPM_RC_VALUE.

2) A TPM-generated private key value is created with the size determined by the parameters of inPublic.publicArea.parameters.

3) If the key is a Storage Key, a TPM-generated TPMT_SENSITIVE.*seedValue* value is created; otherwise, TPMT_SENSITIVE.*seedValue.size* is set to zero.

NOTE 3 An Object that is not a storage key has no child Objects to encrypt, so it does not need a symmetric key.

4) The public *unique* value is computed from the private key according to the methods of the key type.

5) If the key is an ECC key and the scheme required by the curveID is not the same as *scheme* in the public area of the template, then the TPM shall return TPM_RC_SCHEME.

6) If the key is an ECC key and the KDF required by the curveID is not the same as *kdf* in the pubic area of the template, then the TPM shall return TPM_RC_KDF.

NOTE 4 There is currently no command in which the caller may specify the KDF to be used with an ECC decryption key. Since there is no use for this capability, the reference implementation requires that the *kdf* in the template be set to TPM_ALG_NULL or TPM_RC_KDF is returned.

c) If the Object is a *keyedHash* object:

1) If *inSensitive.sensitive.data* is an Empty Buffer, and both *sign* and *decrypt* are CLEAR in the attributes of *inPublic*, the TPM shall return TPM_RC_ATTRIBUTES. This would be a data object with no data.

NOTE 5 Revisions 134 and earlier reference code did not check the error case of *sensitiveDataOrigin* SET and an Empty Buffer. Thus, some TPM implementations may also not have included this error check.

2) If *sign* and *decrypt* are both CLEAR, or if *sign* and *decrypt* are both SET and the *scheme* in the public area of the template is not TPM_ALG_NULL, the TPM shall return TPM_RC_SCHEME.

NOTE 6 Revisions 138 and earlier did not enforce this error case.

3) If *inSensitive.sensitive.data* is not an Empty Buffer, the TPM will copy the *inSensitive.sensitive.data* to TPMT_SENSITIVE.*sensitive.bits* of the new object.

NOTE 7 The size of inSensitive.sensitive.data is limited to be no larger than MAX_SYM_DATA.

4) If *inSensitive.sensitive.data* is an Empty Buffer, a TPM-generated key value that is the size of the digest produced by the *nameAlg* in *inPublic* is placed in TPMT_SENSITIVE.*sensitive.bits*.

 Copyright © TCG 2006-2020

5) A TPM-generated obfuscation value that is the size of the digest produced by the *nameAlg* of *inPublic* is placed in TPMT_SENSITIVE.*seedValue*.

6) The TPMT_PUBLIC.*unique.keyedHash* value for the new object is then generated, as shown in equation (1) above, by hashing the key and obfuscation values in the TPMT_SENSITIVE with the *nameAlg* of the object.

For TPM2_Load(), the TPM will apply normal symmetric protections to the created TPMT_SENSITIVE to create *outPublic*.

NOTE 8 The encryption key is derived from the symmetric seed in the sensitive area of the parent.

In addition to *outPublic* and *outPrivate*, the TPM will build a TPMS_CREATION_DATA structure for the object. TPMS_CREATION_DATA.*outsideInfo* is set to *outsideInfo*. This structure is returned in *creationData*. Additionally, the digest of this structure is returned in *creationHash*, and, finally, a TPMT_TK_CREATION is created so that the association between the creation data and the object may be validated by TPM2_CertifyCreation().

If the object being created is a Storage Key and *fixedParent* is SET in the attributes of *inPublic*, then the symmetric algorithms and parameters of *inPublic* are required to match those of the parent. The algorithms that must match are *inPublic.nameAlg*, and the values in *inPublic.parameters* that select the symmetric scheme. If *inPublic.nameAlg* does not match, the TPM shall return TPM_RC_HASH.If the symmetric scheme of the key does not match, the parent, the TPM shall return TPM_RC_SYMMETRIC. The TPM shall not use different response code to differentiate between mismatches of the components of *inPublic.parameters*. However, after this verification, when using the scheme to encrypt child objects, the TPM ignores the symmetric mode and uses TPM_ALG_CFB.

NOTE 9 The symmetric scheme is a TPMT_SYM_DEF_OBJECT. In a symmetric block ciphier, it is at *inPublic.parameters.symDetail.sym* and in an asymmetric object is at *inPublic.parameters.asymDetail.symmetric*.

NOTE 10 Prior to revision 01.34, the parent asymmetric algorithms were also checked for *fixedParent* storage keys.

 Copyright © TCG 2006-2020

12.1.2 Command and Response

Table 19 — TPM2_Create Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Create
TPMI_DH_OBJECT	@parentHandle	handle of parent for new object Auth Index: 1 Auth Role: USER
TPM2B_SENSITIVE_CREATE	inSensitive	the sensitive data
TPM2B_PUBLIC	inPublic	the public template
TPM2B_DATA	outsideInfo	data that will be included in the creation data for this object to provide permanent, verifiable linkage between this object and some object owner data
TPML_PCR_SELECTION	creationPCR	PCR that will be used in creation data

Table 20 — TPM2_Create Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PRIVATE	outPrivate	the private portion of the object
TPM2B_PUBLIC	outPublic	the public portion of the created object
TPM2B_CREATION_DATA	creationData	contains a TPMS_CREATION_DATA
TPM2B_DIGEST	creationHash	digest of *creationData* using *nameAlg* of *outPublic*
TPMT_TK_CREATION	creationTicket	ticket used by TPM2_CertifyCreation() to validate that the creation data was produced by the TPM

 Copyright © TCG 2006-2020

12.1.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "Object_spt_fp.h"
3  #include "Create_fp.h"
4  #if CC_Create  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*sensitiveDataOrigin* is CLEAR when *sensitive.data* is an Empty Buffer, or is SET when *sensitive.data* is not empty; *fixedTPM*, *fixedParent*, or *encryptedDuplication* attributes are inconsistent between themselves or with those of the parent object; inconsistent *restricted*, *decrypt* and *sign* attributes; attempt to inject sensitive data for an asymmetric key;
TPM_RC_HASH	non-duplicable storage key and its parent have different name algorithm
TPM_RC_KDF	incorrect KDF specified for decrypting keyed hash object
TPM_RC_KEY	invalid key size values in an asymmetric key public area or a provided symmetric key has a value that is not allowed
TPM_RC_KEY_SIZE	key size in public area for symmetric key differs from the size in the sensitive creation area; may also be returned if the TPM does not allow the key size to be used for a Storage Key
TPM_RC_OBJECT_MEMORY	a free slot is not available as scratch memory for object creation
TPM_RC_RANGE	the exponent value of an RSA key is not supported.
TPM_RC_SCHEME	inconsistent attributes *decrypt*, *sign*, or *restricted* and key's scheme ID; or hash algorithm is inconsistent with the scheme ID for keyed hash object
TPM_RC_SIZE	size of public *authPolicy* or sensitive *authValue* does not match digest size of the name algorithm sensitive data size for the keyed hash object is larger than is allowed for the scheme
TPM_RC_SYMMETRIC	a storage key with no symmetric algorithm specified; or non-storage key with symmetric algorithm different from TPM_ALG_NULL
TPM_RC_TYPE	unknown object type; *parentHandle* does not reference a restricted decryption key in the storage hierarchy with both public and sensitive portion loaded
TPM_RC_VALUE	exponent is not prime or could not find a prime using the provided parameters for an RSA key; unsupported name algorithm for an ECC key
TPM_RC_OBJECT_MEMORY	there is no free slot for the object

```
5  TPM_RC
6  TPM2_Create(
7      Create_In       *in,            // IN: input parameter list
8      Create_Out      *out            // OUT: output parameter list
9      )
10 {
11     TPM_RC                  result = TPM_RC_SUCCESS;
12     OBJECT                 *parentObject;
13     OBJECT                 *newObject;
14     TPMT_PUBLIC            *publicArea;
15
16 // Input Validation
17     parentObject = HandleToObject(in->parentHandle);
```

```
18      pAssert(parentObject != NULL);
19
20      // Does parent have the proper attributes?
21      if(!ObjectIsParent(parentObject))
22          return TPM_RCS_TYPE + RC_Create_parentHandle;
23
24      // Get a slot for the creation
25      newObject = FindEmptyObjectSlot(NULL);
26      if(newObject == NULL)
27          return TPM_RC_OBJECT_MEMORY;
28      // If the TPM2B_PUBLIC was passed as a structure, marshal it into is canonical
29      // form for processing
30
31      // to save typing.
32      publicArea = &newObject->publicArea;
33
34      // Copy the input structure to the allocated structure
35      *publicArea = in->inPublic.publicArea;
36
37      // Check attributes in input public area. CreateChecks() checks the things that
38      // are unique to creation and then validates the attributes and values that are
39      // common to create and load.
40      result = CreateChecks(parentObject, publicArea,
41                            in->inSensitive.sensitive.data.t.size);
42      if(result != TPM_RC_SUCCESS)
43          return RcSafeAddToResult(result, RC_Create_inPublic);
44      // Clean up the authValue if necessary
45      if(!AdjustAuthSize(&in->inSensitive.sensitive.userAuth, publicArea->nameAlg))
46          return TPM_RCS_SIZE + RC_Create_inSensitive;
47
48  // Command Output
49      // Create the object using the default TPM random-number generator
50      result = CryptCreateObject(newObject, &in->inSensitive.sensitive, NULL);
51      if(result != TPM_RC_SUCCESS)
52          return result;
53      // Fill in creation data
54      FillInCreationData(in->parentHandle, publicArea->nameAlg,
55                         &in->creationPCR, &in->outsideInfo,
56                         &out->creationData, &out->creationHash);
57
58      // Compute creation ticket
59      TicketComputeCreation(EntityGetHierarchy(in->parentHandle), &newObject->name,
60                            &out->creationHash, &out->creationTicket);
61
62      // Prepare output private data from sensitive
63      SensitiveToPrivate(&newObject->sensitive, &newObject->name, parentObject,
64                         publicArea->nameAlg,
65                         &out->outPrivate);
66
67      // Finish by copying the remaining return values
68      out->outPublic.publicArea = newObject->publicArea;
69
70      return TPM_RC_SUCCESS;
71  }
72  #endif // CC_Create
```

12.2 TPM2_Load

12.2.1 General Description

This command is used to load objects into the TPM. This command is used when both a TPM2B_PUBLIC and TPM2B_PRIVATE are to be loaded. If only a TPM2B_PUBLIC is to be loaded, the TPM2_LoadExternal command is used.

NOTE 1 Loading an object is not the same as restoring a saved object context.

The object's TPMA_OBJECT attributes will be checked according to the rules defined in "TPMA_OBJECT" in TPM 2.0 Part 2 of this specification. If the Object is a not a *keyedHash* object, and the *sign* and *encrypt* attributes are CLEAR, the TPM shall return TPM_RC_ATTRIBUTES.

Objects loaded using this command will have a Name. The Name is the concatenation of *nameAlg* and the digest of the public area using the *nameAlg.*

NOTE 2 *nameAlg* is a parameter in the public area of the inPublic structure.

If *inPrivate.size* is zero, the load will fail.

After *inPrivate.buffer* is decrypted using the symmetric key of the parent, the integrity value shall be checked before the sensitive area is used, or unmarshaled.

NOTE 3 Checking the integrity before the data is used prevents attacks on the sensitive area by fuzzing the data and looking at the differences in the response codes.

The command returns a handle for the loaded object and the Name that the TPM computed for *inPublic.public* (that is, the digest of the TPMT_PUBLIC structure in *inPublic*).

NOTE 4 The TPM-computed Name is provided as a convenience to the caller for those cases where the caller does not implement the hash algorithms specified in the *nameAlg* of the object.

NOTE 5 The returned handle is associated with the object until the object is flushed (TPM2_FlushContext) or until the next TPM2_Startup.

For all objects, the size of the key in the sensitive area shall be consistent with the key size indicated in the public area or the TPM shall return TPM_RC_KEY_SIZE.

Before use, a loaded object shall be checked to validate that the public and sensitive portions are properly linked, cryptographically. Use of an object includes use in any policy command. If the parts of the object are not properly linked, the TPM shall return TPM_RC_BINDING. If a weak symmetric key is in the sensitive portion, the TPM shall return TPM_RC_KEY.

EXAMPLE 1 For a symmetric object, the unique value in the public area shall be the digest of the sensitive key and the obfuscation value.

EXAMPLE 2 For a two-prime RSA key, the remainder when dividing the public modulus by the private key shall be zero and it shall be possible to form a private exponent from the two prime factors of the public modulus.

EXAMPLE 3 For an ECC key, the public point shall be f(x) where x is the private key.

 Copyright © TCG 2006-2020

12.2.2 Command and Response

Table 21 — TPM2_Load Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Load
TPMI_DH_OBJECT	@parentHandle	TPM handle of parent key; shall not be a reserved handle Auth Index: 1 Auth Role: USER
TPM2B_PRIVATE	inPrivate	the private portion of the object
TPM2B_PUBLIC	inPublic	the public portion of the object

Table 22 — TPM2_Load Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM_HANDLE	objectHandle	handle of type TPM_HT_TRANSIENT for the loaded object
TPM2B_NAME	name	Name of the loaded object

 Copyright © TCG 2006-2020

12.2.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "Load_fp.h"
3   #if CC_Load  // Conditional expansion of this file
4   #include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*inPulblic* attributes are not allowed with selected parent
TPM_RC_BINDING	*inPrivate* and *inPublic* are not cryptographically bound
TPM_RC_HASH	incorrect hash selection for signing key or the *nameAlg* for '*inPubic* is not valid
TPM_RC_INTEGRITY	HMAC on *inPrivate* was not valid
TPM_RC_KDF	KDF selection not allowed
TPM_RC_KEY	the size of the object's *unique* field is not consistent with the indicated size in the object's parameters
TPM_RC_OBJECT_MEMORY	no available object slot
TPM_RC_SCHEME	the signing scheme is not valid for the key
TPM_RC_SENSITIVE	the *inPrivate* did not unmarshal correctly
TPM_RC_SIZE	*inPrivate* missing, or *authPolicy* size for *inPublic* or is not valid
TPM_RC_SYMMETRIC	symmetric algorithm not provided when required
TPM_RC_TYPE	*parentHandle* is not a storage key, or the object to load is a storage key but its parameters do not match the parameters of the parent.
TPM_RC_VALUE	decryption failure

```
5   TPM_RC
6   TPM2_Load(
7       Load_In         *in,            // IN: input parameter list
8       Load_Out        *out            // OUT: output parameter list
9       )
10  {
11      TPM_RC                      result = TPM_RC_SUCCESS;
12      TPMT_SENSITIVE              sensitive;
13      OBJECT                     *parentObject;
14      OBJECT                     *newObject;
15
16  // Input Validation
17      // Don't get invested in loading if there is no place to put it.
18      newObject = FindEmptyObjectSlot(&out->objectHandle);
19      if(newObject == NULL)
20          return TPM_RC_OBJECT_MEMORY;
21
22      if(in->inPrivate.t.size == 0)
23          return TPM_RCS_SIZE + RC_Load_inPrivate;
24
25      parentObject = HandleToObject(in->parentHandle);
26      pAssert(parentObject != NULL);
27      // Is the object that is being used as the parent actually a parent.
28      if(!ObjectIsParent(parentObject))
29          return TPM_RCS_TYPE + RC_Load_parentHandle;
30
31      // Compute the name of object. If there isn't one, it is because the nameAlg is
32      // not valid.
```

```
33      PublicMarshalAndComputeName(&in->inPublic.publicArea, &out->name);
34      if(out->name.t.size == 0)
35          return TPM_RCS_HASH + RC_Load_inPublic;
36
37      // Retrieve sensitive data.
38      result = PrivateToSensitive(&in->inPrivate.b, &out->name.b, parentObject,
39                                  in->inPublic.publicArea.nameAlg,
40                                  &sensitive);
41      if(result != TPM_RC_SUCCESS)
42          return RcSafeAddToResult(result, RC_Load_inPrivate);
43
44  // Internal Data Update
45      // Load and validate object
46      result = ObjectLoad(newObject, parentObject,
47                          &in->inPublic.publicArea, &sensitive,
48                          RC_Load_inPublic, RC_Load_inPrivate,
49                          &out->name);
50      if(result == TPM_RC_SUCCESS)
51      {
52          // Set the common OBJECT attributes for a loaded object.
53          ObjectSetLoadedAttributes(newObject, in->parentHandle);
54      }
55      return result;
56
57  }
58  #endif // CC_Load
```

 Copyright © TCG 2006-2020

12.3 TPM2_LoadExternal

12.3.1 General Description

This command is used to load an object that is not a Protected Object into the TPM. The command allows loading of a public area or both a public and sensitive area.

NOTE 1 Typical use for loading a public area is to allow the TPM to validate an asymmetric signature. Typical use for loading both a public and sensitive area is to allow the TPM to be used as a crypto accelerator.

Load of a public external object area allows the object to be associated with a hierarchy so that the correct algorithms may be used when creating tickets. The *hierarchy* parameter provides this association. If the public and sensitive portions of the object are loaded, *hierarchy* is required to be TPM_RH_NULL.

NOTE 2 If both the public and private portions of an object are loaded, the object is not allowed to appear to be part of a hierarchy.

The object's TPMA_OBJECT attributes will be checked according to the rules defined in "TPMA_OBJECT" in TPM 2.0 Part 2. In particular, *fixedTPM*, *fixedParent*, and *restricted* shall be CLEAR if *inPrivate* is not the Empty Buffer.

NOTE 3 The duplication status of a public key needs to be able to be the same as the full key which may be resident on a different TPM. If both the public and private parts of the key are loaded, then it is not possible for the key to be either *fixedTPM* or *fixedParent*, since, its private area would not be available in the clear to load.

Objects loaded using this command will have a Name. The Name is the *nameAlg* of the object concatenated with the digest of the public area using the *nameAlg.* The Qualified Name for the object will be the same as its Name. The TPM will validate that the *authPolicy* is either the size of the digest produced by *nameAlg* or the Empty Buffer.

NOTE 4 If *nameAlg* is TPM_ALG_NULL, then the Name is the Empty Buffer. When the authorization value for an object with no Name is computed, no Name value is included in the HMAC. To ensure that these unnamed entities are not substituted, they should have an *authValue* that is statistically unique.

NOTE 5 The digest size for TPM_ALG_NULL is zero.

If the *nameAlg* is TPM_ALG_NULL, the TPM shall not verify the cryptographic binding between the public and sensitive areas, but the TPM will validate that the size of the key in the sensitive area is consistent with the size indicated in the public area. If it is not, the TPM shall return TPM_RC_KEY_SIZE.

NOTE 6 For an ECC object, the TPM will verify that the public key is on the curve of the key before the public area is used.

If *nameAlg* is not TPM_ALG_NULL, then the same consistency checks between *inPublic* and *inPrivate* are made as for TPM2_Load().

NOTE 7 Consistency checks are necessary because an object with a Name needs to have the public and sensitive portions cryptographically bound so that an attacker cannot mix pubic and sensitive areas.

The command returns a handle for the loaded object and the Name that the TPM computed for *inPublic.public* (that is, the TPMT_PUBLIC structure in *inPublic*).

NOTE 8 The TPM-computed Name is provided as a convenience to the caller for those cases where the caller does not implement the hash algorithm specified in the *nameAlg* of the object.

 Copyright © TCG 2006-2020

The *hierarchy* parameter associates the external object with a hierarchy. External objects are flushed when their associated hierarchy is disabled. If *hierarchy* is TPM_RH_NULL, the object is part of no hierarchy, and there is no implicit flush.

If *hierarchy* is TPM_RH_NULL or *nameAlg* is TPM_ALG_NULL, a ticket produced using the object shall be a NULL Ticket.

EXAMPLE If a key is loaded with hierarchy set to TPM_RH_NULL, then TPM2_VerifySignature() will produce a NULL Ticket of the required type.

External objects are Temporary Objects. The saved external object contexts shall be invalidated at the next TPM Reset.

If a weak symmetric key is in the sensitive area, the TPM shall return TPM_RC_KEY.

For an RSA key, the private exponent is computed using the two prime factors of the public modulus. One of the primes is P, and the second prime (Q) is found by dividing the public modulus by P. A TPM may return an error (TPM_RC_BINDING) if the bit size of P and Q are not the same."

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

12.3.2 Command and Response

Table 23 — TPM2_LoadExternal Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit, encrypt, or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_LoadExternal
TPM2B_SENSITIVE	inPrivate	the sensitive portion of the object (optional)
TPM2B_PUBLIC+	inPublic	the public portion of the object
TPMI_RH_HIERARCHY+	hierarchy	hierarchy with which the object area is associated

Table 24 — TPM2_LoadExternal Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM_HANDLE	objectHandle	handle of type TPM_HT_TRANSIENT for the loaded object
TPM2B_NAME	name	name of the loaded object

 Copyright © TCG 2006-2020

12.3.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "LoadExternal_fp.h"
3   #if CC_LoadExternal  // Conditional expansion of this file
4   #include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	'*fixedParent*", *fixedTPM*, and *restricted* must be CLEAR if sensitive portion of an object is loaded
TPM_RC_BINDING	the *inPublic* and *inPrivate* structures are not cryptographically bound
TPM_RC_HASH	incorrect hash selection for signing key
TPM_RC_HIERARCHY	*hierarchy* is turned off, or only NULL hierarchy is allowed when loading public and private parts of an object
TPM_RC_KDF	incorrect KDF selection for decrypting *keyedHash* object
TPM_RC_KEY	the size of the object's *unique* field is not consistent with the indicated size in the object's parameters
TPM_RC_OBJECT_MEMORY	if there is no free slot for an object
TPM_RC_ECC_POINT	for a public-only ECC key, the ECC point is not on the curve
TPM_RC_SCHEME	the signing scheme is not valid for the key
TPM_RC_SIZE	*authPolicy* is not zero and is not the size of a digest produced by the object's *nameAlg* TPM_RH_NULL hierarchy
TPM_RC_SYMMETRIC	symmetric algorithm not provided when required
TPM_RC_TYPE	*inPublic* and *inPrivate* are not the same type

```
5   TPM_RC
6   TPM2_LoadExternal(
7       LoadExternal_In     *in,            // IN: input parameter list
8       LoadExternal_Out    *out            // OUT: output parameter list
9       )
10  {
11      TPM_RC               result;
12      OBJECT              *object;
13      TPMT_SENSITIVE      *sensitive = NULL;
14
15  // Input Validation
16      // Don't get invested in loading if there is no place to put it.
17      object = FindEmptyObjectSlot(&out->objectHandle);
18      if(object == NULL)
19          return TPM_RC_OBJECT_MEMORY;
20
21      // If the hierarchy to be associated with this object is turned off, the object
22      // cannot be loaded.
23      if(!HierarchyIsEnabled(in->hierarchy))
24          return TPM_RCS_HIERARCHY + RC_LoadExternal_hierarchy;
25
26      // For loading an object with both public and sensitive
27      if(in->inPrivate.size != 0)
28      {
29          // An external object with a sensitive area can only be loaded in the
30          // NULL hierarchy
31          if(in->hierarchy != TPM_RH_NULL)
32              return TPM_RCS_HIERARCHY + RC_LoadExternal_hierarchy;
33          // An external object with a sensitive area must have fixedTPM == CLEAR
```

 Copyright © TCG 2006-2020

```
34            // fixedParent == CLEAR so that it does not appear to be a key created by
35            // this TPM.
36            if(IS_ATTRIBUTE(in->inPublic.publicArea.objectAttributes, TPMA_OBJECT,
37                            fixedTPM)
38               || IS_ATTRIBUTE(in->inPublic.publicArea.objectAttributes, TPMA_OBJECT,
39                               fixedParent)
40               || IS_ATTRIBUTE(in->inPublic.publicArea.objectAttributes, TPMA_OBJECT,
41                               restricted))
42                return TPM_RCS_ATTRIBUTES + RC_LoadExternal_inPublic;
43
44            // Have sensitive point to something other than NULL so that object
45            // initialization will load the sensitive part too
46            sensitive = &in->inPrivate.sensitiveArea;
47        }
48
49        // Need the name to initialize the object structure
50        PublicMarshalAndComputeName(&in->inPublic.publicArea, &out->name);
51
52        // Load and validate key
53        result = ObjectLoad(object, NULL,
54                            &in->inPublic.publicArea, sensitive,
55                            RC_LoadExternal_inPublic, RC_LoadExternal_inPrivate,
56                            &out->name);
57        if(result == TPM_RC_SUCCESS)
58        {
59            object->attributes.external = SET;
60            // Set the common OBJECT attributes for a loaded object.
61            ObjectSetLoadedAttributes(object, in->hierarchy);
62        }
63        return result;
64    }
65    #endif // CC_LoadExternal
```

 Copyright © TCG 2006-2020

12.4 TPM2_ReadPublic

12.4.1 General Description

This command allows access to the public area of a loaded object.

Use of the *objectHandle* does not require authorization.

NOTE Since the caller is not likely to know the public area of the object associated with *objectHandle*, it would not be possible to include the Name associated with *objectHandle* in the *cpHash* computation.

If *objectHandle* references a sequence object, the TPM shall return TPM_RC_SEQUENCE.

 Copyright © TCG 2006-2020

12.4.2 Command and Response

Table 25 — TPM2_ReadPublic Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ReadPublic
TPMI_DH_OBJECT	objectHandle	TPM handle of an object Auth Index: None

Table 26 — TPM2_ReadPublic Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PUBLIC	outPublic	structure containing the public area of an object
TPM2B_NAME	name	name of the object
TPM2B_NAME	qualifiedName	the Qualified Name of the object

 Copyright © TCG 2006-2020

12.4.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "ReadPublic_fp.h"
3   #if CC_ReadPublic  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_SEQUENCE	can not read the public area of a sequence object

```
4   TPM_RC
5   TPM2_ReadPublic(
6       ReadPublic_In    *in,             // IN: input parameter list
7       ReadPublic_Out   *out             // OUT: output parameter list
8       )
9   {
10      OBJECT                  *object = HandleToObject(in->objectHandle);
11
12  // Input Validation
13      // Can not read public area of a sequence object
14      if(ObjectIsSequence(object))
15          return TPM_RC_SEQUENCE;
16
17  // Command Output
18      out->outPublic.publicArea = object->publicArea;
19      out->name = object->name;
20      out->qualifiedName = object->qualifiedName;
21
22      return TPM_RC_SUCCESS;
23  }
24  #endif // CC_ReadPublic
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

12.5 TPM2_ActivateCredential

12.5.1 General Description

This command enables the association of a credential with an object in a way that ensures that the TPM has validated the parameters of the credentialed object.

If both the public and private portions of *activateHandle* and *keyHandle* are not loaded, then the TPM shall return TPM_RC_AUTH_UNAVAILABLE.

If *keyHandle* is not a Storage Key, then the TPM shall return TPM_RC_TYPE.

Authorization for *activateHandle* requires the ADMIN role.

The key associated with *keyHandle* is used to recover a seed from secret, which is the encrypted seed. The Name of the object associated with activateHandle and the recovered seed are used in a KDF to recover the symmetric key. The recovered seed (but not the Name) is used in a KDF to recover the HMAC key.

The HMAC is used to validate that the *credentialBlob* is associated with *activateHandle* and that the data in *credentialBlob* has not been modified. The linkage to the object associated with activateHandle is achieved by including the Name in the HMAC calculation.

If the integrity checks succeed, *credentialBlob* is decrypted and returned as *certInfo*.

NOTE The output *certInfo* parameter is an application defined value. It is typically a symmetric key or seed that is used to decrypt a certificate. See the TPM2_MakeCredential *credential* input parameter.

 Copyright © TCG 2006-2020

12.5.2 Command and Response

Table 27 — TPM2_ActivateCredential Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ActivateCredential
TPMI_DH_OBJECT	@activateHandle	handle of the object associated with certificate in *credentialBlob* Auth Index: 1 Auth Role: ADMIN
TPMI_DH_OBJECT	@keyHandle	loaded key used to decrypt the TPMS_SENSITIVE in *credentialBlob* Auth Index: 2 Auth Role: USER
TPM2B_ID_OBJECT	credentialBlob	the credential
TPM2B_ENCRYPTED_SECRET	secret	*keyHandle* algorithm-dependent encrypted seed that protects *credentialBlob*

Table 28 — TPM2_ActivateCredential Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	certInfo	the decrypted certificate information the data should be no larger than the size of the digest of the *nameAlg* associated with *keyHandle*

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

12.5.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "ActivateCredential_fp.h"
3  #if CC_ActivateCredential  // Conditional expansion of this file
4  #include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*keyHandle* does not reference a decryption key
TPM_RC_ECC_POINT	*secret* is invalid (when *keyHandle* is an ECC key)
TPM_RC_INSUFFICIENT	*secret* is invalid (when *keyHandle* is an ECC key)
TPM_RC_INTEGRITY	*credentialBlob* fails integrity test
TPM_RC_NO_RESULT	*secret* is invalid (when *keyHandle* is an ECC key)
TPM_RC_SIZE	*secret* size is invalid or the *credentialBlob* does not unmarshal correctly
TPM_RC_TYPE	*keyHandle* does not reference an asymmetric key.
TPM_RC_VALUE	*secret* is invalid (when *keyHandle* is an RSA key)

```
5  TPM_RC
6  TPM2_ActivateCredential(
7      ActivateCredential_In   *in,            // IN: input parameter list
8      ActivateCredential_Out  *out            // OUT: output parameter list
9      )
10 {
11     TPM_RC                   result = TPM_RC_SUCCESS;
12     OBJECT                  *object;          // decrypt key
13     OBJECT                  *activateObject;  // key associated with credential
14     TPM2B_DATA               data;          // credential data
15
16 // Input Validation
17
18     // Get decrypt key pointer
19     object = HandleToObject(in->keyHandle);
20
21     // Get certificated object pointer
22     activateObject = HandleToObject(in->activateHandle);
23
24     // input decrypt key must be an asymmetric, restricted decryption key
25     if(!CryptIsAsymAlgorithm(object->publicArea.type)
26        || !IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, decrypt)
27        || !IS_ATTRIBUTE(object->publicArea.objectAttributes,
28                         TPMA_OBJECT, restricted))
29         return TPM_RCS_TYPE + RC_ActivateCredential_keyHandle;
30
31 // Command output
32
33     // Decrypt input credential data via asymmetric decryption.  A
34     // TPM_RC_VALUE, TPM_RC_KEY or unmarshal errors may be returned at this
35     // point
36     result = CryptSecretDecrypt(object, NULL, IDENTITY_STRING, &in->secret, &data);
37     if(result != TPM_RC_SUCCESS)
38     {
39         if(result == TPM_RC_KEY)
40             return TPM_RC_FAILURE;
41         return RcSafeAddToResult(result, RC_ActivateCredential_secret);
42     }
43
```

```
44      // Retrieve secret data.  A TPM_RC_INTEGRITY error or unmarshal
45      // errors may be returned at this point
46      result = CredentialToSecret(&in->credentialBlob.b,
47                                  &activateObject->name.b,
48                                  &data.b,
49                                  object,
50                                  &out->certInfo);
51      if(result != TPM_RC_SUCCESS)
52          return RcSafeAddToResult(result, RC_ActivateCredential_credentialBlob);
53
54      return TPM_RC_SUCCESS;
55  }
56  #endif // CC_ActivateCredential
```

 Copyright © TCG 2006-2020

12.6 TPM2_MakeCredential

12.6.1 General Description

This command allows the TPM to perform the actions required of a Certificate Authority (CA) in creating a TPM2B_ID_OBJECT containing an activation credential.

NOTE The input *credential* parameter is an application defined value. It is typically a symmetric key or seed that is used to encrypt a certificate. See the TPM2_ActivateCredential *certInfo* output parameter.

The TPM will produce a TPM2B_ID_OBJECT according to the methods in “Credential Protection” in TPM 2.0 Part 1.

The loaded public area referenced by *handle* is required to be the public area of a Storage key, otherwise, the credential cannot be properly sealed.

This command does not use any TPM secrets nor does it require authorization. It is a convenience function, using the TPM to perform cryptographic calculations that could be done externally.

 Copyright © TCG 2006-2020

12.6.2 Command and Response

Table 29 — TPM2_MakeCredential Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit, encrypt, or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_MakeCredential
TPMI_DH_OBJECT	handle	loaded public area, used to encrypt the sensitive area containing the credential key Auth Index: None
TPM2B_DIGEST	credential	the credential information
TPM2B_NAME	objectName	Name of the object to which the credential applies

Table 30 — TPM2_MakeCredential Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ID_OBJECT	credentialBlob	the credential
TPM2B_ENCRYPTED_SECRET	secret	*handle* algorithm-dependent data that wraps the key that encrypts *credentialBlob*

 Copyright © TCG 2006-2020

12.6.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "MakeCredential_fp.h"
3  #if CC_MakeCredential  // Conditional expansion of this file
4  #include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_KEY	*handle* referenced an ECC key that has a unique field that is not a point on the curve of the key
TPM_RC_SIZE	*credential* is larger than the digest size of Name algorithm of *handle*
TPM_RC_TYPE	*handle* does not reference an asymmetric decryption key

```
5  TPM_RC
6  TPM2_MakeCredential(
7      MakeCredential_In   *in,            // IN: input parameter list
8      MakeCredential_Out  *out            // OUT: output parameter list
9      )
10 {
11     TPM_RC               result = TPM_RC_SUCCESS;
12
13     OBJECT              *object;
14     TPM2B_DATA           data;
15
16 // Input Validation
17
18     // Get object pointer
19     object = HandleToObject(in->handle);
20
21     // input key must be an asymmetric, restricted decryption key
22     // NOTE: Needs to be restricted to have a symmetric value.
23     if(!CryptIsAsymAlgorithm(object->publicArea.type)
24        || !IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, decrypt)
25        || !IS_ATTRIBUTE(object->publicArea.objectAttributes,
26                         TPMA_OBJECT, restricted))
27         return TPM_RCS_TYPE + RC_MakeCredential_handle;
28
29     // The credential information may not be larger than the digest size used for
30     // the Name of the key associated with handle.
31     if(in->credential.t.size > CryptHashGetDigestSize(object->publicArea.nameAlg))
32         return TPM_RCS_SIZE + RC_MakeCredential_credential;
33
34 // Command Output
35
36     // Make encrypt key and its associated secret structure.
37     out->secret.t.size = sizeof(out->secret.t.secret);
38     result = CryptSecretEncrypt(object, IDENTITY_STRING, &data, &out->secret);
39     if(result != TPM_RC_SUCCESS)
40         return result;
41
42     // Prepare output credential data from secret
43     SecretToCredential(&in->credential, &in->objectName.b, &data.b,
44                        object, &out->credentialBlob);
45
46     return TPM_RC_SUCCESS;
47 }
48 #endif // CC_MakeCredential
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

12.7 TPM2_Unseal

12.7.1 General Description

This command returns the data in a loaded Sealed Data Object.

NOTE 1 A random, TPM-generated, Sealed Data Object may be created by the TPM with TPM2_Create() or TPM2_CreatePrimary() using the template for a Sealed Data Object.

NOTE 2 TPM 1.2 hard coded PCR authorization. TPM 2.0 PCR authorization requires a policy.

The returned value may be encrypted using authorization session encryption.

If either *restricted*, *decrypt,* or *sign* is SET in the attributes of *itemHandle,* then the TPM shall return TPM_RC_ATTRIBUTES. If the *type* of *itemHandle* is not TPM_ALG_KEYEDHASH, then the TPM shall return TPM_RC_TYPE.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

12.7.2 Command and Response

Table 31 — TPM2_Unseal Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Unseal
TPMI_DH_OBJECT	@itemHandle	handle of a loaded data object Auth Index: 1 Auth Role: USER

Table 32 — TPM2_Unseal Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_SENSITIVE_DATA	outData	unsealed data Size of *outData* is limited to be no more than 128 octets.

 Copyright © TCG 2006-2020

12.7.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "Unseal_fp.h"
3   #if CC_Unseal  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*itemHandle* has wrong attributes
TPM_RC_TYPE	*itemHandle* is not a KEYEDHASH data object

```
4   TPM_RC
5   TPM2_Unseal(
6       Unseal_In           *in,
7       Unseal_Out          *out
8       )
9   {
10      OBJECT                  *object;
11  // Input Validation
12      // Get pointer to loaded object
13      object = HandleToObject(in->itemHandle);
14
15      // Input handle must be a data object
16      if(object->publicArea.type != TPM_ALG_KEYEDHASH)
17          return TPM_RCS_TYPE + RC_Unseal_itemHandle;
18      if(IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, decrypt)
19         || IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, sign)
20         || IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, restricted))
21          return TPM_RCS_ATTRIBUTES + RC_Unseal_itemHandle;
22  // Command Output
23      // Copy data
24      out->outData = object->sensitive.sensitive.bits;
25      return TPM_RC_SUCCESS;
26  }
27  #endif // CC_Unseal
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

12.8 TPM2_ObjectChangeAuth

12.8.1 General Description

This command is used to change the authorization secret for a TPM-resident object.

If successful, a new private area for the TPM-resident object associated with *objectHandle* is returned, which includes the new authorization value.

This command does not change the authorization of the TPM-resident object on which it operates. Therefore, the old authValue (of the TPM-resident object) is used when generating the response HMAC key if required.

NOTE 1 The returned *outPrivate* will need to be loaded before the new authorization will apply.

NOTE 2 The TPM-resident object may be persistent and changing the authorization value of the persistent object could prevent other users from accessing the object. This is why this command does not change the TPM-resident object.

EXAMPLE If a persistent key is being used as a Storage Root Key and the authorization of the key is a well-known value so that the key can be used generally, then changing the authorization value in the persistent key would deny access to other users.

This command may not be used to change the authorization value for an NV Index or a Primary Object.

NOTE 3 If an NV Index is to have a new authorization, it is done with TPM2_NV_ChangeAuth().

NOTE 4 If a Primary Object is to have a new authorization, it needs to be recreated (TPM2_CreatePrimary()).

 Copyright © TCG 2006-2020

12.8.2 Command and Response

Table 33 — TPM2_ObjectChangeAuth Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ObjectChangeAuth
TPMI_DH_OBJECT	@objectHandle	handle of the object Auth Index: 1 Auth Role: ADMIN
TPMI_DH_OBJECT	parentHandle	handle of the parent Auth Index: None
TPM2B_AUTH	newAuth	new authorization value

Table 34 — TPM2_ObjectChangeAuth Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PRIVATE	outPrivate	private area containing the new authorization value

 Copyright © TCG 2006-2020

12.8.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "ObjectChangeAuth_fp.h"
3   #if CC_ObjectChangeAuth  // Conditional expansion of this file
4   #include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_SIZE	*newAuth* is larger than the size of the digest of the Name algorithm of *objectHandle*
TPM_RC_TYPE	the key referenced by *parentHandle* is not the parent of the object referenced by *objectHandle*; or *objectHandle* is a sequence object.

```
5   TPM_RC
6   TPM2_ObjectChangeAuth(
7       ObjectChangeAuth_In     *in,            // IN: input parameter list
8       ObjectChangeAuth_Out    *out            // OUT: output parameter list
9       )
10  {
11      TPMT_SENSITIVE           sensitive;
12
13      OBJECT                  *object = HandleToObject(in->objectHandle);
14      TPM2B_NAME               QNCompare;
15
16  // Input Validation
17
18      // Can not change authorization on sequence object
19      if(ObjectIsSequence(object))
20          return TPM_RCS_TYPE + RC_ObjectChangeAuth_objectHandle;
21
22      // Make sure that the authorization value is consistent with the nameAlg
23      if(!AdjustAuthSize(&in->newAuth, object->publicArea.nameAlg))
24          return TPM_RCS_SIZE + RC_ObjectChangeAuth_newAuth;
25
26      // Parent handle should be the parent of object handle.  In this
27      // implementation we verify this by checking the QN of object.  Other
28      // implementation may choose different method to verify this attribute.
29      ComputeQualifiedName(in->parentHandle,
30                           object->publicArea.nameAlg,
31                           &object->name, &QNCompare);
32      if(!MemoryEqual2B(&object->qualifiedName.b, &QNCompare.b))
33          return TPM_RCS_TYPE + RC_ObjectChangeAuth_parentHandle;
34
35  // Command Output
36      // Prepare the sensitive area with the new authorization value
37      sensitive = object->sensitive;
38      sensitive.authValue = in->newAuth;
39
40      // Protect the sensitive area
41      SensitiveToPrivate(&sensitive, &object->name, HandleToObject(in->parentHandle),
42                         object->publicArea.nameAlg,
43                         &out->outPrivate);
44      return TPM_RC_SUCCESS;
45  }
46  #endif // CC_ObjectChangeAuth
```

 Copyright © TCG 2006-2020

12.9 TPM2_CreateLoaded

12.9.1 General Description

This command creates an object and loads it in the TPM. This command allows creation of any type of object (Primary, Ordinary, or Derived) depending on the type of *parentHandle.* If *parentHandle* references a Primary Seed, then a Primary Object is created; if *parentHandle* references a Storage Parent, then an Ordinary Object is created; and if *parentHandle* references a Derivation Parent, then a Derived Object is generated.

The input validation is the same as for TPM2_Create() and TPM2_CreatePrimary() with one exception: when *parentHandle* references a Derivation Parent, then *sensitiveDataOrigin* in *inPublic* is required to be CLEAR.

Note 1 In the general descriptions of TPM2_Create() and TPM2_CreatePrimary() the validations refer to a TPMT_PUBLIC structure that is in *inPublic.* For TPM2_CreateLoaded(), *inPublic* is a TPM2B_TEMPLATE that may contain a TPMT_PUBLIC that is used for object creation. For object derivation, the *unique* field can contain a *label* and *context* that are used in the derivation process. To allow both the TPMT_PUBLIC and the derivation variation, a TPM2B_TEMPLATE is used. When referring to the checks in TPM2_Create() and TPM2_CreatePrimary(), TPM2B_TEMPLATE should be assumed to contain a TPMT_PUBLIC.

If *parentHandle* references a Derivation Parent, then the TPM may return TPM_RC_TYPE if the key type to be generated is an RSA key.

If *parentHandle* references a Derivation Parent or a Primary Seed, then *outPrivate* will be an Empty Buffer.

NOTE 2 Returning outPrivate would imply that the returned primary or derived object can be loaded and it cannot. It can only be re-derived.

A primary key cannot be loaded is because loading a key is a way to attack the protections of a key (e.g. using DPA). A saved context for a primary object is protected. The TPM will go into failure mode if the integrity of a saved context is good but the fingerprint doesn't decrypt. It is not possible to have these protections on loaded objects because this would be a simple way for an attacker to put the TPM into failure mode Saved contexts are assumed to be under control of the driver but loaded objects are not.

If all objects were derived from their parents then, load could not be used as an attack. However, that would preclude importation of objects and key hierarchies.

NOTE 3 Unlike TPM2_Create() and TPM2_CreatePrimary(), this command does not return creation data. If creation data is needed, then TPM2_Create() or TPM2_CreatePrimary() should be used.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

12.9.2 Command and Response

Table 35 — TPM2_CreateLoaded Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_CreateLoade
TPMI_DH_PARENT+	@parentHandle	Handle of a transient storage key, a persistent storage key, TPM_RH_ENDORSEMENT, TPM_RH_OWNER, TPM_RH_PLATFORM+{PP}, or TPM_RH_NULL Auth Index: 1 Auth Role: USER
TPM2B_SENSITIVE_CREATE	inSensitive	the sensitive data, see TPM 2.0 Part 1 Sensitive Values
TPM2B_TEMPLATE	inPublic	the public template

Table 36 — TPM2_CreateLoaded Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM_HANDLE	objectHandle	handle of type TPM_HT_TRANSIENT for created object
TPM2B_PRIVATE	outPrivate	the sensitive area of the object (optional)
TPM2B_PUBLIC	outPublic	the public portion of the created object
TPM2B_NAME	name	the name of the created object

 Copyright © TCG 2006-2020

12.9.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "CreateLoaded_fp.h"
3  #if CC_CreateLoaded  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*sensitiveDataOrigin* is CLEAR when *sensitive.data* is an Empty Buffer; *fixedTPM*, *fixedParent*, or *encryptedDuplication* attributes are inconsistent between themselves or with those of the parent object; inconsistent *restricted*, *decrypt* and *sign* attributes; attempt to inject sensitive data for an asymmetric key; attempt to create a symmetric cipher key that is not a decryption key
TPM_RC_KDF	incorrect KDF specified for decrypting keyed hash object
TPM_RC_KEY	the value of a provided symmetric key is not allowed
TPM_RC_OBJECT_MEMORY	there is no free slot for the object
TPM_RC_SCHEME	inconsistent attributes *decrypt*, *sign*, *restricted* and key's scheme ID; or hash algorithm is inconsistent with the scheme ID for keyed hash object
TPM_RC_SIZE	size of public authorization policy or sensitive authorization value does not match digest size of the name algorithm sensitive data size for the keyed hash object is larger than is allowed for the scheme
TPM_RC_SYMMETRIC	a storage key with no symmetric algorithm specified; or non-storage key with symmetric algorithm different from TPM_ALG_NULL
TPM_RC_TYPE	cannot create the object of the indicated type (usually only occurs if trying to derive an RSA key).

```
4  TPM_RC
5  TPM2_CreateLoaded(
6      CreateLoaded_In     *in,            // IN: input parameter list
7      CreateLoaded_Out    *out            // OUT: output parameter list
8      )
9  {
10     TPM_RC                         result = TPM_RC_SUCCESS;
11     OBJECT                        *parent = HandleToObject(in->parentHandle);
12     OBJECT                        *newObject;
13     BOOL                           derivation;
14     TPMT_PUBLIC                   *publicArea;
15     RAND_STATE                     randState;
16     RAND_STATE                    *rand = &randState;
17     TPMS_DERIVE                    labelContext;
18
19 // Input Validation
20
21     // How the public area is unmarshaled is determined by the parent, so
22     // see if parent is a derivation parent
23     derivation = (parent != NULL && parent->attributes.derivation);
24
25     // If the parent is an object, then make sure that it is either a parent or
26     // derivation parent
27     if(parent != NULL && !parent->attributes.isParent && !derivation)
28         return TPM_RCS_TYPE + RC_CreateLoaded_parentHandle;
29
30     // Get a spot in which to create the newObject
31     newObject = FindEmptyObjectSlot(&out->objectHandle);
32     if(newObject == NULL)
33         return TPM_RC_OBJECT_MEMORY;
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

```
34
35      // Do this to save typing
36      publicArea = &newObject->publicArea;
37
38      // Unmarshal the template into the object space. TPM2_Create() and
39      // TPM2_CreatePrimary() have the publicArea unmarshaled by CommandDispatcher.
40      // This command is different because of an unfortunate property of the
41      // unique field of an ECC key. It is a structure rather than a single TPM2B. If
42      // if had been a TPM2B, then the label and context could be within a TPM2B and
43      // unmarshaled like other public areas. Since it is not, this command needs its
44      // on template that is a TPM2B that is unmarshaled as a BYTE array with a
45      // its own unmarshal function.
46      result = UnmarshalToPublic(publicArea, &in->inPublic, derivation,
47                                 &labelContext);
48      if(result != TPM_RC_SUCCESS)
49          return result + RC_CreateLoaded_inPublic;
50
51      // Validate that the authorization size is appropriate
52      if(!AdjustAuthSize(&in->inSensitive.sensitive.userAuth, publicArea->nameAlg))
53          return TPM_RCS_SIZE + RC_CreateLoaded_inSensitive;
54
55      // Command output
56      if(derivation)
57      {
58          TPMT_KEYEDHASH_SCHEME       *scheme;
59          scheme = &parent->publicArea.parameters.keyedHashDetail.scheme;
60
61          // SP800-108 is the only KDF supported by this implementation and there is
62          // no default hash algorithm.
63          pAssert(scheme->details.xor.hashAlg != TPM_ALG_NULL
64                  && scheme->details.xor.kdf == TPM_ALG_KDF1_SP800_108);
65          // Don't derive RSA keys
66          if(publicArea->type == ALG_RSA_VALUE)
67              return TPM_RCS_TYPE + RC_CreateLoaded_inPublic;
68          // sensitiveDataOrigin has to be CLEAR in a derived object. Since this
69          // is specific to a derived object, it is checked here.
70          if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT,
71                          sensitiveDataOrigin))
72              return TPM_RCS_ATTRIBUTES;
73          // Check the reset of the attributes
74          result = PublicAttributesValidation(parent, publicArea);
75          if(result != TPM_RC_SUCCESS)
76              return RcSafeAddToResult(result, RC_CreateLoaded_inPublic);
77          // Process the template and sensitive areas to get the actual 'label' and
78          // 'context' values to be used for this derivation.
79          result = SetLabelAndContext(&labelContext, &in->inSensitive.sensitive.data);
80          if(result != TPM_RC_SUCCESS)
81              return result;
82          // Set up the KDF for object generation
83          DRBG_InstantiateSeededKdf((KDF_STATE *)rand,
84                                    scheme->details.xor.hashAlg,
85                                    scheme->details.xor.kdf,
86                                    &parent->sensitive.sensitive.bits.b,
87                                    &labelContext.label.b,
88                                    &labelContext.context.b,
89                                    MAX_DERIVATION_BITS);
90          // Clear the sensitive size so that the creation functions will not try
91          // to use this value.
92          in->inSensitive.sensitive.data.t.size = 0;
93      }
94      else
95      {
96          // Check attributes in input public area. CreateChecks() checks the things
97          // that are unique to creation and then validates the attributes and values
98          // that are common to create and load.
99          result = CreateChecks(parent, publicArea,
```

```
100                                 in->inSensitive.sensitive.data.t.size);
101             if(result != TPM_RC_SUCCESS)
102                 return RcSafeAddToResult(result, RC_CreateLoaded_inPublic);
103             // Creating a primary object
104             if(parent == NULL)
105             {
106                 TPM2B_NAME              name;
107                 newObject->attributes.primary = SET;
108                 if(in->parentHandle == TPM_RH_ENDORSEMENT)
109                     newObject->attributes.epsHierarchy = SET;
110                 // If so, use the primary seed and the digest of the template
111                 // to seed the DRBG
112                 result = DRBG_InstantiateSeeded((DRBG_STATE *)rand,
113                                        &HierarchyGetPrimarySeed(in->parentHandle)->b,
114                                        PRIMARY_OBJECT_CREATION,
115                                        (TPM2B *)PublicMarshalAndComputeName(publicArea,
116                                                                             &name),
117                                        &in->inSensitive.sensitive.data.b);
118                 if(result != TPM_RC_SUCCESS)
119                     return result;
120             }
121             else
122             {
123                // This is an ordinary object so use the normal random number generator
124                 rand = NULL;
125             }
126         }
127     // Internal data update
128         // Create the object
129         result = CryptCreateObject(newObject, &in->inSensitive.sensitive, rand);
130         if(result != TPM_RC_SUCCESS)
131             return result;
132         // if this is not a Primary key and not a derived key, then return the sensitive
133         // area
134         if(parent != NULL && !derivation)
135             // Prepare output private data from sensitive
136             SensitiveToPrivate(&newObject->sensitive, &newObject->name,
137                                parent, newObject->publicArea.nameAlg,
138                                &out->outPrivate);
139         else
140             out->outPrivate.t.size = 0;
141         // Set the remaining return values
142         out->outPublic.publicArea = newObject->publicArea;
143         out->name = newObject->name;
144         // Set the remaining attributes for a loaded object
145         ObjectSetLoadedAttributes(newObject, in->parentHandle);
146
147         return result;
148     }
149     #endif // CC_CreateLoaded
```

13 Duplication Commands

13.1 TPM2_Duplicate

13.1.1 General Description

This command duplicates a loaded object so that it may be used in a different hierarchy. The new parent key for the duplicate may be on the same or different TPM or TPM_RH_NULL. Only the public area of *newParentHandle* is required to be loaded.

NOTE 1 Since the new parent may only be extant on a different TPM, it is likely that the new parent's sensitive area could not be loaded in the TPM from which *objectHandle* is being duplicated.

If *encryptedDuplication* is SET in the object being duplicated, then the TPM shall return TPM_RC_SYMMETRIC if *symmetricAlg.algorithm* is TPM_ALG_NULL or TPM_RC_HIERARCHY if *newParentHandle* is TPM_RH_NULL.

The authorization for this command shall be with a policy session.

If *fixedParent* of *objectHandle→attributes* is SET, the TPM shall return TPM_RC_ATTRIBUTES. If *objectHandle→nameAlg* is TPM_ALG_NULL, the TPM shall return TPM_RC_TYPE.

The *policySession→commandCode* parameter in the policy session is required to be TPM_CC_Duplicate to indicate that authorization for duplication has been provided. This indicates that the policy that is being used is a policy that is for duplication, and not a policy that would approve another use. That is, authority to use an object does not grant authority to duplicate the object.

The policy is likely to include cpHash in order to restrict where duplication can occur. If TPM2_PolicyCpHash() has been executed as part of the policy, the *policySession→cpHash* is compared to the cpHash of the command.

If TPM2_PolicyDuplicationSelect() has been executed as part of the policy, the *policySession→nameHash* is compared to

$$\mathbf{H}_{policyAlg}(objectHandle{\rightarrow}Name \mid\mid newParentHandle{\rightarrow}Name) \tag{2}$$

If the compared hashes are not the same, then the TPM shall return TPM_RC_POLICY_FAIL.

NOTE 2 It is allowed that policySesion→nameHash and policySession→cpHash share the same memory space.

NOTE 3 A duplication policy is not required to have either TPM2_PolicyDuplicationSelect() or TPM2_PolicyCpHash() as part of the policy. If neither is present, then the duplication policy may be satisfied with a policy that only contains TPM2_PolicyCommandCode(*code* = TPM_CC_Duplicate).

The TPM shall follow the process of encryption defined in the "Duplication" subclause of "Protected Storage Hierarchy" in TPM 2.0 Part 1.

 Copyright © TCG 2006-2020

13.1.2 Command and Response

Table 37 — TPM2_Duplicate Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Duplicate
TPMI_DH_OBJECT	@objectHandle	loaded object to duplicate Auth Index: 1 Auth Role: DUP
TPMI_DH_OBJECT+	newParentHandle	shall reference the public area of an asymmetric key Auth Index: None
TPM2B_DATA	encryptionKeyIn	optional symmetric encryption key The size for this key is set to zero when the TPM is to generate the key. This parameter may be encrypted.
TPMT_SYM_DEF_OBJECT+	symmetricAlg	definition for the symmetric algorithm to be used for the inner wrapper may be TPM_ALG_NULL if no inner wrapper is applied

Table 38 — TPM2_Duplicate Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DATA	encryptionKeyOut	If the caller provided an encryption key or if *symmetricAlg* was TPM_ALG_NULL, then this will be the Empty Buffer; otherwise, it shall contain the TPM-generated, symmetric encryption key for the inner wrapper.
TPM2B_PRIVATE	duplicate	private area that may be encrypted by *encryptionKeyIn*; and may be doubly encrypted
TPM2B_ENCRYPTED_SECRET	outSymSeed	seed protected by the asymmetric algorithms of new parent (NP)

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

13.1.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "Duplicate_fp.h"
3  #if CC_Duplicate  // Conditional expansion of this file
4  #include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	key to duplicate has *fixedParent* SET
TPM_RC_HASH	for an RSA key, the *nameAlg* digest size for the *newParent* is not compatible with the key size
TPM_RC_HIERARCHY	*encryptedDuplication* is SET and *newParentHandle* specifies Null Hierarchy
TPM_RC_KEY	*newParentHandle* references invalid ECC key (public point not on the curve)
TPM_RC_SIZE	input encryption key size does not match the size specified in symmetric algorithm
TPM_RC_SYMMETRIC	*encryptedDuplication* is SET but no symmetric algorithm is provided
TPM_RC_TYPE	*newParentHandle* is neither a storage key nor TPM_RH_NULL; or the object has a NULL *nameAlg*
TPM_RC_VALUE	for an RSA *newParent*, the sizes of the digest and the encryption key are too large to be OAEP encoded

```
5  TPM_RC
6  TPM2_Duplicate(
7      Duplicate_In    *in,            // IN: input parameter list
8      Duplicate_Out   *out            // OUT: output parameter list
9      )
10 {
11     TPM_RC                  result = TPM_RC_SUCCESS;
12     TPMT_SENSITIVE          sensitive;
13
14     UINT16                  innerKeySize = 0; // encrypt key size for inner wrap
15
16     OBJECT                  *object;
17     OBJECT                  *newParent;
18     TPM2B_DATA              data;
19
20 // Input Validation
21
22     // Get duplicate object pointer
23     object = HandleToObject(in->objectHandle);
24     // Get new parent
25     newParent = HandleToObject(in->newParentHandle);
26
27     // duplicate key must have fixParent bit CLEAR.
28     if(IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, fixedParent))
29         return TPM_RCS_ATTRIBUTES + RC_Duplicate_objectHandle;
30
31     // Do not duplicate object with NULL nameAlg
32     if(object->publicArea.nameAlg == TPM_ALG_NULL)
33         return TPM_RCS_TYPE + RC_Duplicate_objectHandle;
34
35     // new parent key must be a storage object or TPM_RH_NULL
36     if(in->newParentHandle != TPM_RH_NULL
37        && !ObjectIsStorage(in->newParentHandle))
38         return TPM_RCS_TYPE + RC_Duplicate_newParentHandle;
```

```
39
40      // If the duplicated object has encryptedDuplication SET, then there must be
41      // an inner wrapper and the new parent may not be TPM_RH_NULL
42      if(IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT,
43                      encryptedDuplication))
44      {
45          if(in->symmetricAlg.algorithm == TPM_ALG_NULL)
46              return TPM_RCS_SYMMETRIC + RC_Duplicate_symmetricAlg;
47          if(in->newParentHandle == TPM_RH_NULL)
48              return TPM_RCS_HIERARCHY + RC_Duplicate_newParentHandle;
49      }
50
51      if(in->symmetricAlg.algorithm == TPM_ALG_NULL)
52      {
53          // if algorithm is TPM_ALG_NULL, input key size must be 0
54          if(in->encryptionKeyIn.t.size != 0)
55              return TPM_RCS_SIZE + RC_Duplicate_encryptionKeyIn;
56      }
57      else
58      {
59          // Get inner wrap key size
60          innerKeySize = in->symmetricAlg.keyBits.sym;
61
62          // If provided the input symmetric key must match the size of the algorithm
63          if(in->encryptionKeyIn.t.size != 0
64             && in->encryptionKeyIn.t.size != (innerKeySize + 7) / 8)
65              return TPM_RCS_SIZE + RC_Duplicate_encryptionKeyIn;
66      }
67
68  // Command Output
69
70      if(in->newParentHandle != TPM_RH_NULL)
71      {
72          // Make encrypt key and its associated secret structure.  A TPM_RC_KEY
73          // error may be returned at this point
74          out->outSymSeed.t.size = sizeof(out->outSymSeed.t.secret);
75          result = CryptSecretEncrypt(newParent, DUPLICATE_STRING, &data,
76                                      &out->outSymSeed);
77          if(result != TPM_RC_SUCCESS)
78              return result;
79      }
80      else
81      {
82          // Do not apply outer wrapper
83          data.t.size = 0;
84          out->outSymSeed.t.size = 0;
85      }
86
87      // Copy sensitive area
88      sensitive = object->sensitive;
89
90      // Prepare output private data from sensitive.
91      // Note: If there is no encryption key, one will be provided by
92      // SensitiveToDuplicate(). This is why the assignment of encryptionKeyIn to
93      // encryptionKeyOut will work properly and is not conditional.
94      SensitiveToDuplicate(&sensitive, &object->name.b, newParent,
95                           object->publicArea.nameAlg, &data.b,
96                           &in->symmetricAlg, &in->encryptionKeyIn,
97                           &out->duplicate);
98
99      out->encryptionKeyOut = in->encryptionKeyIn;
100
101     return TPM_RC_SUCCESS;
102 }
103 #endif // CC_Duplicate
```

 Copyright © TCG 2006-2020

13.2 TPM2_Rewrap

13.2.1 General Description

This command allows the TPM to serve in the role as a Duplication Authority. If proper authorization for use of the *oldParent* is provided, then an HMAC key and a symmetric key are recovered from *inSymSeed* and used to integrity check and decrypt *inDuplicate.* A new protection seed value is generated according to the methods appropriate for *newParent* and the blob is re-encrypted and a new integrity value is computed. The re-encrypted blob is returned in *outDuplicate* and the symmetric key returned in *outSymKey*.

In the rewrap process, L is “DUPLICATE” (see TPM 2.0 Part 1, *Terms and Definitions*).

If *inSymSeed* has a zero length, then *oldParent* is required to be TPM_RH_NULL and no decryption of *inDuplicate* takes place.

If *newParent* is TPM_RH_NULL, then no encryption is performed on *outDuplicate. outSymSeed* will have a zero length. See TPM 2.0 Part 2 *encryptedDuplication*.

 Copyright © TCG 2006-2020

13.2.2 Command and Response

Table 39 — TPM2_Rewrap Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Rewrap
TPMI_DH_OBJECT+	@oldParent	parent of object Auth Index: 1 Auth Role: User
TPMI_DH_OBJECT+	newParent	new parent of the object Auth Index: None
TPM2B_PRIVATE	inDuplicate	an object encrypted using symmetric key derived from *inSymSeed*
TPM2B_NAME	name	the Name of the object being rewrapped
TPM2B_ENCRYPTED_SECRET	inSymSeed	the seed for the symmetric key and HMAC key needs *oldParent* private key to recover the seed and generate the symmetric key

Table 40 — TPM2_Rewrap Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PRIVATE	outDuplicate	an object encrypted using symmetric key derived from *outSymSeed*
TPM2B_ENCRYPTED_SECRET	outSymSeed	seed for a symmetric key protected by *newParent* asymmetric key

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

13.2.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "Rewrap_fp.h"
3  #if CC_Rewrap  // Conditional expansion of this file
4  #include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*newParent* is not a decryption key
TPM_RC_HANDLE	*oldParent* does not consistent with *inSymSeed*
TPM_RC_INTEGRITY	the integrity check of *inDuplicate* failed
TPM_RC_KEY	for an ECC key, the public key is not on the curve of the curve ID
TPM_RC_KEY_SIZE	the decrypted input symmetric key size does not matches the symmetric algorithm key size of *oldParent*
TPM_RC_TYPE	*oldParent* is not a storage key, or '*newParent* is not a storage key
TPM_RC_VALUE	for an '*oldParent*; RSA key, the data to be decrypted is greater than the public exponent
errors	errors during unmarshaling the input encrypted buffer to a ECC public key, or unmarshal the private buffer to sensitive

```
5  TPM_RC
6  TPM2_Rewrap(
7      Rewrap_In       *in,            // IN: input parameter list
8      Rewrap_Out      *out            // OUT: output parameter list
9      )
10 {
11     TPM_RC                  result = TPM_RC_SUCCESS;
12     TPM2B_DATA              data;               // symmetric key
13     UINT16                  hashSize = 0;
14     TPM2B_PRIVATE           privateBlob;        // A temporary private blob
15                                                 // to transit between old
16                                                 // and new wrappers
17 // Input Validation
18     if((in->inSymSeed.t.size == 0 && in->oldParent != TPM_RH_NULL)
19        || (in->inSymSeed.t.size != 0 && in->oldParent == TPM_RH_NULL))
20         return TPM_RCS_HANDLE + RC_Rewrap_oldParent;
21     if(in->oldParent != TPM_RH_NULL)
22     {
23         OBJECT              *oldParent = HandleToObject(in->oldParent);
24
25         // old parent key must be a storage object
26         if(!ObjectIsStorage(in->oldParent))
27             return TPM_RCS_TYPE + RC_Rewrap_oldParent;
28         // Decrypt input secret data via asymmetric decryption.  A
29         // TPM_RC_VALUE, TPM_RC_KEY or unmarshal errors may be returned at this
30         // point
31         result = CryptSecretDecrypt(oldParent, NULL, DUPLICATE_STRING,
32                                     &in->inSymSeed, &data);
33         if(result != TPM_RC_SUCCESS)
34             return TPM_RCS_VALUE + RC_Rewrap_inSymSeed;
35         // Unwrap Outer
36         result = UnwrapOuter(oldParent, &in->name.b,
37                              oldParent->publicArea.nameAlg, &data.b,
38                              FALSE,
39                              in->inDuplicate.t.size, in->inDuplicate.t.buffer);
40         if(result != TPM_RC_SUCCESS)
41             return RcSafeAddToResult(result, RC_Rewrap_inDuplicate);
```

```
42          // Copy unwrapped data to temporary variable, remove the integrity field
43          hashSize = sizeof(UINT16) +
44              CryptHashGetDigestSize(oldParent->publicArea.nameAlg);
45          privateBlob.t.size = in->inDuplicate.t.size - hashSize;
46          pAssert(privateBlob.t.size <= sizeof(privateBlob.t.buffer));
47          MemoryCopy(privateBlob.t.buffer, in->inDuplicate.t.buffer + hashSize,
48                       privateBlob.t.size);
49      }
50      else
51      {
52          // No outer wrap from input blob.  Direct copy.
53          privateBlob = in->inDuplicate;
54      }
55      if(in->newParent != TPM_RH_NULL)
56      {
57          OBJECT          *newParent;
58          newParent = HandleToObject(in->newParent);
59
60          // New parent must be a storage object
61          if(!ObjectIsStorage(in->newParent))
62              return TPM_RCS_TYPE + RC_Rewrap_newParent;
63          // Make new encrypt key and its associated secret structure.  A
64          // TPM_RC_VALUE error may be returned at this point if RSA algorithm is
65          // enabled in TPM
66          out->outSymSeed.t.size = sizeof(out->outSymSeed.t.secret);
67          result = CryptSecretEncrypt(newParent, DUPLICATE_STRING, &data,
68                                      &out->outSymSeed);
69          if(result != TPM_RC_SUCCESS)
70              return result;
71          // Copy temporary variable to output, reserve the space for integrity
72          hashSize = sizeof(UINT16) +
73              CryptHashGetDigestSize(newParent->publicArea.nameAlg);
74   // Make sure that everything fits into the output buffer
75   // Note: this is mostly only an issue if there was no outer wrapper on
76   // 'inDuplicate'. It could be as large as a TPM2B_PRIVATE buffer. If we add
77   // a digest for an outer wrapper, it won't fit anymore.
78          if((privateBlob.t.size + hashSize) > sizeof(out->outDuplicate.t.buffer))
79              return TPM_RCS_VALUE + RC_Rewrap_inDuplicate;
80  // Command output
81          out->outDuplicate.t.size = privateBlob.t.size;
82          pAssert(privateBlob.t.size
83                  <= sizeof(out->outDuplicate.t.buffer) - hashSize);
84          MemoryCopy(out->outDuplicate.t.buffer + hashSize, privateBlob.t.buffer,
85                     privateBlob.t.size);
86          // Produce outer wrapper for output
87          out->outDuplicate.t.size = ProduceOuterWrap(newParent, &in->name.b,
88                                                      newParent->publicArea.nameAlg,
89                                                      &data.b,
90                                                      FALSE,
91                                                      out->outDuplicate.t.size,
92                                                      out->outDuplicate.t.buffer);
93      }
94      else  // New parent is a null key so there is no seed
95      {
96          out->outSymSeed.t.size = 0;
97
98          // Copy privateBlob directly
99          out->outDuplicate = privateBlob;
100     }
101     return TPM_RC_SUCCESS;
102 }
103 #endif // CC_Rewrap
```

13.3 TPM2_Import

13.3.1 General Description

This command allows an object to be encrypted using the symmetric encryption values of a Storage Key. After encryption, the object may be loaded and used in the new hierarchy. The imported object (*duplicate*) may be singly encrypted, multiply encrypted, or unencrypted.

If *fixedTPM* or *fixedParent* is SET in *objectPublic*, the TPM shall return TPM_RC_ATTRIBUTES.

If *encryptedDuplication* is SET in the object referenced by *parentHandle* and *encryptedDuplication* is CLEAR in *objectPublic,* the TPM may return TPM_RC_ATTRIBUTES.

If encryptedDuplication is SET in objectPublic, then *inSymSeed* and *encryptionKey* shall not be Empty buffers (TPM_RC_ATTRIBUTES). Recovery of the sensitive data of the object occurs in the TPM in a multi--step process in the following order:

a) If *inSymSeed* has a non-zero size:

   1) The asymmetric parameters and private key of *parentHandle* are used to recover the seed used in the creation of the HMAC key and encryption keys used to protect the duplication blob.

      NOTE 1 When recovering the seed from *inSymSeed*, *L* is "DUPLICATE".

   2) The integrity value in *duplicate.buffer.integrityOuter* is used to verify the integrity of the data blob, which is the remainder of *duplicate.buffer* (TPM_RC_INTEGRITY).

      NOTE 2 The data blob will contain a TPMT_SENSITIVE and may contain a TPM2B_DIGEST for the *innerIntegrity.*

   3) The symmetric key recovered in 1) is used to decrypt the data blob.

      NOTE 3 Checking the integrity before the data is used prevents attacks on the sensitive area by fuzzing the data and looking at the differences in the response codes.

b) If *encryptionKey* is not an Empty Buffer:

   1) Use *encryptionKey* to decrypt the inner blob.

   2) Use the TPM2B_DIGEST at the start of the inner blob to verify the integrity of the inner blob (TPM_RC_INTEGRITY).

c) Unmarshal the sensitive area

NOTE 4 It is not necessary to validate that the sensitive area data is cryptographically bound to the public area other than that the Name of the public area is included in the HMAC. However, if the binding is not validated by this command, the binding must be checked each time the object is loaded. For an object that is imported under a parent with *fixedTPM* SET, binding need only be checked at import. If the parent has *fixedTPM* CLEAR, then the binding needs to be checked each time the object is loaded, or before the TPM performs an operation for which the binding affects the outcome of the operation (for example, TPM2_PolicySigned() or TPM2_Certify()).

Similarly, if the new parent's *fixedTPM* is set, the *encryptedDuplication* state need only be checked at import.

If the new parent is not *fixedTPM*, then that object will be loadable on any TPM (including SW versions) on which the new parent exists. This means that, each time an object is loaded under a parent that is not *fixedTPM*, it is necessary to validate all of the properties of that object. If the parent is *fixedTPM*, then the new private blob is integrity protected by the TPM that "owns" the parent. So, it is sufficient to validate the object's properties (attribute and public-private binding) on import and not again.

If a weak symmetric key is being imported, the TPM shall return TPM_RC_KEY.

 Copyright © TCG 2006-2020

After integrity checks and decryption, the TPM will create a new symmetrically encrypted private area using the encryption key of the parent.

NOTE 5 The symmetric re-encryption is the normal integrity generation and symmetric encryption applied to a child object.

NOTE 6 Revision 01.16 of this specification required the ECC private key in *duplicate* to be padded.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

13.3.2 Command and Response

Table 41 — TPM2_Import Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Import
TPMI_DH_OBJECT	@parentHandle	the handle of the new parent for the object Auth Index: 1 Auth Role: USER
TPM2B_DATA	encryptionKey	the optional symmetric encryption key used as the inner wrapper for *duplicate* If *symmetricAlg* is TPM_ALG_NULL, then this parameter shall be the Empty Buffer.
TPM2B_PUBLIC	objectPublic	the public area of the object to be imported This is provided so that the integrity value for *duplicate* and the object attributes can be checked. NOTE Even if the integrity value of the object is not checked on input, the object Name is required to create the integrity value for the imported object.
TPM2B_PRIVATE	duplicate	the symmetrically encrypted duplicate object that may contain an inner symmetric wrapper
TPM2B_ENCRYPTED_SECRET	inSymSeed	the seed for the symmetric key and HMAC key *inSymSeed* is encrypted/encoded using the algorithms of *newParent*.
TPMT_SYM_DEF_OBJECT+	symmetricAlg	definition for the symmetric algorithm to use for the inner wrapper If this algorithm is TPM_ALG_NULL, no inner wrapper is present and *encryptionKey* shall be the Empty Buffer.

Table 42 — TPM2_Import Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PRIVATE	outPrivate	the sensitive area encrypted with the symmetric key of *parentHandle*

 Copyright © TCG 2006-2020

13.3.3 Detailed Actions

```
1 #include "Tpm.h"
2 #include "Import_fp.h"
3 #if CC_Import  // Conditional expansion of this file
4 #include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*FixedTPM* and *fixedParent* of *objectPublic* are not both CLEAR; or *inSymSeed* is nonempty and *parentHandle* does not reference a decryption key; or *objectPublic* and *parentHandle* have incompatible or inconsistent attributes; or *encrytpedDuplication* is SET in *objectPublic* but the inner or outer wrapper is missing. Note that if the TPM provides parameter values, the parameter number will indicate *symmetricKey* (missing inner wrapper) or *inSymSeed* (missing outer wrapper)
TPM_RC_BINDING	*duplicate* and *objectPublic* are not cryptographically bound
TPM_RC_ECC_POINT	*inSymSeed* is nonempty and ECC point in *inSymSeed* is not on the curve
TPM_RC_HASH	*objectPublic* does not have a valid *nameAlg*
TPM_RC_INSUFFICIENT	*inSymSeed* is nonempty and failed to retrieve ECC point from the secret; or unmarshaling sensitive value from *duplicate* failed the result of *inSymSeed* decryption
TPM_RC_INTEGRITY	*duplicate* integrity is broken
TPM_RC_KDF	*objectPublic* representing decrypting keyed hash object specifies invalid KDF
TPM_RC_KEY	inconsistent parameters of *objectPublic*; or *inSymSeed* is nonempty and *parentHandle* does not reference a key of supported type; or invalid key size in *objectPublic* representing an asymmetric key
TPM_RC_NO_RESULT	*inSymSeed* is nonempty and multiplication resulted in ECC point at infinity
TPM_RC_OBJECT_MEMORY	no available object slot
TPM_RC_SCHEME	inconsistent attributes *decrypt*, *sign*, *restricted* and key's scheme ID in *objectPublic*; or hash algorithm is inconsistent with the scheme ID for keyed hash object
TPM_RC_SIZE	*authPolicy* size does not match digest size of the name algorithm in *objectPublic*; or *symmetricAlg* and *encryptionKey* have different sizes; or *inSymSeed* is nonempty and it size is not consistent with the type of *parentHandle*; or unmarshaling sensitive value from *duplicate* failed
TPM_RC_SYMMETRIC	*objectPublic* is either a storage key with no symmetric algorithm or a non-storage key with symmetric algorithm different from TPM_ALG_NULL
TPM_RC_TYPE	unsupported type of *objectPublic*; or *parentHandle* is not a storage key; or only the public portion of *parentHandle* is loaded; or *objectPublic* and *duplicate* are of different types
TPM_RC_VALUE	nonempty *inSymSeed* and its numeric value is greater than the modulus of the key referenced by *parentHandle* or *inSymSeed* is larger than the size of the digest produced by the name algorithm of the symmetric key referenced by *parentHandle*

```
5 TPM_RC
```

 Copyright © TCG 2006-2020

```
6    TPM2_Import(
7        Import_In       *in,            // IN: input parameter list
8        Import_Out      *out            // OUT: output parameter list
9        )
10   {
11       TPM_RC                    result = TPM_RC_SUCCESS;
12       OBJECT                   *parentObject;
13       TPM2B_DATA                data;                 // symmetric key
14       TPMT_SENSITIVE            sensitive;
15       TPM2B_NAME                name;
16       TPMA_OBJECT               attributes;
17       UINT16                    innerKeySize = 0;     // encrypt key size for inner
18                                                       // wrapper
19
20   // Input Validation
21       // to save typing
22       attributes = in->objectPublic.publicArea.objectAttributes;
23       // FixedTPM and fixedParent must be CLEAR
24       if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM)
25          || IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent))
26           return TPM_RCS_ATTRIBUTES + RC_Import_objectPublic;
27
28       // Get parent pointer
29       parentObject = HandleToObject(in->parentHandle);
30
31       if(!ObjectIsParent(parentObject))
32           return TPM_RCS_TYPE + RC_Import_parentHandle;
33
34       if(in->symmetricAlg.algorithm != TPM_ALG_NULL)
35       {
36           // Get inner wrap key size
37           innerKeySize = in->symmetricAlg.keyBits.sym;
38           // Input symmetric key must match the size of algorithm.
39           if(in->encryptionKey.t.size != (innerKeySize + 7) / 8)
40               return TPM_RCS_SIZE + RC_Import_encryptionKey;
41       }
42       else
43       {
44           // If input symmetric algorithm is NULL, input symmetric key size must
45           // be 0 as well
46           if(in->encryptionKey.t.size != 0)
47               return TPM_RCS_SIZE + RC_Import_encryptionKey;
48           // If encryptedDuplication is SET, then the object must have an inner
49           // wrapper
50           if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, encryptedDuplication))
51               return TPM_RCS_ATTRIBUTES + RC_Import_encryptionKey;
52       }
53       // See if there is an outer wrapper
54       if(in->inSymSeed.t.size != 0)
55       {
56           // in->inParentHandle is a parent, but in order to decrypt an outer wrapper,
57           // it must be able to do key exchange and a symmetric key can't do that.
58           if(parentObject->publicArea.type == TPM_ALG_SYMCIPHER)
59               return TPM_RCS_TYPE + RC_Import_parentHandle;
60
61           // Decrypt input secret data via asymmetric decryption. TPM_RC_ATTRIBUTES,
62           // TPM_RC_ECC_POINT, TPM_RC_INSUFFICIENT, TPM_RC_KEY, TPM_RC_NO_RESULT,
63           // TPM_RC_SIZE, TPM_RC_VALUE may be returned at this point
64           result = CryptSecretDecrypt(parentObject, NULL, DUPLICATE_STRING,
65                                       &in->inSymSeed, &data);
66           pAssert(result != TPM_RC_BINDING);
67           if(result != TPM_RC_SUCCESS)
68               return RcSafeAddToResult(result, RC_Import_inSymSeed);
69       }
70       else
71       {
```

 Copyright © TCG 2006-2020

```
72             // If encrytpedDuplication is set, then the object must have an outer
73             // wrapper
74             if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, encryptedDuplication))
75                 return TPM_RCS_ATTRIBUTES + RC_Import_inSymSeed;
76             data.t.size = 0;
77         }
78         // Compute name of object
79         PublicMarshalAndComputeName(&(in->objectPublic.publicArea), &name);
80         if(name.t.size == 0)
81             return TPM_RCS_HASH + RC_Import_objectPublic;
82
83         // Retrieve sensitive from private.
84         // TPM_RC_INSUFFICIENT, TPM_RC_INTEGRITY, TPM_RC_SIZE may be returned here.
85         result = DuplicateToSensitive(&in->duplicate.b, &name.b, parentObject,
86                                       in->objectPublic.publicArea.nameAlg,
87                                       &data.b, &in->symmetricAlg,
88                                       &in->encryptionKey.b, &sensitive);
89         if(result != TPM_RC_SUCCESS)
90             return RcSafeAddToResult(result, RC_Import_duplicate);
91
92         // If the parent of this object has fixedTPM SET, then validate this
93         // object as if it were being loaded so that validation can be skipped
94         // when it is actually loaded.
95         if(IS_ATTRIBUTE(parentObject->publicArea.objectAttributes, TPMA_OBJECT, fixedTPM))
96         {
97             result = ObjectLoad(NULL, NULL, &in->objectPublic.publicArea,
98                                 &sensitive, RC_Import_objectPublic, RC_Import_duplicate,
99                                 NULL);
100        }
101    // Command output
102        if(result == TPM_RC_SUCCESS)
103        {
104            // Prepare output private data from sensitive
105            SensitiveToPrivate(&sensitive, &name, parentObject,
106                               in->objectPublic.publicArea.nameAlg,
107                               &out->outPrivate);
108        }
109        return result;
110    }
111    #endif // CC_Import
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

14 Asymmetric Primitives

14.1 Introduction

The commands in this clause provide low-level primitives for access to the asymmetric algorithms implemented in the TPM. Many of these commands are only allowed if the asymmetric key is an unrestricted key.

14.2 TPM2_RSA_Encrypt

14.2.1 General Description

This command performs RSA encryption using the indicated padding scheme according to IETF RFC 8017. If the *scheme* of *keyHandle* is TPM_ALG_NULL, then the caller may use *inScheme* to specify the padding scheme. If *scheme* of keyHandle is not TPM_ALG_NULL, then *inScheme* shall either be TPM_ALG_NULL or be the same as *scheme* (TPM_RC_SCHEME).

The key referenced by *keyHandle* is required to be an RSA key (TPM_RC_KEY).

The three types of allowed padding are:

1) TPM_ALG_OAEP – Data is OAEP padded as described in 7.1 of IETF RFC 8017 (PKCS#1). The only supported mask generation is MGF1.

2) TPM_ALG_RSAES – Data is padded as described in 7.2 of IETF RFC 8017 (PKCS#1).

3) TPM_ALG_NULL – Data is not padded by the TPM and the TPM will treat *message* as an unsigned integer and perform a modular exponentiation of *message* using the public exponent of the key referenced by *keyHandle*. This scheme is only used if both the *scheme* in the key referenced by *keyHandle* is TPM_ALG_NULL, and the *inScheme* parameter of the command is TPM_ALG_NULL. The input value cannot be larger than the public modulus of the key referenced by *keyHandle*.

Table 43 — Padding Scheme Selection

keyHandle→scheme	***inScheme***	**padding scheme used**
TPM_ALG_NULL	TPM_ALG_NULL	none
	TPM_ALG_RSAES	RSAES
	TPM_ALG_OAEP	OAEP
TPM_ALG_RSAES	TPM_ALG_NULL	RSAES
	TPM_ALG_RSAES	RSAES
	TPM_ALG_OAEP	error (TPM_RC_SCHEME)
TPM_ALG_OAEP	TPM_ALG_NULL	OAEP
	TPM_ALG_RSAES	error (TPM_RC_SCHEME)
	TPM_ALG_OAEP	OAEP

After padding, the data is RSAEP encrypted according to 5.1.1 of IETF RFC 8017 (PKCS#1).

If *inScheme* is used, and the scheme requires a hash algorithm it may not be TPM_ALG_NULL.

NOTE 1 Because only the public portion of the key needs to be loaded for this command, the caller can manipulate the attributes of the key in any way desired. As a result, the TPM shall not check the consistency of the attributes. The only property checking is that the key is an RSA key and that the padding scheme is supported.

 Copyright © TCG 2006-2020

The *message* parameter is limited in size by the padding scheme according to the following table:

Table 44 — Message Size Limits Based on Padding

Scheme	Maximum Message Length (*mLen*) in Octets	Comments
TPM_ALG_OAEP	$mLen \leq k - 2hLen - 2$	
TPM_ALG_RSAES	$mLen \leq k - 11$	
TPM_ALG_NULL	$mLen \leq k$	The numeric value of the message must be less than the numeric value of the public modulus (n).
NOTES 1) $k :=$ the number of byes in the public modulus 2) $hLen :=$ the number of octets in the digest produced by the hash algorithm used in the process		

The *label* parameter is optional. If provided (*label.size* != 0) then the TPM shall return TPM_RC_VALUE if the last octet in *label* is not zero. The terminating octet of zero is included in the *label* used in the padding scheme.

NOTE 2 If the scheme does not use a label, the TPM will still verify that label is properly formatted if label is present.

NOTE 3 Specifications before version 1.54 stated that *label* is truncated after the first zero octet. Applications should not include embedded zero bytes for compatibility.

The function returns padded and encrypted value *outData*.

The *message* parameter in the command may be encrypted using parameter encryption.

NOTE 4 Only the public area of *keyHandle* is required to be loaded. A public key may be loaded with any desired scheme. If the scheme is to be changed, a different public area must be loaded.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

14.2.2 Command and Response

Table 45 — TPM2_RSA_Encrypt Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit, encrypt, or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_RSA_Encrypt
TPMI_DH_OBJECT	keyHandle	reference to public portion of RSA key to use for encryption Auth Index: None
TPM2B_PUBLIC_KEY_RSA	message	message to be encrypted NOTE 1 The data type was chosen because it limits the overall size of the input to no greater than the size of the largest RSA public key. This may be larger than allowed for *keyHandle*.
TPMT_RSA_DECRYPT+	inScheme	the padding scheme to use if *scheme* associated with *keyHandle* is TPM_ALG_NULL
TPM2B_DATA	label	optional label *L* to be associated with the message Size of the buffer is zero if no label is present NOTE 2 See description of label above.

Table 46 — TPM2_RSA_Encrypt Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PUBLIC_KEY_RSA	outData	encrypted output

 Copyright © TCG 2006-2020

14.2.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "RSA_Encrypt_fp.h"
3  #if CC_RSA_Encrypt  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*decrypt* attribute is not SET in key referenced by *keyHandle*
TPM_RC_KEY	*keyHandle* does not reference an RSA key
TPM_RC_SCHEME	incorrect input scheme, or the chosen scheme is not a valid RSA decrypt scheme
TPM_RC_VALUE	the numeric value of *message* is greater than the public modulus of the key referenced by *keyHandle*, or *label* is not a null-terminated string

```
4  TPM_RC
5  TPM2_RSA_Encrypt(
6      RSA_Encrypt_In      *in,            // IN: input parameter list
7      RSA_Encrypt_Out     *out            // OUT: output parameter list
8      )
9  {
10     TPM_RC                  result;
11     OBJECT                  *rsaKey;
12     TPMT_RSA_DECRYPT        *scheme;
13 // Input Validation
14     rsaKey = HandleToObject(in->keyHandle);
15
16     // selected key must be an RSA key
17     if(rsaKey->publicArea.type != TPM_ALG_RSA)
18         return TPM_RCS_KEY + RC_RSA_Encrypt_keyHandle;
19     // selected key must have the decryption attribute
20     if(!IS_ATTRIBUTE(rsaKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
21         return TPM_RCS_ATTRIBUTES + RC_RSA_Encrypt_keyHandle;
22
23     // Is there a label?
24     if(!IsLabelProperlyFormatted(&in->label.b))
25         return TPM_RCS_VALUE + RC_RSA_Encrypt_label;
26 // Command Output
27     // Select a scheme for encryption
28     scheme = CryptRsaSelectScheme(in->keyHandle, &in->inScheme);
29     if(scheme == NULL)
30         return TPM_RCS_SCHEME + RC_RSA_Encrypt_inScheme;
31
32     // Encryption.  TPM_RC_VALUE, or TPM_RC_SCHEME errors my be returned buy
33     // CryptEncyptRSA.
34     out->outData.t.size = sizeof(out->outData.t.buffer);
35
36     result = CryptRsaEncrypt(&out->outData, &in->message.b, rsaKey, scheme,
37                              &in->label.b, NULL);
38     return result;
39 }
40 #endif // CC_RSA_Encrypt
```


Level 00 Revision 01.59 **Copyright © TCG 2006-2020** November 8, 2019

14.3 TPM2_RSA_Decrypt

14.3.1 General Description

This command performs RSA decryption using the indicated padding scheme according to IETF RFC 8017 ((PKCS#1).

The scheme selection for this command is the same as for TPM2_RSA_Encrypt() and is shown in Table 43.

The key referenced by *keyHandle* shall be an RSA key (TPM_RC_KEY) with *restricted* CLEAR and *decrypt* SET (TPM_RC_ATTRIBUTES).

This command uses the private key of *keyHandle* for this operation and authorization is required.

The TPM will perform a modular exponentiation of ciphertext using the private exponent associated with *keyHandle* (this is described in IETF RFC 8017 (PKCS#1), clause 5.1.2). It will then validate the padding according to the selected scheme. If the padding checks fail, TPM_RC_VALUE is returned. Otherwise, the data is returned with the padding removed. If no padding is used, the returned value is an unsigned integer value that is the result of the modular exponentiation of *cipherText* using the private exponent of *keyHandle*. The returned value may include leading octets zeros so that it is the same size as the public modulus. For the other padding schemes, the returned value will be smaller than the public modulus but will contain all the data remaining after padding is removed and this may include leading zeros if the original encrypted value contained leading zeros.

If a label is used in the padding process of the scheme during encryption, the *label* parameter is required to be present in the decryption process and *label* is required to be the same in both cases. If label is not the same, the decrypt operation is very likely to fail ((TPM_RC_VALUE). If *label* is present (*label.size* != 0), it shall be a byte stream whose last byte is zero or the TPM will return TPM_RC_VALUE.

NOTE 1 The size of *label* includes the terminating null.

The *message* parameter in the response may be encrypted using parameter encryption.

If *inScheme* is used, and the scheme requires a hash algorithm it may not be TPM_ALG_NULL.

If the scheme does not require a label, the value in *label* is not used but the size of the label field is checked for consistency with the indicated data type (TPM2B_DATA). That is, the field may not be larger than allowed for a TPM2B_DATA.

 Copyright © TCG 2006-2020

14.3.2 Command and Response

Table 47 — TPM2_RSA_Decrypt Command

Type	**Name**	**Description**
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_RSA_Decrypt
TPMI_DH_OBJECT	@keyHandle	RSA key to use for decryption Auth Index: 1 Auth Role: USER
TPM2B_PUBLIC_KEY_RSA	cipherText	cipher text to be decrypted NOTE An encrypted RSA data block is the size of the public modulus.
TPMT_RSA_DECRYPT+	inScheme	the padding scheme to use if *scheme* associated with *keyHandle* is TPM_ALG_NULL
TPM2B_DATA	label	label whose association with the message is to be verified

Table 48 — TPM2_RSA_Decrypt Response

Type	**Name**	**Description**
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PUBLIC_KEY_RSA	message	decrypted output

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

14.3.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "RSA_Decrypt_fp.h"
3  #if CC_RSA_Decrypt  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*decrypt* is not SET or if *restricted* is SET in the key referenced by *keyHandle*
TPM_RC_BINDING	The public and private parts of the key are not properly bound
TPM_RC_KEY	*keyHandle* does not reference an unrestricted decrypt key
TPM_RC_SCHEME	incorrect input scheme, or the chosen *scheme* is not a valid RSA decrypt scheme
TPM_RC_SIZE	*cipherText* is not the size of the modulus of key referenced by *keyHandle*
TPM_RC_VALUE	*label* is not a null terminated string or the value of *cipherText* is greater that the modulus of *keyHandle* or the encoding of the data is not valid

```
4  TPM_RC
5  TPM2_RSA_Decrypt(
6      RSA_Decrypt_In      *in,            // IN: input parameter list
7      RSA_Decrypt_Out     *out            // OUT: output parameter list
8      )
9  {
10     TPM_RC                         result;
11     OBJECT                        *rsaKey;
12     TPMT_RSA_DECRYPT              *scheme;
13
14 // Input Validation
15
16     rsaKey = HandleToObject(in->keyHandle);
17
18     // The selected key must be an RSA key
19     if(rsaKey->publicArea.type != TPM_ALG_RSA)
20         return TPM_RCS_KEY + RC_RSA_Decrypt_keyHandle;
21
22     // The selected key must be an unrestricted decryption key
23     if(IS_ATTRIBUTE(rsaKey->publicArea.objectAttributes, TPMA_OBJECT, restricted)
24        || !IS_ATTRIBUTE(rsaKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
25         return TPM_RCS_ATTRIBUTES + RC_RSA_Decrypt_keyHandle;
26
27     // NOTE: Proper operation of this command requires that the sensitive area
28     // of the key is loaded. This is assured because authorization is required
29     // to use the sensitive area of the key. In order to check the authorization,
30     // the sensitive area has to be loaded, even if authorization is with policy.
31
32     // If label is present, make sure that it is a NULL-terminated string
33     if(!IsLabelProperlyFormatted(&in->label.b))
34         return TPM_RCS_VALUE + RC_RSA_Decrypt_label;
35 // Command Output
36     // Select a scheme for decrypt.
37     scheme = CryptRsaSelectScheme(in->keyHandle, &in->inScheme);
38     if(scheme == NULL)
39         return TPM_RCS_SCHEME + RC_RSA_Decrypt_inScheme;
40
41     // Decryption.  TPM_RC_VALUE, TPM_RC_SIZE, and TPM_RC_KEY error may be
42     // returned by CryptRsaDecrypt.
```

 Copyright © TCG 2006-2020

```
43      // NOTE: CryptRsaDecrypt can also return TPM_RC_ATTRIBUTES or TPM_RC_BINDING
44      // when the key is not a decryption key but that was checked above.
45      out->message.t.size = sizeof(out->message.t.buffer);
46      result = CryptRsaDecrypt(&out->message.b, &in->cipherText.b, rsaKey,
47                               scheme, &in->label.b);
48      return result;
49  }
50  #endif // CC_RSA_Decrypt
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

14.4 TPM2_ECDH_KeyGen

14.4.1 General Description

This command uses the TPM to generate an ephemeral key pair (d_e, Q_e where $Q_e \coloneqq [d_e]G$). It uses the private ephemeral key and a loaded public key (Q_S) to compute the shared secret value ($P \coloneqq [hd_e]Q_S$).

keyHandle shall refer to a loaded, ECC key (TPM_RC_KEY). The sensitive portion of this key need not be loaded.

The curve parameters of the loaded ECC key are used to generate the ephemeral key.

NOTE This function is the equivalent of encrypting data to another object's public key. The *seed* value is used in a KDF to generate a symmetric key and that key is used to encrypt the data. Once the data is encrypted and the symmetric key discarded, only the object with the private portion of the *keyHandle* will be able to decrypt it.

The *zPoint* in the response may be encrypted using parameter encryption.

14.4.2 Command and Response

Table 49 — TPM2_ECDH_KeyGen Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ECDH_KeyGen
TPMI_DH_OBJECT	keyHandle	Handle of a loaded ECC key public area. Auth Index: None

Table 50 — TPM2_ECDH_KeyGen Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ECC_POINT	zPoint	results of $P \coloneqq h[d_e]Q_s$
TPM2B_ECC_POINT	pubPoint	generated ephemeral public point (Q_e)

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

14.4.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "ECDH_KeyGen_fp.h"
3  #if CC_ECDH_KeyGen  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_KEY	*keyHandle* does not reference an ECC key

```
4  TPM_RC
5  TPM2_ECDH_KeyGen(
6      ECDH_KeyGen_In      *in,            // IN: input parameter list
7      ECDH_KeyGen_Out     *out            // OUT: output parameter list
8      )
9  {
10     OBJECT                  *eccKey;
11     TPM2B_ECC_PARAMETER      sensitive;
12     TPM_RC                   result;
13
14 // Input Validation
15
16     eccKey = HandleToObject(in->keyHandle);
17
18     // Referenced key must be an ECC key
19     if(eccKey->publicArea.type != TPM_ALG_ECC)
20         return TPM_RCS_KEY + RC_ECDH_KeyGen_keyHandle;
21
22 // Command Output
23     do
24     {
25         TPMT_PUBLIC         *keyPublic = &eccKey->publicArea;
26         // Create ephemeral ECC key
27         result = CryptEccNewKeyPair(&out->pubPoint.point, &sensitive,
28                                     keyPublic->parameters.eccDetail.curveID);
29         if(result == TPM_RC_SUCCESS)
30         {
31             // Compute Z
32             result = CryptEccPointMultiply(&out->zPoint.point,
33                                            keyPublic->parameters.eccDetail.curveID,
34                                            &keyPublic->unique.ecc,
35                                            &sensitive,
36                                            NULL, NULL);
37                     // The point in the key is not on the curve. Indicate
38                     // that the key is bad.
39             if(result == TPM_RC_ECC_POINT)
40                 return TPM_RCS_KEY + RC_ECDH_KeyGen_keyHandle;
41              // The other possible error from CryptEccPointMultiply is
42              // TPM_RC_NO_RESULT indicating that the multiplication resulted in
43              // the point at infinity, so get a new random key and start over
44              // BTW, this never happens.
45         }
46     } while(result == TPM_RC_NO_RESULT);
47     return result;
48 }
49 #endif // CC_ECDH_KeyGen
```

14.5 TPM2_ECDH_ZGen

14.5.1 General Description

This command uses the TPM to recover the *Z* value from a public point (Q_B) and a private key (d_s). It will perform the multiplication of the provided *inPoint* (Q_B) with the private key (d_s) and return the coordinates of the resultant point ($Z = (x_Z, y_Z) \coloneqq [hd_s]Q_B$; where h is the cofactor of the curve).

keyHandle shall refer to a loaded, ECC key (TPM_RC_KEY) with the *restricted* attribute CLEAR and the *decrypt* attribute SET (TPM_RC_ATTRIBUTES).

NOTE While TPM_RC_ATTRIBUTES is preferred, TPM_RC_KEY is acceptable.

The *scheme* of the key referenced by *keyHandle* is required to be either TPM_ALG_ECDH or TPM_ALG_NULL (TPM_RC_SCHEME).

inPoint is required to be on the curve of the key referenced by *keyHandle* (TPM_RC_ECC_POINT).

The parameters of the key referenced by *keyHandle* are used to perform the point multiplication.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

14.5.2 Command and Response

Table 51 — TPM2_ECDH_ZGen Command

Type	**Name**	**Description**
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ECDH_ZGen
TPMI_DH_OBJECT	@keyHandle	handle of a loaded ECC key Auth Index: 1 Auth Role: USER
TPM2B_ECC_POINT	inPoint	a public key

Table 52 — TPM2_ECDH_ZGen Response

Type	**Name**	**Description**
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ECC_POINT	outPoint	X and Y coordinates of the product of the multiplication $Z = (x_Z, y_Z) := [hd_S]Q_B$

Copyright © TCG 2006-2020

14.5.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "ECDH_ZGen_fp.h"
3  #if CC_ECDH_ZGen  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	key referenced by *keyA* is restricted or not a decrypt key
TPM_RC_KEY	key referenced by *keyA* is not an ECC key
TPM_RC_NO_RESULT	multiplying *inPoint* resulted in a point at infinity
TPM_RC_SCHEME	the scheme of the key referenced by *keyA* is not TPM_ALG_NULL, TPM_ALG_ECDH,

```
4  TPM_RC
5  TPM2_ECDH_ZGen(
6      ECDH_ZGen_In     *in,             // IN: input parameter list
7      ECDH_ZGen_Out    *out             // OUT: output parameter list
8      )
9  {
10     TPM_RC                    result;
11     OBJECT                   *eccKey;
12
13 // Input Validation
14     eccKey = HandleToObject(in->keyHandle);
15
16     // Selected key must be a non-restricted, decrypt ECC key
17     if(eccKey->publicArea.type != TPM_ALG_ECC)
18         return TPM_RCS_KEY + RC_ECDH_ZGen_keyHandle;
19     // Selected key needs to be unrestricted with the 'decrypt' attribute
20     if(IS_ATTRIBUTE(eccKey->publicArea.objectAttributes, TPMA_OBJECT, restricted)
21        || !IS_ATTRIBUTE(eccKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
22         return TPM_RCS_ATTRIBUTES + RC_ECDH_ZGen_keyHandle;
23     // Make sure the scheme allows this use
24     if(eccKey->publicArea.parameters.eccDetail.scheme.scheme != TPM_ALG_ECDH
25        &&  eccKey->publicArea.parameters.eccDetail.scheme.scheme != TPM_ALG_NULL)
26         return TPM_RCS_SCHEME + RC_ECDH_ZGen_keyHandle;
27 // Command Output
28     // Compute Z. TPM_RC_ECC_POINT or TPM_RC_NO_RESULT may be returned here.
29     result = CryptEccPointMultiply(&out->outPoint.point,
30                                    eccKey->publicArea.parameters.eccDetail.curveID,
31                                    &in->inPoint.point,
32                                    &eccKey->sensitive.sensitive.ecc,
33                                    NULL, NULL);
34     if(result != TPM_RC_SUCCESS)
35         return RcSafeAddToResult(result, RC_ECDH_ZGen_inPoint);
36     return result;
37 }
38 #endif // CC_ECDH_ZGen
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

14.6 TPM2_ECC_Parameters

14.6.1 General Description

This command returns the parameters of an ECC curve identified by its TCG-assigned *curveID*.

The value returned is the same as that from the TCG Algorithm Registry, but may not be the same size.

EXAMPLE The value 01 may be returned as 00000001.

 Copyright © TCG 2006-2020

14.6.2 Command and Response

Table 53 — TPM2_ECC_Parameters Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ECC_Parameters
TPMI_ECC_CURVE	curveID	parameter set selector

Table 54 — TPM2_ECC_Parameters Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMS_ALGORITHM_DETAIL_ECC	parameters	ECC parameters for the selected curve

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

14.6.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "ECC_Parameters_fp.h"
3  #if CC_ECC_Parameters  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_VALUE	Unsupported ECC curve ID

```
4  TPM_RC
5  TPM2_ECC_Parameters(
6      ECC_Parameters_In    *in,            // IN: input parameter list
7      ECC_Parameters_Out   *out            // OUT: output parameter list
8      )
9  {
10 // Command Output
11
12     // Get ECC curve parameters
13     if(CryptEccGetParameters(in->curveID, &out->parameters))
14         return TPM_RC_SUCCESS;
15     else
16         return TPM_RCS_VALUE + RC_ECC_Parameters_curveID;
17 }
18 #endif // CC_ECC_Parameters
```

14.7 TPM2_ZGen_2Phase

14.7.1 General Description

This command supports two-phase key exchange protocols. The command is used in combination with TPM2_EC_Ephemeral(). TPM2_EC_Ephemeral() generates an ephemeral key and returns the public point of that ephemeral key along with a numeric value that allows the TPM to regenerate the associated private key.

The input parameters for this command are a static public key (*inQsU*), an ephemeral key (*inQeU*) from party B, and the *commitCounter* returned by TPM2_EC_Ephemeral(). The TPM uses the counter value to regenerate the ephemeral private key ($d_{e,V}$) and the associated public key ($Q_{e,V}$). *keyA* provides the static ephemeral elements $d_{s,V}$ and $Q_{s,V}$. This provides the two pairs of ephemeral and static keys that are required for the schemes supported by this command.

The TPM will compute $Z\ or\ Z_s$ and Z_e according to the selected scheme. If the scheme is not a two-phase key exchange scheme or if the scheme is not supported, the TPM will return TPM_RC_SCHEME.

It is an error if *inQsB* or *inQeB* are not on the curve of *keyA* (TPM_RC_ECC_POINT).

The two-phase key schemes that were assigned an algorithm ID as of the time of the publication of this specification are TPM_ALG_ECDH, TPM_ALG_ECMQV, and TPM_ALG_SM2.

If this command is supported, then support for TPM_ALG_ECDH is required. Support for TPM_ALG_ECMQV or TPM_ALG_SM2 is optional.

NOTE 1 If SM2 is supported and this command is supported, then the implementation is required to support the key exchange protocol of SM2, part 3.

For TPM_ALG_ECDH *outZ1* will be Zs and *outZ2* will Ze as defined in 6.1.1.2 of SP800-56A.

NOTE 2 An unrestricted decryption key using ECDH may be used in either TPM2_ECDH_ZGen() or TPM2_ZGen_2Phase as the computation done with the private part of *keyA* is the same in both cases.

For TPM_ALG_ECMQV or TPM_ALG_SM2 *outZ1* will be Z and *outZ2* will be an Empty Point.

NOTE 3 An Empty Point has two Empty Buffers as coordinates meaning the minimum *size* value for *outZ2* will be four.

If the input scheme is TPM_ALG_ECDH, then *outZ1* will be Z_s and outZ2 will be Z_e. For schemes like MQV (including SM2), *outZ1* will contain the computed value and *outZ2* will be an Empty Point.

NOTE 4 The Z values returned by the TPM are a full point and not just an x-coordinate.

If a computation of either Z produces the point at infinity, then the corresponding Z value will be an Empty Point.

 Copyright © TCG 2006-2020

14.7.2 Command and Response

Table 55 — TPM2_ZGen_2Phase Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ZGen_2Phase
TPMI_DH_OBJECT	@keyA	handle of an unrestricted decryption key ECC The private key referenced by this handle is used as $d_{S,A}$ Auth Index: 1 Auth Role: USER
TPM2B_ECC_POINT	inQsB	other party's static public key ($Q_{s,B} = (X_{s,B}, Y_{s,B})$)
TPM2B_ECC_POINT	inQeB	other party's ephemeral public key ($Q_{e,B} = (X_{e,B}, Y_{e,B})$)
TPMI_ECC_KEY_EXCHANGE	inScheme	the key exchange scheme
UINT16	counter	value returned by TPM2_EC_Ephemeral()

Table 56 — TPM2_ZGen_2Phase Response

Type	Name	Description
TPM_ST	tag	
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ECC_POINT	outZ1	X and Y coordinates of the computed value (scheme dependent)
TPM2B_ECC_POINT	outZ2	X and Y coordinates of the second computed value (scheme dependent)

14.7.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "ZGen_2Phase_fp.h"
3  #if CC_ZGen_2Phase  // Conditional expansion of this file
```

This command uses the TPM to recover one or two Z values in a two phase key exchange protocol

Error Returns	Meaning
TPM_RC_ATTRIBUTES	key referenced by *keyA* is restricted or not a decrypt key
TPM_RC_ECC_POINT	*inQsB* or *inQeB* is not on the curve of the key reference by *keyA*
TPM_RC_KEY	key referenced by *keyA* is not an ECC key
TPM_RC_SCHEME	the scheme of the key referenced by *keyA* is not TPM_ALG_NULL, TPM_ALG_ECDH, ALG_ECMQV or TPM_ALG_SM2

```
4  TPM_RC
5  TPM2_ZGen_2Phase(
6      ZGen_2Phase_In      *in,            // IN: input parameter list
7      ZGen_2Phase_Out     *out            // OUT: output parameter list
8      )
9  {
10     TPM_RC                    result;
11     OBJECT                   *eccKey;
12     TPM2B_ECC_PARAMETER       r;
13     TPM_ALG_ID                scheme;
14
15 // Input Validation
16
17     eccKey = HandleToObject(in->keyA);
18
19     // keyA must be an ECC key
20     if(eccKey->publicArea.type != TPM_ALG_ECC)
21         return TPM_RCS_KEY + RC_ZGen_2Phase_keyA;
22
23     // keyA must not be restricted and must be a decrypt key
24     if(IS_ATTRIBUTE(eccKey->publicArea.objectAttributes, TPMA_OBJECT, restricted)
25        || !IS_ATTRIBUTE(eccKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
26         return TPM_RCS_ATTRIBUTES + RC_ZGen_2Phase_keyA;
27
28     // if the scheme of keyA is TPM_ALG_NULL, then use the input scheme; otherwise
29     // the input scheme must be the same as the scheme of keyA
30     scheme = eccKey->publicArea.parameters.asymDetail.scheme.scheme;
31     if(scheme != TPM_ALG_NULL)
32     {
33         if(scheme != in->inScheme)
34             return TPM_RCS_SCHEME + RC_ZGen_2Phase_inScheme;
35     }
36     else
37         scheme = in->inScheme;
38     if(scheme == TPM_ALG_NULL)
39         return TPM_RCS_SCHEME + RC_ZGen_2Phase_inScheme;
40
41     // Input points must be on the curve of keyA
42     if(!CryptEccIsPointOnCurve(eccKey->publicArea.parameters.eccDetail.curveID,
43                                &in->inQsB.point))
44         return TPM_RCS_ECC_POINT + RC_ZGen_2Phase_inQsB;
45
46     if(!CryptEccIsPointOnCurve(eccKey->publicArea.parameters.eccDetail.curveID,
47                                &in->inQeB.point))
48         return TPM_RCS_ECC_POINT + RC_ZGen_2Phase_inQeB;
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

```
49
50      if(!CryptGenerateR(&r, &in->counter,
51                         eccKey->publicArea.parameters.eccDetail.curveID,
52                         NULL))
53          return TPM_RCS_VALUE + RC_ZGen_2Phase_counter;
54
55  // Command Output
56
57      result =
58          CryptEcc2PhaseKeyExchange(&out->outZ1.point,
59                                    &out->outZ2.point,
60                                    eccKey->publicArea.parameters.eccDetail.curveID,
61                                    scheme,
62                                    &eccKey->sensitive.sensitive.ecc,
63                                    &r,
64                                    &in->inQsB.point,
65                                    &in->inQeB.point);
66      if(result == TPM_RC_SCHEME)
67          return TPM_RCS_SCHEME + RC_ZGen_2Phase_inScheme;
68
69      if(result == TPM_RC_SUCCESS)
70          CryptEndCommit(in->counter);
71
72      return result;
73  }
74  #endif // CC_ZGen_2Phase
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

15 Symmetric Primitives

15.1 Introduction

The commands in this clause provide low-level primitives for access to the symmetric algorithms implemented in the TPM that operate on blocks of data. These include symmetric encryption and decryption as well as hash and HMAC. All of the commands in this group are stateless. That is, they have no persistent state that is retained in the TPM when the command is complete.

For hashing, HMAC, and Events that require large blocks of data with retained state, the sequence commands are provided (see clause 1).

Some of the symmetric encryption/decryption modes use an IV. When an IV is used, it may be an initiation value or a chained value from a previous stage. The chaining for each mode is:

 Copyright © TCG 2006-2020

Table 57 — Symmetric Chaining Process

Mode	Chaining process
TPM_ALG_CTR	The TPM will increment the entire IV provided by the caller. The next count value will be returned to the caller as *ivOut*. This can be the input value to the next encrypt or decrypt operation. *ivIn* is required to be the size of a block encrypted by the selected algorithm and key combination. If the size of *ivIn* is not correct, the TPM shall return TPM_RC_SIZE. EXAMPLE 1 AES requires that *ivIn* be 128 bits (16 octets). *ivOut* will be the size of a cipher block and not the size of the last encrypted block. NOTE *ivOut* will be the value of the counter after the last block is encrypted. EXAMPLE 2 If *ivIn* were 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00_{16} and four data blocks were encrypted, *ivOut* will have a value of 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 04_{16}. All the bits of the IV are incremented as if it were an unsigned integer.
TPM_ALG_OFB	In Output Feedback (OFB), the output of the pseudo-random function (the block encryption algorithm) is XORed with a plaintext block to produce a ciphertext block. *ivOut* will be the value that was XORed with the last plaintext block. That value can be used as the *ivIn* for a next buffer. *ivIn* is required to be the size of a block encrypted by the selected algorithm and key combination. If the size of *ivIn* is not correct, the TPM shall return TPM_RC_SIZE. *ivOut* will be the size of a cipher block and not the size of the last encrypted block.
TPM_ALG_CBC	For Cipher Block Chaining (CBC), a block of ciphertext is XORed with the next plaintext block and that block is encrypted. The encrypted block is then input to the encryption of the next block. The last ciphertext block then is used as an IV for the next buffer. Even though the last ciphertext block is evident in the encrypted data, it is also returned in *ivOut*. *ivIn* is required to be the size of a block encrypted by the selected algorithm and key combination. If the size of *ivIn* is not correct, the TPM shall return TPM_RC_SIZE. *inData* is required to be an even multiple of the block encrypted by the selected algorithm and key combination. If the size of *inData* is not correct, the TPM shall return TPM_RC_SIZE.
TPM_ALG_CFB	Similar to CBC in that the last ciphertext block is an input to the encryption of the next block. *ivOut* will be the value that was XORed with the last plaintext block. That value can be used as the *ivIn* for a next buffer. *ivIn* is required to be the size of a block encrypted by the selected algorithm and key combination. If the size of *ivIn* is not correct, the TPM shall return TPM_RC_SIZE. *ivOut* will be the size of a cipher block and not the size of the last encrypted block.
TPM_ALG_ECB	Electronic Codebook (ECB) has no chaining. Each block of plaintext is encrypted using the key. ECB does not support chaining and *ivIn* shall be the Empty Buffer. *ivOut* will be the Empty Buffer. *inData* is required to be an even multiple of the block encrypted by the selected algorithm and key combination. If the size of *inData* is not correct, the TPM shall return TPM_RC_SIZE.

 Copyright © TCG 2006-2020

15.2 TPM2_EncryptDecrypt

15.2.1 General Description

NOTE 1 This command is deprecated, and TPM2_EncryptDecrypt2() is preferred. This should be reflected in platform-specific specifications.

This command performs symmetric encryption or decryption using the symmetric key referenced by keyHandle and the selected mode.

keyHandle shall reference a symmetric cipher object (TPM_RC_KEY) with the *restricted* attribute CLEAR (TPM_RC_ATTRIBUTES).

If the *decrypt* parameter of the command is TRUE, then the *decrypt* attribute of the key is required to be SET (TPM_RC_ATTRIBUTES). If the *decrypt* parameter of the command is FALSE, then the *sign* attribute of the key is required to be SET (TPM_RC_ATTRIBUTES).

NOTE 2 A key may have both *decrypt* and *sign* SET.

If the mode of the key is not TPM_ALG_NULL, then that is the only mode that can be used with the key and the caller is required to set *mode* either to TPM_ALG_NULL or to the same mode as the key (TPM_RC_MODE). If the mode of the key is TPM_ALG_NULL, then the caller may set *mode* to any valid symmetric encryption/decryption mode but may not select TPM_ALG_NULL (TPM_RC_MODE).

If the TPM allows this command to be canceled before completion, then the TPM may produce incremental results and return TPM_RC_SUCCESS rather than TPM_RC_CANCELED. In such case, *outData* may be less than *inData*.

NOTE 3 If all the data is encrypted/decrypted, the size of outData will be the same as inData.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

15.2.2 Command and Response

Table 58 — TPM2_EncryptDecrypt Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_EncryptDecrypt
TPMI_DH_OBJECT	@keyHandle	the symmetric key used for the operation Auth Index: 1 Auth Role: USER
TPMI_YES_NO	decrypt	if YES, then the operation is decryption; if NO, the operation is encryption
TPMI_ALG_CIPHER_MODE+	mode	symmetric encryption/decryption mode this field shall match the default mode of the key or be TPM_ALG_NULL.
TPM2B_IV	ivIn	an initial value as required by the algorithm
TPM2B_MAX_BUFFER	inData	the data to be encrypted/decrypted

Table 59 — TPM2_EncryptDecrypt Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_MAX_BUFFER	outData	encrypted or decrypted output
TPM2B_IV	ivOut	chaining value to use for IV in next round

 Copyright © TCG 2006-2020

15.2.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "EncryptDecrypt_fp.h"
3   #if CC_EncryptDecrypt2
4   #include  "EncryptDecrypt_spt_fp.h"
5   #endif
6   #if CC_EncryptDecrypt  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_KEY	is not a symmetric decryption key with both public and private portions loaded
TPM_RC_SIZE	*IvIn* size is incompatible with the block cipher mode; or *inData* size is not an even multiple of the block size for CBC or ECB mode
TPM_RC_VALUE	*keyHandle* is restricted and the argument *mode* does not match the key's mode

```
7   TPM_RC
8   TPM2_EncryptDecrypt(
9       EncryptDecrypt_In   *in,            // IN: input parameter list
10      EncryptDecrypt_Out  *out            // OUT: output parameter list
11      )
12  {
13  #if CC_EncryptDecrypt2
14      return EncryptDecryptShared(in->keyHandle, in->decrypt, in->mode,
15                                  &in->ivIn, &in->inData, out);
16  #else
17      OBJECT              *symKey;
18      UINT16               keySize;
19      UINT16               blockSize;
20      BYTE                *key;
21      TPM_ALG_ID           alg;
22      TPM_ALG_ID           mode;
23      TPM_RC               result;
24      BOOL                 OK;
25      TPMA_OBJECT          attributes;
26
27  // Input Validation
28      symKey = HandleToObject(in->keyHandle);
29      mode = symKey->publicArea.parameters.symDetail.sym.mode.sym;
30      attributes = symKey->publicArea.objectAttributes;
31
32      // The input key should be a symmetric key
33      if(symKey->publicArea.type != TPM_ALG_SYMCIPHER)
34          return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;
35      // The key must be unrestricted and allow the selected operation
36      OK = IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
37      if(YES == in->decrypt)
38          OK = OK && IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt);
39      else
40          OK = OK && IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign);
41      if(!OK)
42          return TPM_RCS_ATTRIBUTES + RC_EncryptDecrypt_keyHandle;
43
44      // If the key mode is not TPM_ALG_NULL...
45      // or TPM_ALG_NULL
46      if(mode != TPM_ALG_NULL)
47      {
48          // then the input mode has to be TPM_ALG_NULL or the same as the key
49          if((in->mode != TPM_ALG_NULL) && (in->mode != mode))
```

 Copyright © TCG 2006-2020

```
50              return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
51      }
52      else
53      {
54          // if the key mode is null, then the input can't be null
55          if(in->mode == TPM_ALG_NULL)
56              return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
57          mode = in->mode;
58      }
59      // The input iv for ECB mode should be an Empty Buffer.  All the other modes
60      // should have an iv size same as encryption block size
61      keySize = symKey->publicArea.parameters.symDetail.sym.keyBits.sym;
62      alg = symKey->publicArea.parameters.symDetail.sym.algorithm;
63      blockSize = CryptGetSymmetricBlockSize(alg, keySize);
64
65      // reverify the algorithm. This is mainly to keep static analysis tools happy
66      if(blockSize == 0)
67          return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;
68
69      // Note: When an algorithm is not supported by a TPM, the TPM_ALG_xxx for that
70      // algorithm is not defined. However, it is assumed that the ALG_xxx_VALUE for
71      // the algorithm is always defined. Both have the same numeric value.
72      // ALG_xxx_VALUE is used here so that the code does not get cluttered with
73      // #ifdef's. Having this check does not mean that the algorithm is supported.
74      // If it was not supported the unmarshaling code would have rejected it before
75      // this function were called. This means that, depending on the implementation,
76      // the check could be redundant but it doesn't hurt.
77      if(((mode == ALG_ECB_VALUE) && (in->ivIn.t.size != 0))
78         || ((mode != ALG_ECB_VALUE) && (in->ivIn.t.size != blockSize)))
79          return TPM_RCS_SIZE + RC_EncryptDecrypt_ivIn;
80
81      // The input data size of CBC mode or ECB mode must be an even multiple of
82      // the symmetric algorithm's block size
83      if(((mode == ALG_CBC_VALUE) || (mode == ALG_ECB_VALUE))
84         && ((in->inData.t.size % blockSize) != 0))
85          return TPM_RCS_SIZE + RC_EncryptDecrypt_inData;
86
87      // Copy IV
88      // Note: This is copied here so that the calls to the encrypt/decrypt functions
89      // will modify the output buffer, not the input buffer
90      out->ivOut = in->ivIn;
91
92  // Command Output
93      key = symKey->sensitive.sensitive.sym.t.buffer;
94      // For symmetric encryption, the cipher data size is the same as plain data
95      // size.
96      out->outData.t.size = in->inData.t.size;
97      if(in->decrypt == YES)
98      {
99          // Decrypt data to output
100         result = CryptSymmetricDecrypt(out->outData.t.buffer, alg, keySize, key,
101                                        &(out->ivOut), mode, in->inData.t.size,
102                                        in->inData.t.buffer);
103     }
104     else
105     {
106         // Encrypt data to output
107         result = CryptSymmetricEncrypt(out->outData.t.buffer, alg, keySize, key,
108                                        &(out->ivOut), mode, in->inData.t.size,
109                                        in->inData.t.buffer);
110     }
111     return result;
112 #endif // CC_EncryptDecrypt2
113
114 }
115 #endif // CC_EncryptDecrypt
```

15.3 TPM2_EncryptDecrypt2

15.3.1 General Description

This command is identical to TPM2_EncryptDecrypt(), except that the *inData* parameter is the first parameter. This permits *inData* to be parameter encrypted.

NOTE In platform specification updates, this command is preferred and TPM2_EncryptDecrypt() should be deprecated.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

15.3.2 Comand and Response

Table 60 — TPM2_EncryptDecrypt2 Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_EncryptDecrypt2
TPMI_DH_OBJECT	@keyHandle	the symmetric key used for the operation Auth Index: 1 Auth Role: USER
TPM2B_MAX_BUFFER	inData	the data to be encrypted/decrypted
TPMI_YES_NO	decrypt	if YES, then the operation is decryption; if NO, the operation is encryption
TPMI_ALG_CIPHER_MODE+	mode	symmetric mode this field shall match the default mode of the key or be TPM_ALG_NULL.
TPM2B_IV	ivIn	an initial value as required by the algorithm

Table 61 — TPM2_EncryptDecrypt2 Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_MAX_BUFFER	outData	encrypted or decrypted output
TPM2B_IV	ivOut	chaining value to use for IV in next round

 Copyright © TCG 2006-2020

15.3.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "EncryptDecrypt2_fp.h"
3  #include "EncryptDecrypt_fp.h"
4  #include "EncryptDecrypt_spt_fp.h"
5  #if CC_EncryptDecrypt2  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_KEY	is not a symmetric decryption key with both public and private portions loaded
TPM_RC_SIZE	*IvIn* size is incompatible with the block cipher mode; or *inData* size is not an even multiple of the block size for CBC or ECB mode
TPM_RC_VALUE	*keyHandle* is restricted and the argument *mode* does not match the key's mode

```
6  TPM_RC
7  TPM2_EncryptDecrypt2(
8      EncryptDecrypt2_In   *in,            // IN: input parameter list
9      EncryptDecrypt2_Out  *out            // OUT: output parameter list
10     )
11 {
12     TPM_RC                result;
13     // EncryptDecyrptShared() performs the operations as shown in
14     // TPM2_EncrypDecrypt
15     result = EncryptDecryptShared(in->keyHandle, in->decrypt, in->mode,
16                                   &in->ivIn, &in->inData,
17                                   (EncryptDecrypt_Out *)out);
18     // Handle response code swizzle.
19     switch(result)
20     {
21         case TPM_RCS_MODE + RC_EncryptDecrypt_mode:
22             result = TPM_RCS_MODE + RC_EncryptDecrypt2_mode;
23             break;
24         case TPM_RCS_SIZE + RC_EncryptDecrypt_ivIn:
25             result = TPM_RCS_SIZE + RC_EncryptDecrypt2_ivIn;
26             break;
27         case TPM_RCS_SIZE + RC_EncryptDecrypt_inData:
28             result = TPM_RCS_SIZE + RC_EncryptDecrypt2_inData;
29             break;
30         default:
31             break;
32     }
33     return result;
34 }
35 #endif // CC_EncryptDecrypt2
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

15.4 TPM2_Hash

15.4.1 General Description

This command performs a hash operation on a data buffer and returns the results.

NOTE If the data buffer to be hashed is larger than will fit into the TPM's input buffer, then the sequence hash commands will need to be used.

If the results of the hash will be used in a signing operation that uses a restricted signing key, then the ticket returned by this command can indicate that the hash is safe to sign.

If the digest is not safe to sign, then the TPM will return a TPMT_TK_HASHCHECK with the hierarchy set to TPM_RH_NULL and *digest* set to the Empty Buffer.

If *hierarchy* is TPM_RH_NULL, then *digest* in the ticket will be the Empty Buffer.

 Copyright © TCG 2006-2020

15.4.2 Command and Response

Table 62 — TPM2_Hash Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit, decrypt, or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Hash
TPM2B_MAX_BUFFER	data	data to be hashed
TPMI_ALG_HASH	hashAlg	algorithm for the hash being computed – shall not be TPM_ALG_NULL
TPMI_RH_HIERARCHY+	hierarchy	hierarchy to use for the ticket (TPM_RH_NULL allowed)

Table 63 — TPM2_Hash Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	outHash	results
TPMT_TK_HASHCHECK	validation	ticket indicating that the sequence of octets used to compute *outDigest* did not start with TPM_GENERATED_VALUE will be a NULL ticket if the digest may not be signed with a restricted key

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

15.4.3 Detailed Actions

```
1 #include "Tpm.h"
2 #include "Hash_fp.h"
3 #if CC_Hash  // Conditional expansion of this file
4 TPM_RC
5 TPM2_Hash(
6     Hash_In         *in,            // IN: input parameter list
7     Hash_Out        *out            // OUT: output parameter list
8     )
9 {
10     HASH_STATE          hashState;
11
12 // Command Output
13
14     // Output hash
15         // Start hash stack
16     out->outHash.t.size = CryptHashStart(&hashState, in->hashAlg);
17         // Adding hash data
18     CryptDigestUpdate2B(&hashState, &in->data.b);
19         // Complete hash
20     CryptHashEnd2B(&hashState, &out->outHash.b);
21
22     // Output ticket
23     out->validation.tag = TPM_ST_HASHCHECK;
24     out->validation.hierarchy = in->hierarchy;
25
26     if(in->hierarchy == TPM_RH_NULL)
27     {
28         // Ticket is not required
29         out->validation.hierarchy = TPM_RH_NULL;
30         out->validation.digest.t.size = 0;
31     }
32     else if(in->data.t.size >= sizeof(TPM_GENERATED)
33             && !TicketIsSafe(&in->data.b))
34     {
35         // Ticket is not safe
36         out->validation.hierarchy = TPM_RH_NULL;
37         out->validation.digest.t.size = 0;
38     }
39     else
40     {
41         // Compute ticket
42         TicketComputeHashCheck(in->hierarchy, in->hashAlg,
43                                &out->outHash, &out->validation);
44     }
45
46     return TPM_RC_SUCCESS;
47 }
48 #endif // CC_Hash
```

15.5 TPM2_HMAC

15.5.1 General Description

This command performs an HMAC on the supplied data using the indicated hash algorithm.

NOTE 1 A TPM may implement either TPM2_HMAC() or TPM2_MAC() but not both, as they have the same command code and there is no way to distinguish them. A TPM that supports TPM2_MAC() will support any code that was written to use TPM2_HMAC(), but a TPM that supports TPM2_HMAC() will not support a MAC based on symmetric block ciphers.

The caller shall provide proper authorization for use of *handle*.

If the *sign* attribute is not SET in the key referenced by *handle* then the TPM shall return TPM_RC_KEY. If the key type is not TPM_ALG_KEYEDHASH then the TPM shall return TPM_RC_TYPE. If the key referenced by *handle* has the *restricted* attribute SET, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE 2 For symmetric signing with a restricted key, see TPM2_Sign.

If the default scheme of the key referenced by *handle* is not TPM_ALG_NULL, then the *hashAlg* parameter is required to be either the same as the key's default or TPM_ALG_NULL (TPM_RC_VALUE). If the default scheme of the key is TPM_ALG_NULL, then hashAlg is required to be a valid hash and not TPM_ALG_NULL (TPM_RC_VALUE) (see hash selection matrix in

Table 72).

NOTE 3 A key may only have both sign and decrypt SET if the key is unrestricted. When both sign and decrypt are set, there is no default scheme for the key and the hash algorithm must be specified.

 Copyright © TCG 2006-2020

15.5.2 Command and Response

Table 64 — TPM2_HMAC Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_HMAC
TPMI_DH_OBJECT	@handle	handle for the symmetric signing key providing the HMAC key Auth Index: 1 Auth Role: USER
TPM2B_MAX_BUFFER	buffer	HMAC data
TPMI_ALG_HASH+	hashAlg	algorithm to use for HMAC

Table 65 — TPM2_HMAC Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	outHMAC	the returned HMAC in a sized buffer

 Copyright © TCG 2006-2020

15.5.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "HMAC_fp.h"
3   #if CC_HMAC  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	key referenced by *handle* is a restricted key
TPM_RC_KEY	*handle* does not reference a signing key
TPM_RC_TYPE	key referenced by *handle* is not an HMAC key
TPM_RC_VALUE	*hashAlg* is not compatible with the hash algorithm of the scheme of the object referenced by *handle*

```
4   TPM_RC
5   TPM2_HMAC(
6       HMAC_In         *in,            // IN: input parameter list
7       HMAC_Out        *out            // OUT: output parameter list
8       )
9   {
10      HMAC_STATE                 hmacState;
11      OBJECT                    *hmacObject;
12      TPMI_ALG_HASH              hashAlg;
13      TPMT_PUBLIC               *publicArea;
14
15  // Input Validation
16
17      // Get HMAC key object and public area pointers
18      hmacObject = HandleToObject(in->handle);
19      publicArea = &hmacObject->publicArea;
20      // Make sure that the key is an HMAC key
21      if(publicArea->type != TPM_ALG_KEYEDHASH)
22          return TPM_RCS_TYPE + RC_HMAC_handle;
23
24      // and that it is unrestricted
25      if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
26          return TPM_RCS_ATTRIBUTES + RC_HMAC_handle;
27
28      // and that it is a signing key
29      if(!IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
30          return TPM_RCS_KEY + RC_HMAC_handle;
31
32      // See if the key has a default
33      if(publicArea->parameters.keyedHashDetail.scheme.scheme == TPM_ALG_NULL)
34          // it doesn't so use the input value
35          hashAlg = in->hashAlg;
36      else
37      {
38          // key has a default so use it
39          hashAlg
40              = publicArea->parameters.keyedHashDetail.scheme.details.hmac.hashAlg;
41          // and verify that the input was either the  TPM_ALG_NULL or the default
42          if(in->hashAlg != TPM_ALG_NULL && in->hashAlg != hashAlg)
43              hashAlg = TPM_ALG_NULL;
44      }
45      // if we ended up without a hash algorithm then return an error
46      if(hashAlg == TPM_ALG_NULL)
47          return TPM_RCS_VALUE + RC_HMAC_hashAlg;
48
49  // Command Output
50
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

```
51      // Start HMAC stack
52      out->outHMAC.t.size = CryptHmacStart2B(&hmacState, hashAlg,
53                                             &hmacObject->sensitive.sensitive.bits.b);
54      // Adding HMAC data
55      CryptDigestUpdate2B(&hmacState.hashState, &in->buffer.b);
56
57      // Complete HMAC
58      CryptHmacEnd2B(&hmacState, &out->outHMAC.b);
59
60      return TPM_RC_SUCCESS;
61  }
62  #endif // CC_HMAC
```

15.6 TPM2_MAC

15.6.1 General Description

This command performs an HMAC or a block cipher MAC on the supplied data using the indicated algorithm.

NOTE 1 A TPM may implement either TPM2_HMAC() or TPM2_MAC() but not both as they have the same command code and there is no way to distinguish them. A TPM that supports TPM2_MAC() will support any code that was written to use TPM2_HMAC() but a TPM that supports TPM2_HMAC () will not support a MAC based on symmetric block ciphers.

The caller shall provide proper authorization for use of *handle*.

If the *sign* attribute is not SET in the key referenced by *handle* then the TPM shall return TPM_RC_KEY. If the key type is neither TPM_ALG_KEYEDHASH nor TPM_ALG_SYMCIPHER then the TPM shall return TPM_RC_TYPE. If the key referenced by *handle* has the *restricted* attribute SET, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE 2 For symmetric signing with a restricted key, see TPM2_Sign.

If the default scheme or mode of the key referenced by *handle* is not TPM_ALG_NULL, then the *inScheme* parameter is required to be either the same as the key's default or TPM_ALG_NULL (TPM_RC_VALUE).

If the default scheme of an HMAC key is TPM_ALG_NULL, then *inScheme* is required to be a valid hash and not TPM_ALG_NULL (TPM_RC_VALUE) (see algorithm selection matrix in

Table 75).

If the default mode of a symmetric cipher key is TPM_ALG_NULL, then *inScheme* is required to be a valid block cipher mode for authentication and not TPM_ALG_NULL (TPM_RC_VALUE)

NOTE 3 A key may only have both sign and decrypt SET if the key is unrestricted. When both sign and decrypt are set, there is no default scheme for the key and *inScheme* may not be TPM_ALG_NULL.

NOTE 4 TPM2_MAC() was added in revision 01.43.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

15.6.2 Command and Response

Table 66 — TPM2_MAC Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_MAC
TPMI_DH_OBJECT	@handle	handle for the symmetric signing key providing the MAC key Auth Index: 1 Auth Role: USER
TPM2B_MAX_BUFFER	buffer	MAC data
TPMI_ALG_MAC_SCHEME+	inScheme	algorithm to use for MAC

Table 67 — TPM2_MAC Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	outMAC	the returned MAC in a sized buffer

 Copyright © TCG 2006-2020

15.6.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "MAC_fp.h"
3  #if CC_MAC  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	key referenced by *handle* is a restricted key
TPM_RC_KEY	*handle* does not reference a signing key
TPM_RC_TYPE	key referenced by *handle* is not an HMAC key
TPM_RC_VALUE	*hashAlg* is not compatible with the hash algorithm of the scheme of the object referenced by *handle*

```
4  TPM_RC
5  TPM2_MAC(
6      MAC_In          *in,            // IN: input parameter list
7      MAC_Out         *out            // OUT: output parameter list
8      )
9  {
10     OBJECT                  *keyObject;
11     HMAC_STATE               state;
12     TPMT_PUBLIC             *publicArea;
13     TPM_RC                   result;
14
15 // Input Validation
16     // Get MAC key object and public area pointers
17     keyObject = HandleToObject(in->handle);
18     publicArea = &keyObject->publicArea;
19
20     // If the key is not able to do a MAC, indicate that the handle selects an
21     // object that can't do a MAC
22     result = CryptSelectMac(publicArea, &in->inScheme);
23     if(result == TPM_RCS_TYPE)
24         return TPM_RCS_TYPE + RC_MAC_handle;
25     // If there is another error type, indicate that the scheme and key are not
26     // compatible
27     if(result != TPM_RC_SUCCESS)
28         return RcSafeAddToResult(result, RC_MAC_inScheme);
29     // Make sure that the key is not restricted
30     if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
31         return TPM_RCS_ATTRIBUTES + RC_MAC_handle;
32     // and that it is a signing key
33     if(!IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
34         return TPM_RCS_KEY + RC_MAC_handle;
35 // Command Output
36     out->outMAC.t.size = CryptMacStart(&state, &publicArea->parameters,
37                                        in->inScheme,
38                                        &keyObject->sensitive.sensitive.any.b);
39     // If the mac can't start, treat it as a fatal error
40     if(out->outMAC.t.size == 0)
41         return TPM_RC_FAILURE;
42     CryptDigestUpdate2B(&state.hashState, &in->buffer.b);
43     // If the MAC result is not what was expected, it is a fatal error
44     if(CryptHmacEnd2B(&state, &out->outMAC.b) != out->outMAC.t.size)
45         return TPM_RC_FAILURE;
46     return TPM_RC_SUCCESS;
47 }
48 #endif // CC_MAC
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

16 Random Number Generator

16.1 TPM2_GetRandom

16.1.1 General Description

This command returns the next *bytesRequested* octets from the random number generator (RNG).

NOTE 1 It is recommended that a TPM implement the RNG in a manner that would allow it to return RNG octets such that, as long as the value of *bytesRequested* is not greater than the maximum digest size, the frequency of *bytesRequested* being more than the number of octets available is an infrequent occurrence.

If *bytesRequested* is more than will fit into a TPM2B_DIGEST on the TPM, no error is returned but the TPM will only return as much data as will fit into a TPM2B_DIGEST buffer for the TPM.

NOTE 2 TPM2B_DIGEST is large enough to hold the largest digest that may be produced by the TPM. Because that digest size changes according to the implemented hashes, the maximum amount of data returned by this command is TPM implementation-dependent.

 Copyright © TCG 2006-2020

16.1.2 Command and Response

Table 68 — TPM2_GetRandom Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_GetRandom
UINT16	bytesRequested	number of octets to return

Table 69 — TPM2_GetRandom Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	randomBytes	the random octets

 Copyright © TCG 2006-2020

16.1.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "GetRandom_fp.h"
3  #if CC_GetRandom  // Conditional expansion of this file
4  TPM_RC
5  TPM2_GetRandom(
6      GetRandom_In    *in,             // IN: input parameter list
7      GetRandom_Out   *out             // OUT: output parameter list
8      )
9  {
10 // Command Output
11
12     // if the requested bytes exceed the output buffer size, generates the
13     // maximum bytes that the output buffer allows
14     if(in->bytesRequested > sizeof(TPMU_HA))
15         out->randomBytes.t.size = sizeof(TPMU_HA);
16     else
17         out->randomBytes.t.size = in->bytesRequested;
18
19     CryptRandomGenerate(out->randomBytes.t.size, out->randomBytes.t.buffer);
20
21     return TPM_RC_SUCCESS;
22 }
23 #endif // CC_GetRandom
```

16.2 TPM2_StirRandom

16.2.1 General Description

This command is used to add "additional information" to the RNG state.

NOTE The "additional information" is as defined in SP800-90A.

The *inData* parameter may not be larger than 128 octets.

 Copyright © TCG 2006-2020

16.2.2 Command and Response

Table 70 — TPM2_StirRandom Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_StirRandom {NV}
TPM2B_SENSITIVE_DATA	inData	additional information

Table 71 — TPM2_StirRandom Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

16.2.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "StirRandom_fp.h"
3  #if CC_StirRandom  // Conditional expansion of this file
4  TPM_RC
5  TPM2_StirRandom(
6      StirRandom_In   *in              // IN: input parameter list
7      )
8  {
9  // Internal Data Update
10     CryptRandomStir(in->inData.t.size, in->inData.t.buffer);
11
12     return TPM_RC_SUCCESS;
13 }
14 #endif // CC_StirRandom
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

17 Hash/HMAC/Event Sequences

17.1 Introduction

All of the commands in this group are to support sequences for which an intermediate state must be maintained. For a description of sequences, see "Hash, MAC, and Event Sequences" in TPM 2.0 Part 1.

A TPM may implement either TPM2_HMAC_Start() or TPM2_MAC_Start() but not both as they have the same command code and there is no way to distinguish them. A TPM that supports TPM2_MAC_Start() will support any code that was written to use TPM2_HMAC_Start() but a TPM that supports TPM2_HMAC_Start() will not support a MAC based on symmetric block ciphers.

17.2 TPM2_HMAC_Start

17.2.1 General Description

This command starts an HMAC sequence. The TPM will create and initialize an HMAC sequence structure, assign a handle to the sequence, and set the *authValue* of the sequence object to the value in *auth*.

NOTE 1 The structure of a sequence object is vendor-dependent.

The caller shall provide proper authorization for use of *handle*.

If the *sign* attribute is not SET in the key referenced by *handle* then the TPM shall return TPM_RC_KEY. If the key type is not TPM_ALG_KEYEDHASH then the TPM shall return TPM_RC_TYPE. If the key referenced by *handle* has the *restricted* attribute SET, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE 2 For symmetric signing with a restricted key, see TPM2_Sign.

If the default scheme of the key referenced by *handle* is not TPM_ALG_NULL, then the *hashAlg* parameter is required to be either the same as the key's default or TPM_ALG_NULL (TPM_RC_VALUE). If the default scheme of the key is TPM_ALG_NULL, then hashAlg is required to be a valid hash and not TPM_ALG_NULL (TPM_RC_VALUE).

Table 72 — Hash Selection Matrix

handle→restricted **(key's restricted attribute)**	***handle→scheme*** **(hash algorithm from key's *scheme*)**	***hashAlg***	**hash used**
CLEAR (unrestricted)	TPM_ALG_NULL[1]	TPM_ALG_NULL	error[1] (TPM_RC_VALUE)
CLEAR	TPM_ALG_NULL	valid hash	*hashAlg*
CLEAR	valid hash	TPM_ALG_NULL or same as *handle→scheme*	*handle→scheme*
CLEAR	valid hash	valid hash	error (TPM_RC_VALUE) if *hashAlg* != *handle->scheme*
SET (restricted)	don't care	don't care	TPM_RC_ATTRIBUTES

NOTES:
1) A hash algorithm is required for the HMAC.

NOTE 1 A TPM may implement either TPM2_HMAC_Start() or TPM2_MAC_Start() but not both, as they have the same command code and there is no way to distinguish them. A TPM that supports TPM2_MAC_Start() will support any code that was written to use TPM2_HMAC_Start(), but a TPM that supports TPM2_HMAC_Start() will not support a MAC based on symmetric block ciphers.

17.2.2 Command and Response

Table 73 — TPM2_HMAC_Start Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_HMAC_Start
TPMI_DH_OBJECT	@handle	handle of an HMAC key Auth Index: 1 Auth Role: USER
TPM2B_AUTH	auth	authorization value for subsequent use of the sequence
TPMI_ALG_HASH+	hashAlg	the hash algorithm to use for the HMAC

Table 74 — TPM2_HMAC_Start Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_DH_OBJECT	sequenceHandle	a handle to reference the sequence

 Copyright © TCG 2006-2020

17.2.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "HMAC_Start_fp.h"
3   #if CC_HMAC_Start  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	key referenced by *handle* is not a signing key or is restricted
TPM_RC_OBJECT_MEMORY	no space to create an internal object
TPM_RC_KEY	key referenced by *handle* is not an HMAC key
TPM_RC_VALUE	*hashAlg* is not compatible with the hash algorithm of the scheme of the object referenced by *handle*

```
4   TPM_RC
5   TPM2_HMAC_Start(
6       HMAC_Start_In    *in,            // IN: input parameter list
7       HMAC_Start_Out   *out            // OUT: output parameter list
8       )
9   {
10      OBJECT                    *keyObject;
11      TPMT_PUBLIC               *publicArea;
12      TPM_ALG_ID                 hashAlg;
13
14  // Input Validation
15
16      // Get HMAC key object and public area pointers
17      keyObject = HandleToObject(in->handle);
18      publicArea = &keyObject->publicArea;
19
20      // Make sure that the key is an HMAC key
21      if(publicArea->type != TPM_ALG_KEYEDHASH)
22          return TPM_RCS_TYPE + RC_HMAC_Start_handle;
23
24      // and that it is unrestricted
25      if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
26          return TPM_RCS_ATTRIBUTES + RC_HMAC_Start_handle;
27
28      // and that it is a signing key
29      if(!IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
30          return TPM_RCS_KEY + RC_HMAC_Start_handle;
31
32      // See if the key has a default
33      if(publicArea->parameters.keyedHashDetail.scheme.scheme == TPM_ALG_NULL)
34          // it doesn't so use the input value
35          hashAlg = in->hashAlg;
36      else
37      {
38          // key has a default so use it
39          hashAlg
40              = publicArea->parameters.keyedHashDetail.scheme.details.hmac.hashAlg;
41          // and verify that the input was either the  TPM_ALG_NULL or the default
42          if(in->hashAlg != TPM_ALG_NULL && in->hashAlg != hashAlg)
43              hashAlg = TPM_ALG_NULL;
44      }
45      // if we ended up without a hash algorithm then return an error
46      if(hashAlg == TPM_ALG_NULL)
47          return TPM_RCS_VALUE + RC_HMAC_Start_hashAlg;
48
49  // Internal Data Update
50
```

 Copyright © TCG 2006-2020

```
51      // Create a HMAC sequence object. A TPM_RC_OBJECT_MEMORY error may be
52      // returned at this point
53      return ObjectCreateHMACSequence(hashAlg,
54                                      keyObject,
55                                      &in->auth,
56                                      &out->sequenceHandle);
57  }
58  #endif // CC_HMAC_Start
```

 Copyright © TCG 2006-2020

17.3 TPM2_MAC_Start

17.3.1 General Description

This command starts a MAC sequence. The TPM will create and initialize a MAC sequence structure, assign a handle to the sequence, and set the *authValue* of the sequence object to the value in *auth*.

NOTE 1 The structure of a sequence object is vendor-dependent.

The caller shall provide proper authorization for use of *handle*.

If the *sign* attribute is not SET in the key referenced by *handle* then the TPM shall return TPM_RC_KEY. If the key type is not TPM_ALG_KEYEDHASH or TPM_ALG_SYMCIPHER then the TPM shall return TPM_RC_TYPE. If the key referenced by *handle* has the *restricted* attribute SET, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE 2 For symmetric signing with a restricted key, see TPM2_Sign.

If the default scheme of the key referenced by *handle* is not TPM_ALG_NULL, then the *inScheme* parameter is required to be either the same as the key's default or TPM_ALG_NULL (TPM_RC_VALUE). If the default scheme of the key is TPM_ALG_NULL, then *inScheme*is required to be a valid hash or symmetric MAC scheme and not TPM_ALG_NULL (TPM_RC_VALUE).

Table 75 — Algorithm Selection Matrix

handle→restricted **(key's restricted attribute)**	***handle→scheme*** **(algorithm from key's *scheme*)**	***inScheme***	**algorithm used**
CLEAR (unrestricted)	TPM_ALG_NULL(1)	TPM_ALG_NULL	error(1) (TPM_RC_VALUE)
CLEAR	TPM_ALG_NULL	valid hash or symmetric MAC	*inScheme*
CLEAR	not TPM_ALG_NULL	TPM_ALG_NULL or same as *handle→scheme*	*handle→scheme*
CLEAR	not TPM_ALG_NULL	not TPM_AGL_NULL	error (TPM_RC_VALUE) if*inScheme*!= *handle->scheme*
SET (restricted)	don't care	don't care	TPM_RC_ATTRIBUTES

NOTES:
1) A hash algorithm is required for the HMAC.
2) hashAlg shall be TPM_ALG_NULL for handle referencing a CMAC key.

NOTE 3 For a TPM_ALG_SYMCIPHER key, the symmetric block cipher algorithm is part of the key definition.

NOTE 4 TPM2_MAC_Start() was added in revision 01.43.

 Copyright © TCG 2006-2020

17.3.2 Command and Response

Table 76 — TPM2_MAC_Start Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_MAC_Start
TPMI_DH_OBJECT	@handle	handle of a MAC key Auth Index: 1 Auth Role: USER
TPM2B_AUTH	auth	authorization value for subsequent use of the sequence
TPMI_ALG_MAC_SCHEME+	inScheme	the algorithm to use for the MAC

Table 77 — TPM2_MAC_Start Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_DH_OBJECT	sequenceHandle	a handle to reference the sequence

 Copyright © TCG 2006-2020

17.3.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "MAC_Start_fp.h"
3   #if CC_MAC_Start  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	key referenced by *handle* is not a signing key or is restricted
TPM_RC_OBJECT_MEMORY	no space to create an internal object
TPM_RC_KEY	key referenced by *handle* is not an HMAC key
TPM_RC_VALUE	*hashAlg* is not compatible with the hash algorithm of the scheme of the object referenced by *handle*

```
4   TPM_RC
5   TPM2_MAC_Start(
6       MAC_Start_In    *in,            // IN: input parameter list
7       MAC_Start_Out   *out            // OUT: output parameter list
8       )
9   {
10      OBJECT                  *keyObject;
11      TPMT_PUBLIC             *publicArea;
12      TPM_RC                   result;
13
14  // Input Validation
15
16      // Get HMAC key object and public area pointers
17      keyObject = HandleToObject(in->handle);
18      publicArea = &keyObject->publicArea;
19
20      // Make sure that the key can do what is required
21      result = CryptSelectMac(publicArea, &in->inScheme);
22      // If the key is not able to do a MAC, indicate that the handle selects an
23      // object that can't do a MAC
24      if(result == TPM_RCS_TYPE)
25          return TPM_RCS_TYPE + RC_MAC_Start_handle;
26      // If there is another error type, indicate that the scheme and key are not
27      // compatible
28      if(result != TPM_RC_SUCCESS)
29          return RcSafeAddToResult(result, RC_MAC_Start_inScheme);
30      // Make sure that the key is not restricted
31      if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
32          return TPM_RCS_ATTRIBUTES + RC_MAC_Start_handle;
33      // and that it is a signing key
34      if(!IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
35          return TPM_RCS_KEY + RC_MAC_Start_handle;
36
37  // Internal Data Update
38      // Create a HMAC sequence object. A TPM_RC_OBJECT_MEMORY error may be
39      // returned at this point
40      return ObjectCreateHMACSequence(in->inScheme,
41                                      keyObject,
42                                      &in->auth,
43                                      &out->sequenceHandle);
44  }
45  #endif // CC_MAC_Start
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

17.4 TPM2_HashSequenceStart

17.4.1 General Description

This command starts a hash or an Event Sequence. If *hashAlg* is an implemented hash, then a hash sequence is started. If *hashAlg* is TPM_ALG_NULL, then an Event Sequence is started. If *hashAlg* is neither an implemented algorithm nor TPM_ALG_NULL, then the TPM shall return TPM_RC_HASH.

Depending on *hashAlg*, the TPM will create and initialize a Hash Sequence context or an Event Sequence context. Additionally, it will assign a handle to the context and set the *authValue* of the context to the value in *auth*. A sequence context for an Event (*hashAlg* = TPM_ALG_NULL) contains a hash context for each of the PCR banks implemented on the TPM.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

17.4.2 Command and Response

Table 78 — TPM2_HashSequenceStart Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_HashSequenceStart
TPM2B_AUTH	auth	authorization value for subsequent use of the sequence
TPMI_ALG_HASH+	hashAlg	the hash algorithm to use for the hash sequence An Event Sequence starts if this is TPM_ALG_NULL.

Table 79 — TPM2_HashSequenceStart Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_DH_OBJECT	sequenceHandle	a handle to reference the sequence

 Copyright © TCG 2006-2020

17.4.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "HashSequenceStart_fp.h"
3  #if CC_HashSequenceStart  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_OBJECT_MEMORY	no space to create an internal object

```
4  TPM_RC
5  TPM2_HashSequenceStart(
6      HashSequenceStart_In     *in,            // IN: input parameter list
7      HashSequenceStart_Out    *out            // OUT: output parameter list
8      )
9  {
10 // Internal Data Update
11
12     if(in->hashAlg == TPM_ALG_NULL)
13         // Start a event sequence.  A TPM_RC_OBJECT_MEMORY error may be
14         // returned at this point
15         return ObjectCreateEventSequence(&in->auth, &out->sequenceHandle);
16
17     // Start a hash sequence.  A TPM_RC_OBJECT_MEMORY error may be
18     // returned at this point
19     return ObjectCreateHashSequence(in->hashAlg, &in->auth, &out->sequenceHandle);
20 }
21 #endif // CC_HashSequenceStart
```


 Copyright © TCG 2006-2020

17.5 TPM2_SequenceUpdate

17.5.1 General Description

This command is used to add data to a hash or HMAC sequence. The amount of data in buffer may be any size up to the limits of the TPM.

NOTE 1 In all TPM, a *buffer* size of 1,024 octets is allowed.

Proper authorization for the sequence object associated with *sequenceHandle* is required. If an authorization or audit of this command requires computation of a *cpHash* and an *rpHash*, the Name associated with *sequenceHandle* will be the Empty Buffer.

If the command does not return TPM_RC_SUCCESS, the state of the sequence is unmodified.

If the sequence is intended to produce a digest that will be signed by a restricted signing key, then the first block of data shall contain sizeof(TPM_GENERATED) octets and the first octets shall not be TPM_GENERATED_VALUE.

NOTE 2 This requirement allows the TPM to validate that the first block is safe to sign without having to accumulate octets over multiple calls.

 Copyright © TCG 2006-2020

17.5.2 Command and Response

Table 80 — TPM2_SequenceUpdate Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_SequenceUpdate
TPMI_DH_OBJECT	@sequenceHandle	handle for the sequence object Auth Index: 1 Auth Role: USER
TPM2B_MAX_BUFFER	buffer	data to be added to hash

Table 81 — TPM2_SequenceUpdate Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

17.5.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "SequenceUpdate_fp.h"
3   #if CC_SequenceUpdate  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_MODE	*sequenceHandle* does not reference a hash or HMAC sequence object

```
4   TPM_RC
5   TPM2_SequenceUpdate(
6       SequenceUpdate_In   *in              // IN: input parameter list
7       )
8   {
9       OBJECT                  *object;
10      HASH_OBJECT             *hashObject;
11
12  // Input Validation
13
14      // Get sequence object pointer
15      object = HandleToObject(in->sequenceHandle);
16      hashObject = (HASH_OBJECT *)object;
17
18      // Check that referenced object is a sequence object.
19      if(!ObjectIsSequence(object))
20          return TPM_RCS_MODE + RC_SequenceUpdate_sequenceHandle;
21
22  // Internal Data Update
23
24      if(object->attributes.eventSeq == SET)
25      {
26          // Update event sequence object
27          UINT32          i;
28          for(i = 0; i < HASH_COUNT; i++)
29          {
30              // Update sequence object
31              CryptDigestUpdate2B(&hashObject->state.hashState[i], &in->buffer.b);
32          }
33      }
34      else
35      {
36          // Update hash/HMAC sequence object
37          if(hashObject->attributes.hashSeq == SET)
38          {
39              // Is this the first block of the sequence
40              if(hashObject->attributes.firstBlock == CLEAR)
41              {
42                  // If so, indicate that first block was received
43                  hashObject->attributes.firstBlock = SET;
44
45                  // Check the first block to see if the first block can contain
46                  // the TPM_GENERATED_VALUE.  If it does, it is not safe for
47                  // a ticket.
48                  if(TicketIsSafe(&in->buffer.b))
49                      hashObject->attributes.ticketSafe = SET;
50              }
51              // Update sequence object hash/HMAC stack
52              CryptDigestUpdate2B(&hashObject->state.hashState[0], &in->buffer.b);
53          }
54          else if(object->attributes.hmacSeq == SET)
55          {
```

```
56              // Update sequence object HMAC stack
57              CryptDigestUpdate2B(&hashObject->state.hmacState.hashState,
58                                  &in->buffer.b);
59          }
60      }
61      return TPM_RC_SUCCESS;
62  }
63  #endif // CC_SequenceUpdate
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

17.6 TPM2_SequenceComplete

17.6.1 General Description

This command adds the last part of data, if any, to a hash/HMAC sequence and returns the result.

NOTE 1 This command is not used to complete an Event Sequence. TPM2_EventSequenceComplete() is used for that purpose.

For a hash sequence, if the results of the hash will be used in a signing operation that uses a restricted signing key, then the ticket returned by this command can indicate that the hash is safe to sign.

If the *digest* is not safe to sign, then *validation* will be a TPMT_TK_HASHCHECK with the hierarchy set to TPM_RH_NULL and *digest* set to the Empty Buffer.

If *hierarchy* is TPM_RH_NULL, then *digest* in the ticket will be the Empty Buffer.

NOTE 2 Regardless of the contents of the first octets of the hashed message, if the first buffer sent to the TPM had fewer than sizeof(TPM_GENERATED) octets, then the TPM will operate as if *digest* is not safe to sign.

NOTE 3 The ticket is only required for a signing operation that uses a restricted signing key. It is always returned, but can be ignored if not needed.

If *sequenceHandle* references an Event Sequence, then the TPM shall return TPM_RC_MODE.

Proper authorization for the sequence object associated with *sequenceHandle* is required. If an authorization or audit of this command requires computation of a *cpHash* and an *rpHash*, the Name associated with *sequenceHandle* will be the Empty Buffer.

If this command completes successfully, the *sequenceHandle* object will be flushed.

 Copyright © TCG 2006-2020

17.6.2 Command and Response

Table 82 — TPM2_SequenceComplete Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_SequenceComplete {F}
TPMI_DH_OBJECT	@sequenceHandle	authorization for the sequence Auth Index: 1 Auth Role: USER
TPM2B_MAX_BUFFER	buffer	data to be added to the hash/HMAC
TPMI_RH_HIERARCHY+	hierarchy	hierarchy of the ticket for a hash

Table 83 — TPM2_SequenceComplete Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	result	the returned HMAC or digest in a sized buffer
TPMT_TK_HASHCHECK	validation	ticket indicating that the sequence of octets used to compute *outDigest* did not start with TPM_GENERATED_VALUE This is a NULL Ticket when the sequence is HMAC.

 Copyright © TCG 2006-2020

17.6.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "SequenceComplete_fp.h"
3  #if CC_SequenceComplete  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_MODE	*sequenceHandle* does not reference a hash or HMAC sequence object

```
4  TPM_RC
5  TPM2_SequenceComplete(
6      SequenceComplete_In      *in,            // IN: input parameter list
7      SequenceComplete_Out     *out            // OUT: output parameter list
8      )
9  {
10     HASH_OBJECT                      *hashObject;
11 // Input validation
12     // Get hash object pointer
13     hashObject = (HASH_OBJECT *)HandleToObject(in->sequenceHandle);
14
15     // input handle must be a hash or HMAC sequence object.
16     if(hashObject->attributes.hashSeq == CLEAR
17        && hashObject->attributes.hmacSeq == CLEAR)
18         return TPM_RCS_MODE + RC_SequenceComplete_sequenceHandle;
19 // Command Output
20     if(hashObject->attributes.hashSeq == SET)           // sequence object for hash
21     {
22        // Get the hash algorithm before the algorithm is lost in CryptHashEnd
23         TPM_ALG_ID       hashAlg = hashObject->state.hashState[0].hashAlg;
24
25         // Update last piece of the data
26         CryptDigestUpdate2B(&hashObject->state.hashState[0], &in->buffer.b);
27
28         // Complete hash
29         out->result.t.size = CryptHashEnd(&hashObject->state.hashState[0],
30                                           sizeof(out->result.t.buffer),
31                                           out->result.t.buffer);
32         // Check if the first block of the sequence has been received
33         if(hashObject->attributes.firstBlock == CLEAR)
34         {
35             // If not, then this is the first block so see if it is 'safe'
36             // to sign.
37             if(TicketIsSafe(&in->buffer.b))
38                 hashObject->attributes.ticketSafe = SET;
39         }
40         // Output ticket
41         out->validation.tag = TPM_ST_HASHCHECK;
42         out->validation.hierarchy = in->hierarchy;
43
44         if(in->hierarchy == TPM_RH_NULL)
45         {
46             // Ticket is not required
47             out->validation.digest.t.size = 0;
48         }
49         else if(hashObject->attributes.ticketSafe == CLEAR)
50         {
51             // Ticket is not safe to generate
52             out->validation.hierarchy = TPM_RH_NULL;
53             out->validation.digest.t.size = 0;
54         }
55         else
```

```
56            {
57                // Compute ticket
58                TicketComputeHashCheck(out->validation.hierarchy, hashAlg,
59                                       &out->result, &out->validation);
60            }
61        }
62        else
63        {
64            //   Update last piece of data
65            CryptDigestUpdate2B(&hashObject->state.hmacState.hashState, &in->buffer.b);
66    #if !SMAC_IMPLEMENTED
67            // Complete HMAC
68            out->result.t.size = CryptHmacEnd(&(hashObject->state.hmacState),
69                                              sizeof(out->result.t.buffer),
70                                              out->result.t.buffer);
71    #else
72            // Complete the MAC
73            out->result.t.size = CryptMacEnd(&hashObject->state.hmacState,
74                                             sizeof(out->result.t.buffer),
75                                             out->result.t.buffer);
76    #endif
77            // No ticket is generated for HMAC sequence
78            out->validation.tag = TPM_ST_HASHCHECK;
79            out->validation.hierarchy = TPM_RH_NULL;
80            out->validation.digest.t.size = 0;
81        }
82    // Internal Data Update
83        // mark sequence object as evict so it will be flushed on the way out
84        hashObject->attributes.evict = SET;
85
86        return TPM_RC_SUCCESS;
87    }
88    #endif // CC_SequenceComplete
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

17.7 TPM2_EventSequenceComplete

17.7.1 General Description

This command adds the last part of data, if any, to an Event Sequence and returns the result in a digest list. If *pcrHandle* references a PCR and not TPM_RH_NULL, then the returned digest list is processed in the same manner as the digest list input parameter to TPM2_PCR_Extend(). That is, if a bank contains a PCR associated with *pcrHandle,* it is extended with the associated digest value from the list.

If *sequenceHandle* references a hash or HMAC sequence, the TPM shall return TPM_RC_MODE.

Proper authorization for the sequence object associated with *sequenceHandle* is required. If an authorization or audit of this command requires computation of a *cpHash* and an *rpHash*, the Name associated with *sequenceHandle* will be the Empty Buffer.

If this command completes successfully, the *sequenceHandle* object will be flushed.

NOTE: Unlike TPM2_PCR_Event(), a digest is always returned for each implemented hash algorithm. There is no option to only return digests for which *pcrHandle* is allocated.

 Copyright © TCG 2006-2020

17.7.2 Command and Response

Table 84 — TPM2_EventSequenceComplete Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_EventSequenceComplete {NV F}
TPMI_DH_PCR+	@pcrHandle	PCR to be extended with the Event data Auth Index: 1 Auth Role: USER
TPMI_DH_OBJECT	@sequenceHandle	authorization for the sequence Auth Index: 2 Auth Role: USER
TPM2B_MAX_BUFFER	buffer	data to be added to the Event

Table 85 — TPM2_EventSequenceComplete Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPML_DIGEST_VALUES	results	list of digests computed for the PCR

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

17.7.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "EventSequenceComplete_fp.h"
3  #if CC_EventSequenceComplete  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_LOCALITY	PCR extension is not allowed at the current locality
TPM_RC_MODE	input handle is not a valid event sequence object

```
4  TPM_RC
5  TPM2_EventSequenceComplete(
6      EventSequenceComplete_In    *in,             // IN: input parameter list
7      EventSequenceComplete_Out   *out             // OUT: output parameter list
8      )
9  {
10     HASH_OBJECT          *hashObject;
11     UINT32                i;
12     TPM_ALG_ID            hashAlg;
13 // Input validation
14     // get the event sequence object pointer
15     hashObject = (HASH_OBJECT *)HandleToObject(in->sequenceHandle);
16
17     // input handle must reference an event sequence object
18     if(hashObject->attributes.eventSeq != SET)
19         return TPM_RCS_MODE + RC_EventSequenceComplete_sequenceHandle;
20
21     // see if a PCR extend is requested in call
22     if(in->pcrHandle != TPM_RH_NULL)
23     {
24         // see if extend of the PCR is allowed at the locality of the command,
25         if(!PCRIsExtendAllowed(in->pcrHandle))
26             return TPM_RC_LOCALITY;
27         // if an extend is going to take place, then check to see if there has
28         // been an orderly shutdown. If so, and the selected PCR is one of the
29         // state saved PCR, then the orderly state has to change. The orderly state
30         // does not change for PCR that are not preserved.
31         // NOTE: This doesn't just check for Shutdown(STATE) because the orderly
32         // state will have to change if this is a state-saved PCR regardless
33         // of the current state. This is because a subsequent Shutdown(STATE) will
34         // check to see if there was an orderly shutdown and not do anything if
35         // there was. So, this must indicate that a future Shutdown(STATE) has
36         // something to do.
37         if(PCRIsStateSaved(in->pcrHandle))
38             RETURN_IF_ORDERLY;
39     }
40 // Command Output
41     out->results.count = 0;
42
43     for(i = 0; i < HASH_COUNT; i++)
44     {
45         hashAlg = CryptHashGetAlgByIndex(i);
46         // Update last piece of data
47         CryptDigestUpdate2B(&hashObject->state.hashState[i], &in->buffer.b);
48         // Complete hash
49         out->results.digests[out->results.count].hashAlg = hashAlg;
50         CryptHashEnd(&hashObject->state.hashState[i],
51                      CryptHashGetDigestSize(hashAlg),
52                      (BYTE *)&out->results.digests[out->results.count].digest);
53      // Extend PCR
54         if(in->pcrHandle != TPM_RH_NULL)
```

```
55                PCRExtend(in->pcrHandle, hashAlg,
56                          CryptHashGetDigestSize(hashAlg),
57                          (BYTE *)&out->results.digests[out->results.count].digest);
58            out->results.count++;
59        }
60    // Internal Data Update
61        // mark sequence object as evict so it will be flushed on the way out
62        hashObject->attributes.evict = SET;
63
64        return TPM_RC_SUCCESS;
65    }
66    #endif // CC_EventSequenceComplete
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

18 Attestation Commands

18.1 Introduction

The attestation commands cause the TPM to sign an internally generated data structure. The contents of the data structure vary according to the command.

If the *sign* attribute is not SET in the key referenced by *signHandle* then the TPM shall return TPM_RC_KEY.

All signing commands include a parameter (typically *inScheme*) for the caller to specify a scheme to be used for the signing operation. This scheme will be applied only if the scheme of the key is TPM_ALG_NULL or the key handle is TPM_RH_NULL. If the scheme for *signHandle* is not TPM_ALG_NULL, then *inScheme.scheme* shall be TPM_ALG_NULL or the same as *scheme* in the public area of the key. If the scheme for *signHandle* is TPM_ALG_NULL or the key handle is TPM_RH_NULL, then *inScheme* will be used for the signing operation and may not be TPM_ALG_NULL. The TPM shall return TPM_RC_SCHEME to indicate that the scheme is not appropriate.

For a signing key that is not restricted, the caller may specify the scheme to be used as long as the scheme is compatible with the family of the key (for example, TPM_ALG_RSAPSS cannot be selected for an ECC key). If the caller sets *scheme* to TPM_ALG_NULL, then the default scheme of the key is used. For a restricted signing key, the key's scheme cannot be TPM_ALG_NULL and cannot be overridden.

If the handle for the signing key (*signHandle*) is TPM_RH_NULL, then all of the actions of the command are performed and the attestation block is "signed" with the NULL Signature.

NOTE 1 This mechanism is provided so that additional commands are not required to access the data that might be in an attestation structure.

NOTE 2 When *signHandle* is TPM_RH_NULL, *scheme* is still required to be a valid signing scheme (may be TPM_ALG_NULL), but the scheme will have no effect on the format of the signature. It will always be the NULL Signature.

TPM2_NV_Certify() is an attestation command that is documented in 1. The remaining attestation commands are collected in the remainder of this clause.

Each of the attestation structures contains a TPMS_CLOCK_INFO structure and a firmware version number. These values may be considered privacy-sensitive, because they would aid in the correlation of attestations by different keys. To provide improved privacy, the *resetCount*, *restartCount*, and *firmwareVersion* numbers are obfuscated when the signing key is not in the Endorsement or Platform hierarchies.

The obfuscation value is computed by:

$$obfuscation \coloneqq \mathbf{KDFa}(signHandle{\rightarrow}nameAlg, shProof, \text{“OBFUSCATE”}, signHandle{\rightarrow}QN, 0, 128) \quad (3)$$

Of the returned 128 bits, 64 bits are added to the *versionNumber* field of the attestation structure; 32 bits are added to the *clockInfo.resetCount* and 32 bits are added to the *clockInfo.restartCount*. The order in which the bits are added is implementation-dependent.

NOTE 3 The obfuscation value for each signing key will be unique to that key in a specific location. That is, each version of a duplicated signing key will have a different obfuscation value.

When the signing key is TPM_RH_NULL, the data structure is produced but not signed; and the values in the signed data structure are obfuscated. When computing the obfuscation value for TPM_RH_NULL, the hash used for context integrity is used.

NOTE 4 The QN for TPM_RH_NULL is TPM_RH_NULL.

If the signing scheme of *signHandle* is an anonymous scheme, then the attestation blocks will not contain the Qualified Name of the *signHandle*.

 Copyright © TCG 2006-2020

Each of the attestation structures allows the caller to provide some qualifying data (*qualifyingData*). For most signing schemes, this value will be placed in the TPMS_ATTEST.*extraData* parameter that is then hashed and signed. However, for some schemes such as ECDAA, the *qualifyingData* is used in a different manner (for details, see “ECDAA” in TPM 2.0 Part 1).

 Copyright © TCG 2006-2020

18.2 TPM2_Certify

18.2.1 General Description

The purpose of this command is to prove that an object with a specific Name is loaded in the TPM. By certifying that the object is loaded, the TPM warrants that a public area with a given Name is self-consistent and associated with a valid sensitive area. If a relying party has a public area that has the same Name as a Name certified with this command, then the values in that public area are correct.

NOTE 1 See 18.1 for description of how the signing scheme is selected.

Authorization for *objectHandle* requires ADMIN role authorization. If performed with a policy session, the session shall have a policySession→*commandCode* set to TPM_CC_Certify. This indicates that the policy that is being used is a policy that is for certification, and not a policy that would approve another use. That is, authority to use an object does not grant authority to certify the object.

The object may be any object that is loaded with TPM2_Load() or TPM2_CreatePrimary(). An object that only has its public area loaded cannot be certified.

NOTE 2 The restriction occurs because the Name is used to identify the object being certified. If the TPM has not validated that the public area is associated with a matched sensitive area, then the public area may not represent a valid object and cannot be certified.

The certification includes the Name and Qualified Name of the certified object as well as the Name and the Qualified Name of the certifying object.

NOTE 3 If *signHandle* is TPM_RH_NULL, the TPMS_ATTEST structure is returned and *signature* is a NULL Signature.

 Copyright © TCG 2006-2020

18.2.2 Command and Response

Table 86 — TPM2_Certify Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Certify
TPMI_DH_OBJECT	@objectHandle	handle of the object to be certified Auth Index: 1 Auth Role: ADMIN
TPMI_DH_OBJECT+	@signHandle	handle of the key used to sign the attestation structure Auth Index: 2 Auth Role: USER
TPM2B_DATA	qualifyingData	user provided qualifying data
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL

Table 87 — TPM2_Certify Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPM2B_ATTEST	certifyInfo	the structure that was signed
TPMT_SIGNATURE	signature	the asymmetric signature over *certifyInfo* using the key referenced by *signHandle*

 Copyright © TCG 2006-2020

18.2.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "Attest_spt_fp.h"
3   #include "Certify_fp.h"
4   #if CC_Certify  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_KEY	key referenced by *signHandle* is not a signing key
TPM_RC_SCHEME	*inScheme* is not compatible with *signHandle*
TPM_RC_VALUE	digest generated for *inScheme* is greater or has larger size than the modulus of *signHandle*, or the buffer for the result in *signature* is too small (for an RSA key); invalid commit status (for an ECC key with a split scheme)

```
5   TPM_RC
6   TPM2_Certify(
7       Certify_In      *in,            // IN: input parameter list
8       Certify_Out     *out            // OUT: output parameter list
9       )
10  {
11      TPMS_ATTEST             certifyInfo;
12      OBJECT                  *signObject = HandleToObject(in->signHandle);
13      OBJECT                  *certifiedObject = HandleToObject(in->objectHandle);
14  // Input validation
15      if(!IsSigningObject(signObject))
16          return TPM_RCS_KEY + RC_Certify_signHandle;
17      if(!CryptSelectSignScheme(signObject, &in->inScheme))
18          return TPM_RCS_SCHEME + RC_Certify_inScheme;
19
20  // Command Output
21      // Filling in attest information
22      // Common fields
23      FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData,
24                       &certifyInfo);
25
26      // Certify specific fields
27      certifyInfo.type = TPM_ST_ATTEST_CERTIFY;
28      // NOTE: the certified object is not allowed to be TPM_ALG_NULL so
29      // 'certifiedObject' will never be NULL
30      certifyInfo.attested.certify.name = certifiedObject->name;
31
32      // When using an anonymous signing scheme, need to set the qualified Name to the
33      // empty buffer to avoid correlation between keys
34      if(CryptIsSchemeAnonymous(in->inScheme.scheme))
35          certifyInfo.attested.certify.qualifiedName.t.size = 0;
36      else
37          certifyInfo.attested.certify.qualifiedName = certifiedObject->qualifiedName;
38
39      // Sign attestation structure.  A NULL signature will be returned if
40      // signHandle is TPM_RH_NULL.  A TPM_RC_NV_UNAVAILABLE, TPM_RC_NV_RATE,
41      // TPM_RC_VALUE, TPM_RC_SCHEME or TPM_RC_ATTRIBUTES error may be returned
42      // by SignAttestInfo()
43      return SignAttestInfo(signObject, &in->inScheme, &certifyInfo,
44                            &in->qualifyingData, &out->certifyInfo, &out->signature);
45  }
46  #endif // CC_Certify
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

18.3 TPM2_CertifyCreation

18.3.1 General Description

This command is used to prove the association between an object and its creation data. The TPM will validate that the ticket was produced by the TPM and that the ticket validates the association between a loaded public area and the provided hash of the creation data (*creationHash*).

NOTE 1 See 18.1 for description of how the signing scheme is selected.

The TPM will create a test ticket using the Name associated with *objectHandle* and *creationHash* as:

$$\mathbf{HMAC}(proof, (\mathrm{TPM_ST_CREATION} \mathbin{||} objectHandle{\rightarrow}Name \mathbin{||} creationHash)) \tag{4}$$

This ticket is then compared to creation ticket. If the tickets are not the same, the TPM shall return TPM_RC_TICKET.

If the ticket is valid, then the TPM will create a TPMS_ATTEST structure and place *creationHash* of the command in the *creationHash* field of the structure. The Name associated with *objectHandle* will be included in the attestation data that is then signed using the key associated with *signHandle*.

NOTE 2 If *signHandle* is TPM_RH_NULL, the TPMS_ATTEST structure is returned and *signature* is a NULL Signature.

objectHandle may be any object that is loaded with TPM2_Load() or TPM2_CreatePrimary().

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

18.3.2 Command and Response

Table 88 — TPM2_CertifyCreation Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_CertifyCreation
TPMI_DH_OBJECT+	@signHandle	handle of the key that will sign the attestation block Auth Index: 1 Auth Role: USER
TPMI_DH_OBJECT	objectHandle	the object associated with the creation data Auth Index: None
TPM2B_DATA	qualifyingData	user-provided qualifying data
TPM2B_DIGEST	creationHash	hash of the creation data produced by TPM2_Create() or TPM2_CreatePrimary()
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL
TPMT_TK_CREATION	creationTicket	ticket produced by TPM2_Create() or TPM2_CreatePrimary()

Table 89 — TPM2_CertifyCreation Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ATTEST	certifyInfo	the structure that was signed
TPMT_SIGNATURE	signature	the signature over *certifyInfo*

 Copyright © TCG 2006-2020

18.3.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "Attest_spt_fp.h"
3  #include "CertifyCreation_fp.h"
4  #if CC_CertifyCreation  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_KEY	key referenced by *signHandle* is not a signing key
TPM_RC_SCHEME	*inScheme* is not compatible with *signHandle*
TPM_RC_TICKET	*creationTicket* does not match *objectHandle*
TPM_RC_VALUE	digest generated for *inScheme* is greater or has larger size than the modulus of *signHandle*, or the buffer for the result in *signature* is too small (for an RSA key); invalid commit status (for an ECC key with a split scheme).

```
5  TPM_RC
6  TPM2_CertifyCreation(
7      CertifyCreation_In      *in,            // IN: input parameter list
8      CertifyCreation_Out     *out            // OUT: output parameter list
9      )
10 {
11     TPMT_TK_CREATION        ticket;
12     TPMS_ATTEST             certifyInfo;
13     OBJECT                  *certified = HandleToObject(in->objectHandle);
14     OBJECT                  *signObject = HandleToObject(in->signHandle);
15 // Input Validation
16     if(!IsSigningObject(signObject))
17         return TPM_RCS_KEY + RC_CertifyCreation_signHandle;
18     if(!CryptSelectSignScheme(signObject, &in->inScheme))
19         return TPM_RCS_SCHEME + RC_CertifyCreation_inScheme;
20
21     // CertifyCreation specific input validation
22     // Re-compute ticket
23     TicketComputeCreation(in->creationTicket.hierarchy, &certified->name,
24                           &in->creationHash, &ticket);
25     // Compare ticket
26     if(!MemoryEqual2B(&ticket.digest.b, &in->creationTicket.digest.b))
27         return TPM_RCS_TICKET + RC_CertifyCreation_creationTicket;
28
29 // Command Output
30     // Common fields
31     FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData,
32                      &certifyInfo);
33
34     // CertifyCreation specific fields
35     // Attestation type
36     certifyInfo.type = TPM_ST_ATTEST_CREATION;
37     certifyInfo.attested.creation.objectName = certified->name;
38
39     // Copy the creationHash
40     certifyInfo.attested.creation.creationHash = in->creationHash;
41
42     // Sign attestation structure.  A NULL signature will be returned if
43     // signObject is TPM_RH_NULL.  A TPM_RC_NV_UNAVAILABLE, TPM_RC_NV_RATE,
44     // TPM_RC_VALUE, TPM_RC_SCHEME or TPM_RC_ATTRIBUTES error may be returned at
45     // this point
46     return SignAttestInfo(signObject, &in->inScheme, &certifyInfo,
47                           &in->qualifyingData, &out->certifyInfo,
48                           &out->signature);
```

 Copyright © TCG 2006-2020

```
49  }
50  #endif // CC_CertifyCreation
```

18.4 TPM2_Quote

18.4.1 General Description

This command is used to quote PCR values.

The TPM will hash the list of PCR selected by *PCRselect* using the hash algorithm in the selected signing scheme. If the selected signing scheme or the scheme hash algorithm is TPM_ALG_NULL, then the TPM shall return TPM_RC_SCHEME.

NOTE 1 See 18.1 for description of how the signing scheme is selected.

The digest is computed as the hash of the concatenation of all of the digest values of the selected PCR.

The concatenation of PCR is described in TPM 2.0 Part 1, *Selecting Multiple PCR.*

NOTE 2 If *signHandle* is TPM_RH_NULL, the TPMS_ATTEST structure is returned and *signature* is a NULL Signature.

NOTE 3 A TPM may optionally return TPM_RC_SCHEME if *signHandle* is TPM_RH_NULL.

NOTE 4 Unlike TPM 1.2, TPM2_Quote does not return the PCR values. See Part 1, “Attesting to PCR” for a discussion of this issue.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

18.4.2 Command and Response

Table 90 — TPM2_Quote Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Quote
TPMI_DH_OBJECT+	@signHandle	handle of key that will perform signature Auth Index: 1 Auth Role: USER
TPM2B_DATA	qualifyingData	data supplied by the caller
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL
TPML_PCR_SELECTION	PCRselect	PCR set to quote

Table 91 — TPM2_Quote Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ATTEST	quoted	the quoted information
TPMT_SIGNATURE	signature	the signature over *quoted*

 Copyright © TCG 2006-2020

18.4.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "Attest_spt_fp.h"
3   #include "Quote_fp.h"
4   #if CC_Quote  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_KEY	*signHandle* does not reference a signing key;
TPM_RC_SCHEME	the scheme is not compatible with sign key type, or input scheme is not compatible with default scheme, or the chosen scheme is not a valid sign scheme

```
5   TPM_RC
6   TPM2_Quote(
7       Quote_In        *in,             // IN: input parameter list
8       Quote_Out       *out             // OUT: output parameter list
9       )
10  {
11      TPMI_ALG_HASH            hashAlg;
12      TPMS_ATTEST              quoted;
13      OBJECT                  *signObject = HandleToObject(in->signHandle);
14  // Input Validation
15      if(!IsSigningObject(signObject))
16          return TPM_RCS_KEY + RC_Quote_signHandle;
17      if(!CryptSelectSignScheme(signObject, &in->inScheme))
18          return TPM_RCS_SCHEME + RC_Quote_inScheme;
19
20  // Command Output
21
22      // Filling in attest information
23      // Common fields
24      // FillInAttestInfo may return TPM_RC_SCHEME or TPM_RC_KEY
25      FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData, &quoted);
26
27      // Quote specific fields
28      // Attestation type
29      quoted.type = TPM_ST_ATTEST_QUOTE;
30
31      // Get hash algorithm in sign scheme.  This hash algorithm is used to
32      // compute PCR digest. If there is no algorithm, then the PCR cannot
33      // be digested and this command returns TPM_RC_SCHEME
34      hashAlg = in->inScheme.details.any.hashAlg;
35
36      if(hashAlg == TPM_ALG_NULL)
37          return TPM_RCS_SCHEME + RC_Quote_inScheme;
38
39      // Compute PCR digest
40      PCRComputeCurrentDigest(hashAlg, &in->PCRselect,
41                              &quoted.attested.quote.pcrDigest);
42
43      // Copy PCR select.  "PCRselect" is modified in PCRComputeCurrentDigest
44      // function
45      quoted.attested.quote.pcrSelect = in->PCRselect;
46
47      // Sign attestation structure.  A NULL signature will be returned if
48      // signObject is NULL.
49      return SignAttestInfo(signObject, &in->inScheme, &quoted, &in->qualifyingData,
50                            &out->quoted, &out->signature);
51  }
52  #endif // CC_Quote
```

 Copyright © TCG 2006-2020

18.5 TPM2_GetSessionAuditDigest

18.5.1 General Description

This command returns a digital signature of the audit session digest.

NOTE 1 See 18.1 for description of how the signing scheme is selected.

If *sessionHandle* is not an audit session, the TPM shall return TPM_RC_TYPE.

NOTE 2 A session does not become an audit session until the successful completion of the command in which the session is first used as an audit session.

This command requires authorization from the privacy administrator of the TPM (expressed with Endorsement Authorization) as well as authorization to use the key associated with *signHandle*.

If this command is audited, then the audit digest that is signed will not include the digest of this command because the audit digest is only updated when the command completes successfully.

This command does not cause the audit session to be closed and does not reset the digest value.

NOTE 3 If sessionHandle is used as an audit session for this command, the command is audited in the same manner as any other command.

NOTE 4 If *signHandle* is TPM_RH_NULL, the TPMS_ATTEST structure is returned and *signature* is a NULL Signature.

 Copyright © TCG 2006-2020

18.5.2 Command and Response

Table 92 — TPM2_GetSessionAuditDigest Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_GetSessionAuditDigest
TPMI_RH_ENDORSEMENT	@privacyAdminHandle	handle of the privacy administrator (TPM_RH_ENDORSEMENT) Auth Index: 1 Auth Role: USER
TPMI_DH_OBJECT+	@signHandle	handle of the signing key Auth Index: 2 Auth Role: USER
TPMI_SH_HMAC	sessionHandle	handle of the audit session Auth Index: None
TPM2B_DATA	qualifyingData	user-provided qualifying data – may be zero-length
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL

Table 93 — TPM2_GetSessionAuditDigest Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ATTEST	auditInfo	the audit information that was signed
TPMT_SIGNATURE	signature	the signature over *auditInfo*

 Copyright © TCG 2006-2020

18.5.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "Attest_spt_fp.h"
3  #include "GetSessionAuditDigest_fp.h"
4  #if CC_GetSessionAuditDigest  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_KEY	key referenced by *signHandle* is not a signing key
TPM_RC_SCHEME	*inScheme* is incompatible with *signHandle* type; or both *scheme* and key's default scheme are empty; or *scheme* is empty while key's default scheme requires explicit input scheme (split signing); or non-empty default key scheme differs from *scheme*
TPM_RC_TYPE	*sessionHandle* does not reference an audit session
TPM_RC_VALUE	digest generated for the given *scheme* is greater than the modulus of *signHandle* (for an RSA key); invalid commit status or failed to generate r value (for an ECC key)

```
5  TPM_RC
6  TPM2_GetSessionAuditDigest(
7      GetSessionAuditDigest_In    *in,            // IN: input parameter list
8      GetSessionAuditDigest_Out   *out            // OUT: output parameter list
9      )
10 {
11     SESSION                  *session = SessionGet(in->sessionHandle);
12     TPMS_ATTEST               auditInfo;
13     OBJECT                  *signObject = HandleToObject(in->signHandle);
14 // Input Validation
15     if(!IsSigningObject(signObject))
16         return TPM_RCS_KEY + RC_GetSessionAuditDigest_signHandle;
17     if(!CryptSelectSignScheme(signObject, &in->inScheme))
18         return TPM_RCS_SCHEME + RC_GetSessionAuditDigest_inScheme;
19
20     // session must be an audit session
21     if(session->attributes.isAudit == CLEAR)
22         return TPM_RCS_TYPE + RC_GetSessionAuditDigest_sessionHandle;
23
24 // Command Output
25     // Fill in attest information common fields
26     FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData,
27                      &auditInfo);
28
29     // SessionAuditDigest specific fields
30     auditInfo.type = TPM_ST_ATTEST_SESSION_AUDIT;
31     auditInfo.attested.sessionAudit.sessionDigest = session->u2.auditDigest;
32
33     // Exclusive audit session
34     auditInfo.attested.sessionAudit.exclusiveSession
35         = (g_exclusiveAuditSession == in->sessionHandle);
36
37     // Sign attestation structure.  A NULL signature will be returned if
38     // signObject is NULL.
39     return SignAttestInfo(signObject, &in->inScheme, &auditInfo,
40                           &in->qualifyingData, &out->auditInfo,
41                           &out->signature);
42 }
43 #endif // CC_GetSessionAuditDigest
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

18.6 TPM2_GetCommandAuditDigest

18.6.1 General Description

This command returns the current value of the command audit digest, a digest of the commands being audited, and the audit hash algorithm. These values are placed in an attestation structure and signed with the key referenced by *signHandle*.

NOTE 1 See 18.1 for description of how the signing scheme is selected.

When this command completes successfully, and *signHandle* is not TPM_RH_NULL, the audit digest is cleared. If signHandle is TPM_RH_NULL, *signature* is the Empty Buffer and the audit digest is not cleared.

NOTE 2 The way that the TPM tracks that the digest is clear is vendor-dependent. The reference implementation resets the size of the digest to zero.

If this command is being audited, then the signed digest produced by the command will not include the command. At the end of this command, the audit digest will be extended with *cpHash* and the *rpHash* of the command, which would change the command audit digest signed by the next invocation of this command.

This command requires authorization from the privacy administrator of the TPM (expressed with Endorsement Authorization) as well as authorization to use the key associated with *signHandle*.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

18.6.2 Command and Response

Table 94 — TPM2_GetCommandAuditDigest Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_GetCommandAuditDigest {NV}
TPMI_RH_ENDORSEMENT	@privacyHandle	handle of the privacy administrator (TPM_RH_ENDORSEMENT) Auth Index: 1 Auth Role: USER
TPMI_DH_OBJECT+	@signHandle	the handle of the signing key Auth Index: 2 Auth Role: USER
TPM2B_DATA	qualifyingData	other data to associate with this audit digest
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL

Table 95 — TPM2_GetCommandAuditDigest Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ATTEST	auditInfo	the auditInfo that was signed
TPMT_SIGNATURE	signature	the signature over *auditInfo*

 Copyright © TCG 2006-2020

18.6.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "Attest_spt_fp.h"
3  #include "GetCommandAuditDigest_fp.h"
4  #if CC_GetCommandAuditDigest  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_KEY	key referenced by *signHandle* is not a signing key
TPM_RC_SCHEME	*inScheme* is incompatible with *signHandle* type; or both *scheme* and key's default scheme are empty; or *scheme* is empty while key's default scheme requires explicit input scheme (split signing); or non-empty default key scheme differs from *scheme*
TPM_RC_VALUE	digest generated for the given *scheme* is greater than the modulus of *signHandle* (for an RSA key); invalid commit status or failed to generate r value (for an ECC key)

```
5  TPM_RC
6  TPM2_GetCommandAuditDigest(
7      GetCommandAuditDigest_In    *in,            // IN: input parameter list
8      GetCommandAuditDigest_Out   *out            // OUT: output parameter list
9      )
10 {
11     TPM_RC                  result;
12     TPMS_ATTEST             auditInfo;
13     OBJECT                 *signObject = HandleToObject(in->signHandle);
14 // Input validation
15     if(!IsSigningObject(signObject))
16         return TPM_RCS_KEY + RC_GetCommandAuditDigest_signHandle;
17     if(!CryptSelectSignScheme(signObject, &in->inScheme))
18         return TPM_RCS_SCHEME + RC_GetCommandAuditDigest_inScheme;
19
20 // Command Output
21     // Fill in attest information common fields
22     FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData,
23                      &auditInfo);
24
25     // CommandAuditDigest specific fields
26     auditInfo.type = TPM_ST_ATTEST_COMMAND_AUDIT;
27     auditInfo.attested.commandAudit.digestAlg = gp.auditHashAlg;
28     auditInfo.attested.commandAudit.auditCounter = gp.auditCounter;
29
30     // Copy command audit log
31     auditInfo.attested.commandAudit.auditDigest = gr.commandAuditDigest;
32     CommandAuditGetDigest(&auditInfo.attested.commandAudit.commandDigest);
33
34     // Sign attestation structure.  A NULL signature will be returned if
35     // signHandle is TPM_RH_NULL.  A TPM_RC_NV_UNAVAILABLE, TPM_RC_NV_RATE,
36     // TPM_RC_VALUE, TPM_RC_SCHEME or TPM_RC_ATTRIBUTES error may be returned at
37     // this point
38     result = SignAttestInfo(signObject, &in->inScheme, &auditInfo,
39                             &in->qualifyingData, &out->auditInfo,
40                             &out->signature);
41     // Internal Data Update
42     if(result == TPM_RC_SUCCESS && in->signHandle != TPM_RH_NULL)
43         // Reset log
44         gr.commandAuditDigest.t.size = 0;
45
46     return result;
47 }
```

 Copyright © TCG 2006-2020

```
48 #endif // CC_GetCommandAuditDigest
```

18.7 TPM2_GetTime

18.7.1 General Description

This command returns the current values of *Time* and *Clock*.

NOTE 1 See 18.1 for description of how the signing scheme is selected.

The values of *Clock*, *resetCount* and *restartCount* appear in two places in *timeInfo*: once in TPMS_ATTEST.*clockInfo* and again in TPMS_ATTEST.*attested.time.clockInfo.* The firmware version number also appears in two places (TPMS_ATTEST.*firmwareVersion* and TPMS_ATTEST.*attested.time.firmwareVersion*). If *signHandle* is in the endorsement or platform hierarchies, both copies of the data will be the same. However, if *signHandle* is in the storage hierarchy or is TPM_RH_NULL, the values in TPMS_ATTEST.*clockInfo* and TPMS_ATTEST.*firmwareVersion* are obfuscated but the values in TPMS_ATTEST.*attested.time* are not.

NOTE 2 The purpose of this duplication is to allow an entity who is trusted by the privacy Administrator to correlate the obfuscated values with the clear-text values. This command requires Endorsement Authorization.

NOTE 3 If *signHandle* is TPM_RH_NULL, the TPMS_ATTEST structure is returned and *signature* is a NULL Signature.

 Copyright © TCG 2006-2020

18.7.2 Command and Response

Table 96 — TPM2_GetTime Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_GetTime
TPMI_RH_ENDORSEMENT	@privacyAdminHandle	handle of the privacy administrator (TPM_RH_ENDORSEMENT) Auth Index: 1 Auth Role: USER
TPMI_DH_OBJECT+	@signHandle	the *keyHandle* identifier of a loaded key that can perform digital signatures Auth Index: 2 Auth Role: USER
TPM2B_DATA	qualifyingData	data to tick stamp
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL

Table 97 — TPM2_GetTime Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPM2B_ATTEST	timeInfo	standard TPM-generated attestation block
TPMT_SIGNATURE	signature	the signature over *timeInfo*

 Copyright © TCG 2006-2020

18.7.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "Attest_spt_fp.h"
3   #include "GetTime_fp.h"
4   #if CC_GetTime  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_KEY	key referenced by *signHandle* is not a signing key
TPM_RC_SCHEME	*inScheme* is incompatible with *signHandle* type; or both *scheme* and key's default scheme are empty; or *scheme* is empty while key's default scheme requires explicit input scheme (split signing); or non-empty default key scheme differs from *scheme*
TPM_RC_VALUE	digest generated for the given *scheme* is greater than the modulus of *signHandle* (for an RSA key); invalid commit status or failed to generate r value (for an ECC key)

```
5   TPM_RC
6   TPM2_GetTime(
7       GetTime_In      *in,            // IN: input parameter list
8       GetTime_Out     *out            // OUT: output parameter list
9       )
10  {
11      TPMS_ATTEST              timeInfo;
12      OBJECT                  *signObject = HandleToObject(in->signHandle);
13  // Input Validation
14      if(!IsSigningObject(signObject))
15          return TPM_RCS_KEY + RC_GetTime_signHandle;
16      if(!CryptSelectSignScheme(signObject, &in->inScheme))
17          return TPM_RCS_SCHEME + RC_GetTime_inScheme;
18
19  // Command Output
20      // Fill in attest common fields
21      FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData, &timeInfo);
22
23      // GetClock specific fields
24      timeInfo.type = TPM_ST_ATTEST_TIME;
25      timeInfo.attested.time.time.time = g_time;
26      TimeFillInfo(&timeInfo.attested.time.time.clockInfo);
27
28      // Firmware version in plain text
29      timeInfo.attested.time.firmwareVersion
30          = (((UINT64)gp.firmwareV1) << 32) + gp.firmwareV2;
31
32      // Sign attestation structure.  A NULL signature will be returned if
33      // signObject is NULL.
34      return SignAttestInfo(signObject, &in->inScheme, &timeInfo, &in->qualifyingData,
35                            &out->timeInfo, &out->signature);
36  }
37  #endif // CC_GetTime
```

18.8 TPM2_CertifyX509

18.8.1 General Description

The purpose of this command is to generate an X.509 certificate that proves an object with a specific public key and attributes is loaded in the TPM. In contrast to TPM2_Certify, which uses a TCG-defined data structure to convey attestation information, TPM2_CertifyX509 encodes the attestation information in

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

a DER-encoded X.509 certificate that is compliant with RFC5280 *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.*

As described in RFC, an X.509 certificate contains a collection of data that is hashed and signed. The full signature is the combination of the *to be signed* (TBS) data, a description of the signature algorithm, and the signature over the TBS data. The elements of the TBS data structure are DER-encoded values. They are:

1) Version [0] – integer value of 2 indicating version 3
2) Certificate Serial Number – integer value
3) Signature Algorithm Identifier – values (usually a collection of OIDs) identifying the algorithm used for the signature
4) Issuer Name – X.501 type *Name* to identify the entity that has authorized the use of *signHandle* to create the certificate.
5) Validity – two time values indicating the period during which the certificate is valid
6) Subject Name – X.501 type *Name* that identifies the entity that authorized the use of *objectHandle*
7) Subject Public Key Info – the public key associated with *objectHandle*,
8) Extensions [3] – a set of values that "provide methods for associating additional attributes with users or public keys and for managing relationships between CAs."

NOTE 1: The numbers in square brackets (e.g., [0]) indicate application-specific tag values that are used to identify the type of the field.

NOTE 2: RFC 5280 describes two fields (issuerUniqueID and subjectUniqueID) but goes on to say: "CAs conforming to this profile MUST NOT generate certificates with unique identifiers." The TPM does not allow them to be present.

The caller provides a partial certificate (*partialCertificate*) parameter that contains four or five of the elements enumerated above in a DER encoded SEQUENCE. They are:

1) Signature Algorithm Identifier (optional)
2) Issuer (mandatory)
3) Validity (mandatory)
4) Subject Name (mandatory)
5) Extensions (mandatory)

The fields are required to be in the order in which they are listed above.

NOTE 3: The TPM determines if the Signature Algorithm Identifier element is present by counting the elements.

The optional Signature Algorithm Identifier may be provided by the caller. If it is not present, the TPM will generate the value based on the selected signing scheme. If the caller provides this value, then the TPM will use it in the completed TBS. The TPM will not validate that the provided values are compatible with the signing scheme. If the caller does not provide this field and the TPM does not have OID values for the signing scheme, then the TPM will return an error (TPM_RC_SCHEME).

NOTE 4: The TPM may implement signing schemes for which OIDs are not defined at the time the TPM was manufactured. Those schemes may still be used if the caller can provide the Signature Algorithm Identifier.

The Extensions element is required to contain a Key Usage extension. The TPM will extract the Key Usage values and verify that the attributes of *objectHandle* are consistent with the selected values (TPM_RC_ATTRIBUTES)(See Part 2, *TPMA_X509_KEY_USAGE*).

The Extensions element may contain a TPMA_OBJECT extension. If present, the TPM will extract the value and verify that the extension value exactly matches the TPMA_OBJECT of *objectKey* (TPM_RC_ATTRIBUTES). The element uses the TCG OID tcg-tpmaObject, 2.23.133.10.1.1.1. It is a SEQUENCE containing that OID and an OCTET STRING encapsulating a 4-byte BIT STRING holding the big endian TPMA_OBJECT.

signHandle is required to have the *sign* attribute SET (TPM_RC_KEY).

NOTE 5: See 18.1 for description of how the signing scheme is selected.

Authorization for *objectHandle* requires ADMIN role authorization. If performed with a policy session, the session shall have a policySession→*commandCode* set to TPM_CC_CertifyX509. This indicates that the policy that is being used is a policy that is for certification, and not a policy that would approve another use. That is, authority to use an object does not grant authority to certify the object.

If *objectHandle* does not have a sensitive area loaded, the TPM will return an error (TPM_RC_AUTH_UNAVAILABLE).

NOTE 6: The command requires that authorization be provided for use of *objectHandle.* An object that only has its *publicArea* loaded does not have an authorization value and the *authPolicy* has no meaning as the sensitive area is not present.

The TPM will create the Version, the Certificate Serial Number, the Subject Public Key Info, and, if not provided by the caller, the Signature Algorithm Identifier. These TPM-created values will be combined with the provided values to make a full TBSCerfificate structure (See RFC 5280, clause 4.1). The TPM will then sign the certificate using the selected signing scheme.

The TPM-created values will be returned in *addedToCertificate.* If the TPM creates the Signature Algorithm Identifier, it will be in *addedToCertificate* before the Subject Public Key Info. The TPM returns *tbsDigest* as a debugging aid.

NOTE 7: These returned fields allow the caller to unambiguously create a full RFC5280-defined TBSCertificate.

NOTE 8: This command was added in revision 01.53.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

18.8.2 Command and Response

Table 98 — TPM2_CertifyX509 Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_CertifyX509
TPMI_DH_OBJECT	@objectHandle	handle of the object to be certified Auth Index: 1 Auth Role: ADMIN
TPMI_DH_OBJECT+	@signHandle	handle of the key used to sign the attestation structure Auth Index: 2 Auth Role: USER
TPM2B_DATA	reserved	shall be an Empty Buffer
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL
TPM2B_MAX_BUFFER	partialCertificate	a DER encoded partial certificate

Table 99 — TPM2_CertifyX509 Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPM2B_MAX_BUFFER	addedToCertificate	a DER encoded SEQUENCE containing the DER encoded fields added to partialCertificate to make it a complete RFC5280 TBSCertificate.
TPM2B_DIGEST	tbsDigest	the digest that was signed
TPMT_SIGNATURE	signature	The signature over *tbsDigest*

18.8.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "CertifyX509_fp.h"
3  #include "X509.h"
4  #include "TpmASN1_fp.h"
5  #include "X509_spt_fp.h"
6  #include "Attest_spt_fp.h"
7  #include "Platform_fp.h"
8  #if CC_CertifyX509 // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	the attributes of *objectHandle* are not compatible with the KeyUsage() or TPMA_OBJECT values in the extensions fields
TPM_RC_BINDING	the public and private portions of the key are not properly bound.
TPM_RC_HASH	the hash algorithm in the scheme is not supported
TPM_RC_KEY	*signHandle* does not reference a signing key;
TPM_RC_SCHEME	the scheme is not compatible with sign key type, or input scheme is not compatible with default scheme, or the chosen scheme is not a valid sign scheme
TPM_RC_VALUE	most likely a problem with the format of *partialCertificate*

```
9  TPM_RC
10 TPM2_CertifyX509(
11     CertifyX509_In          *in,         // IN: input parameter list
12     CertifyX509_Out         *out           // OUT: output parameter list
13 )
14 {
15     TPM_RC                   result;
16     OBJECT                  *signKey = HandleToObject(in->signHandle);
17     OBJECT                  *object = HandleToObject(in->objectHandle);
18     HASH_STATE               hash;
19     INT16                    length;        // length for a tagged element
20     ASN1UnmarshalContext     ctx;
21     ASN1MarshalContext       ctxOut;
22     // certTBS holds an array of pointers and lengths. Each entry references the
23     // corresponding value in a TBSCertificate structure. For example, the 1th
24     // element references the version number
25     stringRef                certTBS[REF_COUNT] = {{0}};
26 #define ALLOWED_SEQUENCES   (SUBJECT_PUBLIC_KEY_REF - SIGNATURE_REF)
27     stringRef                partial[ALLOWED_SEQUENCES] = {{0}};
28     INT16                    countOfSequences = 0;
29     INT16                    i;
30     //
31 #if CERTIFYX509_DEBUG
32     DebugFileOpen();
33     DebugDumpBuffer(in->partialCertificate.t.size, in->partialCertificate.t.buffer,
34         "partialCertificate");
35 #endif
36
37     // Input Validation
38     if(in->reserved.b.size != 0)
39         return TPM_RC_SIZE + RC_CertifyX509_reserved;
40     // signing key must be able to sign
41     if(!IsSigningObject(signKey))
42         return TPM_RCS_KEY + RC_CertifyX509_signHandle;
43     // Pick a scheme for sign.  If the input sign scheme is not compatible with
44     // the default scheme, return an error.
```

 Copyright © TCG 2006-2020

```
45       if(!CryptSelectSignScheme(signKey, &in->inScheme))
46           return TPM_RCS_SCHEME + RC_CertifyX509_inScheme;
47       // Make sure that the public Key encoding is known
48       if(X509AddPublicKey(NULL, object) == 0)
49           return TPM_RCS_ASYMMETRIC + RC_CertifyX509_objectHandle;
50       // Unbundle 'partialCertificate'.
51           // Initialize the unmarshaling context
52       if(!ASN1UnmarshalContextInitialize(&ctx, in->partialCertificate.t.size,
53           in->partialCertificate.t.buffer))
54           return TPM_RCS_VALUE + RC_CertifyX509_partialCertificate;
55       // Make sure that this is a constructed SEQUENCE
56       length = ASN1NextTag(&ctx);
57       // Must be a constructed SEQUENCE that uses all of the input parameter
58       if((ctx.tag != (ASN1_CONSTRUCTED_SEQUENCE))
59           || ((ctx.offset + length) != in->partialCertificate.t.size))
60           return TPM_RCS_SIZE + RC_CertifyX509_partialCertificate;
61
62       // This scans through the contents of the outermost SEQUENCE. This would be the
63       // 'issuer', 'validity', 'subject', 'issuerUniqueID' (optional),
64       // 'subjectUniqueID' (optional), and 'extensions.'
65       while(ctx.offset < ctx.size)
66       {
67           INT16           startOfElement = ctx.offset;
68           //
69               // Read the next tag and length field.
70           length = ASN1NextTag(&ctx);
71           if(length < 0)
72               break;
73           if(ctx.tag == ASN1_CONSTRUCTED_SEQUENCE)
74           {
75               partial[countOfSequences].buf = &ctx.buffer[startOfElement];
76               ctx.offset += length;
77               partial[countOfSequences].len = (INT16)ctx.offset - startOfElement;
78               if(++countOfSequences > ALLOWED_SEQUENCES)
79                   break;
80           }
81           else if(ctx.tag  == X509_EXTENSIONS)
82           {
83               if(certTBS[EXTENSIONS_REF].len != 0)
84                   return TPM_RCS_VALUE + RC_CertifyX509_partialCertificate;
85               certTBS[EXTENSIONS_REF].buf = &ctx.buffer[startOfElement];
86               ctx.offset += length;
87               certTBS[EXTENSIONS_REF].len =
88                   (INT16)ctx.offset - startOfElement;
89           }
90           else
91               return TPM_RCS_VALUE + RC_CertifyX509_partialCertificate;
92       }
93       // Make sure that we used all of the data and found at least the required
94       // number of elements.
95       if((ctx.offset != ctx.size) || (countOfSequences < 3)
96           || (countOfSequences > 4)
97           || (certTBS[EXTENSIONS_REF].buf == NULL))
98           return TPM_RCS_VALUE + RC_CertifyX509_partialCertificate;
99       // Now that we know how many sequences there were, we can put them where they
100      // belong
101      for(i = 0; i < countOfSequences; i++)
102          certTBS[SUBJECT_KEY_REF - i] = partial[countOfSequences - 1 - i];
103
104      // If only three SEQUENCES, then the TPM needs to produce the signature algorithm.
105      // See if it can
106      if((countOfSequences == 3) &&
107          (X509AddSigningAlgorithm(NULL, signKey, &in->inScheme) == 0))
108              return TPM_RCS_SCHEME + RC_CertifyX509_signHandle;
109
110      // Process the extensions
```

```
111         result = X509ProcessExtensions(object, &certTBS[EXTENSIONS_REF]);
112         if(result != TPM_RC_SUCCESS)
113             // If the extension has the TPMA_OBJECT extension and the attributes don't
114             // match, then the error code will be TPM_RCS_ATTRIBUTES. Otherwise, the error
115             // indicates a malformed partialCertificate.
116             return result + ((result == TPM_RCS_ATTRIBUTES)
117                              ? RC_CertifyX509_objectHandle
118                              : RC_CertifyX509_partialCertificate);
119 // Command Output
120 // Create the addedToCertificate values
121
122     // Build the addedToCertificate from the bottom up.
123     // Initialize the context structure
124     ASN1InitializeMarshalContext(&ctxOut, sizeof(out->addedToCertificate.t.buffer),
125                                  out->addedToCertificate.t.buffer);
126     // Place a marker for the overall context
127     ASN1StartMarshalContext(&ctxOut);  // SEQUENCE for addedToCertificate
128
129     // Add the subject public key descriptor
130     certTBS[SUBJECT_PUBLIC_KEY_REF].len = X509AddPublicKey(&ctxOut, object);
131     certTBS[SUBJECT_PUBLIC_KEY_REF].buf = ctxOut.buffer + ctxOut.offset;
132     // If the caller didn't provide the algorithm identifier, create it
133     if(certTBS[SIGNATURE_REF].len == 0)
134     {
135         certTBS[SIGNATURE_REF].len = X509AddSigningAlgorithm(&ctxOut, signKey,
136             &in->inScheme);
137         certTBS[SIGNATURE_REF].buf = ctxOut.buffer + ctxOut.offset;
138     }
139     // Create the serial number value. Use the out->tbsDigest as scratch.
140     {
141         TPM2B                   *digest = &out->tbsDigest.b;
142         //
143         digest->size = (INT16)CryptHashStart(&hash, signKey->publicArea.nameAlg);
144         pAssert(digest->size != 0);
145
146         // The serial number size is the smaller of the digest and the vendor-defined
147         // value
148         digest->size = MIN(digest->size, SIZE_OF_X509_SERIAL_NUMBER);
149         // Add all the parts of the certificate other than the serial number
150         // and version number
151         for(i = SIGNATURE_REF; i < REF_COUNT; i++)
152             CryptDigestUpdate(&hash, certTBS[i].len, certTBS[i].buf);
153         // throw in the Name of the signing key...
154         CryptDigestUpdate2B(&hash, &signKey->name.b);
155         // ...and the Name of the signed key.
156         CryptDigestUpdate2B(&hash, &object->name.b);
157         // Done
158         CryptHashEnd2B(&hash, digest);
159     }
160
161     // Add the serial number
162     certTBS[SERIAL_NUMBER_REF].len =
163         ASN1PushInteger(&ctxOut, out->tbsDigest.t.size, out->tbsDigest.t.buffer);
164     certTBS[SERIAL_NUMBER_REF].buf = ctxOut.buffer + ctxOut.offset;
165
166     // Add the static version number
167     ASN1StartMarshalContext(&ctxOut);
168     ASN1PushUINT(&ctxOut, 2);
169     certTBS[VERSION_REF].len =
170         ASN1EndEncapsulation(&ctxOut, ASN1_APPLICAIION_SPECIFIC);
171     certTBS[VERSION_REF].buf = ctxOut.buffer + ctxOut.offset;
172
173     // Create a fake tag and length for the TBS in the space used for
174     // 'addedToCertificate'
175     {
176         for(length = 0, i = 0; i < REF_COUNT; i++)
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

```
177                 length += certTBS[i].len;
178             // Put a fake tag and length into the buffer for use in the tbsDigest
179             certTBS[ENCODED_SIZE_REF].len =
180                 ASN1PushTagAndLength(&ctxOut, ASN1_CONSTRUCTED_SEQUENCE, length);
181             certTBS[ENCODED_SIZE_REF].buf = ctxOut.buffer + ctxOut.offset;
182             // Restore the buffer pointer to add back the number of octets used for the
183             // tag and length
184             ctxOut.offset += certTBS[ENCODED_SIZE_REF].len;
185         }
186         // sanity check
187         if(ctxOut.offset < 0)
188             return TPM_RC_FAILURE;
189         // Create the tbsDigest to sign
190         out->tbsDigest.t.size = CryptHashStart(&hash, in->inScheme.details.any.hashAlg);
191         for(i = 0; i < REF_COUNT; i++)
192             CryptDigestUpdate(&hash, certTBS[i].len, certTBS[i].buf);
193         CryptHashEnd2B(&hash, &out->tbsDigest.b);
194
195     #if CERTIFYX509_DEBUG
196         {
197             BYTE                 fullTBS[4096];
198             BYTE                *fill = fullTBS;
199             int                  j;
200             for (j = 0; j < REF_COUNT; j++)
201             {
202                 MemoryCopy(fill, certTBS[j].buf, certTBS[j].len);
203                 fill += certTBS[j].len;
204             }
205             DebugDumpBuffer((int)(fill - &fullTBS[0]), fullTBS, "\nfull TBS");
206         }
207     #endif
208
209     // Finish up the processing of addedToCertificate
210         // Create the actual tag and length for the addedToCertificate structure
211         out->addedToCertificate.t.size =
212             ASN1EndEncapsulation(&ctxOut, ASN1_CONSTRUCTED_SEQUENCE);
213         // Now move all the addedToContext to the start of the buffer
214         MemoryCopy(out->addedToCertificate.t.buffer, ctxOut.buffer + ctxOut.offset,
215                    out->addedToCertificate.t.size);
216     #if CERTIFYX509_DEBUG
217         DebugDumpBuffer(out->addedToCertificate.t.size, out->addedToCertificate.t.buffer,
218                         "\naddedToCertificate");
219     #endif
220         // only thing missing is the signature
221         result = CryptSign(signKey, &in->inScheme, &out->tbsDigest, &out->signature);
222
223         return result;
224     }
225     #endif // CC_CertifyX509
```

19 Ephemeral EC Keys

19.1 Introduction

The TPM generates keys that have different lifetimes. TPM keys in a hierarchy can be persistent for as long as the seed of the hierarchy is unchanged and these keys may be used multiple times. Other TPM-generated keys are only useful for a single operation. Some of these single-use keys are used in the command in which they are created. Examples of this use are TPM2_Duplicate() where an ephemeral key is created for a single pass key exchange with another TPM. However, there are other cases, such as anonymous attestation, where the protocol requires two passes where the public part of the ephemeral key is used outside of the TPM before the final command "consumes" the ephemeral key.

For these uses, TPM2_Commit() or TPM2_EC_Ephemeral() may be used to have the TPM create an ephemeral EC key and return the public part of the key for external use. Then in a subsequent command, the caller provides a reference to the ephemeral key so that the TPM can retrieve or recreate the associated private key.

When an ephemeral EC key is created, it is assigned a number and that number is returned to the caller as the identifier for the key. This number is not a handle. A handle is assigned to a key that may be context saved but these ephemeral EC keys may not be saved and do not have a full key context. When a subsequent command uses the ephemeral key, the caller provides the number of the ephemeral key. The TPM uses that number to either look up or recompute the associated private key. After the key is used, the TPM records the fact that the key has been used so that it cannot be used again.

As mentioned, the TPM can keep each assigned private ephemeral key in memory until it is used. However, this could consume a large amount of memory. To limit the memory size, the TPM is allowed to restrict the number of pending private keys – keys that have been allocated but not used.

NOTE The minimum number of ephemeral keys is determined by a platform specific specification

To further reduce the memory requirements for the ephemeral private keys, the TPM is allowed to use pseudo-random values for the ephemeral keys. Instead of keeping the full value of the key in memory, the TPM can use a counter as input to a KDF. Incrementing the counter will cause the TPM to generate a new pseudo-random value.

Using the counter to generate pseudo-random private ephemeral keys greatly simplifies tracking of key usage. When a counter value is used to create a key, a bit in an array may be set to indicate that the key use is pending. When the ephemeral key is consumed, the bit is cleared. This prevents the key from being used more than once.

Since the TPM is allowed to restrict the number of pending ephemeral keys, the array size can be limited. For example, a 128 bit array would allow 128 keys to be "pending".

The management of the array is described in greater detail in the *Split Operations* clause in Annex C of TPM 2.0 Part 1.

 Copyright © TCG 2006-2020

19.2 TPM2_Commit

19.2.1 General Description

TPM2_Commit() performs the first part of an ECC anonymous signing operation. The TPM will perform the point multiplications on the provided points and return intermediate signing values. The *signHandle* parameter shall refer to an ECC key and the signing scheme must be anonymous (TPM_RC_SCHEME).

NOTE 1 Currently, TPM_ALG_ECDAA is the only defined anonymous scheme.

NOTE 2 This command cannot be used with a sign+decrypt key because that type of key is required to have a scheme of TPM_ALG_NULL.

For this command, *p1*, *s2* and *y2* are optional parameters. If *s2* is an Empty Buffer, then the TPM shall return TPM_RC_SIZE if *y2* is not an Empty Buffer.

The algorithm is specified in the TPM 2.0 Part 1 Annex for ECC, TPM2_Commit().

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

19.2.2 Command and Response

Table 100 — TPM2_Commit Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Commit
TPMI_DH_OBJECT	@signHandle	handle of the key that will be used in the signing operation Auth Index: 1 Auth Role: USER
TPM2B_ECC_POINT	P1	a point (M) on the curve used by *signHandle*
TPM2B_SENSITIVE_DATA	s2	octet array used to derive x-coordinate of a base point
TPM2B_ECC_PARAMETER	y2	y coordinate of the point associated with *s2*

Table 101 — TPM2_Commit Response

Type	Name	Description
TPM_ST	tag	see 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ECC_POINT	K	ECC point $K \coloneqq [d_s](x2, y2)$
TPM2B_ECC_POINT	L	ECC point $L \coloneqq [r](x2, y2)$
TPM2B_ECC_POINT	E	ECC point $E \coloneqq [r]P1$
UINT16	counter	least-significant 16 bits of *commitCount*

 Copyright © TCG 2006-2020

19.2.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "Commit_fp.h"
3  #if CC_Commit  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*keyHandle* references a restricted key that is not a signing key
TPM_RC_ECC_POINT	either *P1* or the point derived from *s2* is not on the curve of *keyHandle*
TPM_RC_HASH	invalid name algorithm in *keyHandle*
TPM_RC_KEY	*keyHandle* does not reference an ECC key
TPM_RC_SCHEME	the scheme of *keyHandle* is not an anonymous scheme
TPM_RC_NO_RESULT	*K*, *L* or *E* was a point at infinity; or failed to generate r value
TPM_RC_SIZE	*s2* is empty but *y2* is not or *s2* provided but *y2* is not

```
4  TPM_RC
5  TPM2_Commit(
6      Commit_In       *in,            // IN: input parameter list
7      Commit_Out      *out            // OUT: output parameter list
8      )
9  {
10     OBJECT                     *eccKey;
11     TPMS_ECC_POINT              P2;
12     TPMS_ECC_POINT             *pP2 = NULL;
13     TPMS_ECC_POINT             *pP1 = NULL;
14     TPM2B_ECC_PARAMETER         r;
15     TPM2B_ECC_PARAMETER         p;
16     TPM_RC                      result;
17     TPMS_ECC_PARMS             *parms;
18
19 // Input Validation
20
21     eccKey = HandleToObject(in->signHandle);
22     parms = &eccKey->publicArea.parameters.eccDetail;
23
24     // Input key must be an ECC key
25     if(eccKey->publicArea.type != TPM_ALG_ECC)
26         return TPM_RCS_KEY + RC_Commit_signHandle;
27
28     // This command may only be used with a sign-only key using an anonymous
29     // scheme.
30     // NOTE: a sign + decrypt key has no scheme so it will not be an anonymous one
31     // and an unrestricted sign key might no have a signing scheme but it can't
32     // be use in Commit()
33     if(!CryptIsSchemeAnonymous(parms->scheme.scheme))
34         return TPM_RCS_SCHEME + RC_Commit_signHandle;
35
36 // Make sure that both parts of P2 are present if either is present
37     if((in->s2.t.size == 0) != (in->y2.t.size == 0))
38         return TPM_RCS_SIZE + RC_Commit_y2;
39
40     // Get prime modulus for the curve. This is needed later but getting this now
41     // allows confirmation that the curve exists.
42     if(!CryptEccGetParameter(&p, 'p', parms->curveID))
43         return TPM_RCS_KEY + RC_Commit_signHandle;
44
45     // Get the random value that will be used in the point multiplications
```

```
46      // Note: this does not commit the count.
47      if(!CryptGenerateR(&r, NULL, parms->curveID, &eccKey->name))
48          return TPM_RC_NO_RESULT;
49
50      // Set up P2 if s2 and Y2 are provided
51      if(in->s2.t.size != 0)
52      {
53          TPM2B_DIGEST            x2;
54
55          pP2 = &P2;
56
57          // copy y2 for P2
58          P2.y = in->y2;
59
60          // Compute x2  HnameAlg(s2) mod p
61          //      do the hash operation on s2 with the size of curve 'p'
62          x2.t.size = CryptHashBlock(eccKey->publicArea.nameAlg,
63                                     in->s2.t.size,
64                                     in->s2.t.buffer,
65                                     sizeof(x2.t.buffer),
66                                     x2.t.buffer);
67
68          // If there were error returns in the hash routine, indicate a problem
69          // with the hash algorithm selection
70          if(x2.t.size == 0)
71              return TPM_RCS_HASH + RC_Commit_signHandle;
72          // The size of the remainder will be same as the size of p. DivideB() will
73          // pad the results (leading zeros) if necessary to make the size the same
74          P2.x.t.size = p.t.size;
75          //  set p2.x = hash(s2) mod p
76          if(DivideB(&x2.b, &p.b, NULL, &P2.x.b) != TPM_RC_SUCCESS)
77              return TPM_RC_NO_RESULT;
78
79          if(!CryptEccIsPointOnCurve(parms->curveID, pP2))
80              return TPM_RCS_ECC_POINT + RC_Commit_s2;
81
82          if(eccKey->attributes.publicOnly == SET)
83              return TPM_RCS_KEY + RC_Commit_signHandle;
84      }
85      // If there is a P1, make sure that it is on the curve
86      // NOTE: an "empty" point has two UINT16 values which are the size values
87      // for each of the coordinates.
88      if(in->P1.size > 4)
89      {
90          pP1 = &in->P1.point;
91          if(!CryptEccIsPointOnCurve(parms->curveID, pP1))
92              return TPM_RCS_ECC_POINT + RC_Commit_P1;
93      }
94
95      // Pass the parameters to CryptCommit.
96      // The work is not done in-line because it does several point multiplies
97      // with the same curve.  It saves work by not having to reload the curve
98      // parameters multiple times.
99      result = CryptEccCommitCompute(&out->K.point,
100                                    &out->L.point,
101                                    &out->E.point,
102                                    parms->curveID,
103                                    pP1,
104                                    pP2,
105                                    &eccKey->sensitive.sensitive.ecc,
106                                    &r);
107     if(result != TPM_RC_SUCCESS)
108         return result;
109
110     // The commit computation was successful so complete the commit by setting
111     // the bit
```

 Copyright © TCG 2006-2020

```
112     out->counter = CryptCommit();
113
114     return TPM_RC_SUCCESS;
115 }
116 #endif // CC_Commit
```

19.3 TPM2_EC_Ephemeral

19.3.1 General Description

TPM2_EC_Ephemeral() creates an ephemeral key for use in a two-phase key exchange protocol.

The TPM will use the commit mechanism to assign an ephemeral key r and compute a public point $Q \coloneqq [r]G$ where G is the generator point associated with *curveID*.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

19.3.2 Command and Response

Table 102 — TPM2_EC_Ephemeral Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_EC_Ephemeral
TPMI_ECC_CURVE	curveID	The curve for the computed ephemeral point

Table 103 — TPM2_EC_Ephemeral Response

Type	Name	Description
TPM_ST	tag	see 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ECC_POINT	Q	ephemeral public key $Q \coloneqq [r]G$
UINT16	counter	least-significant 16 bits of *commitCount*

 Copyright © TCG 2006-2020

19.3.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "EC_Ephemeral_fp.h"
3  #if CC_EC_Ephemeral  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_NO_RESULT	the TPM is not able to generate an *r* value

```
4  TPM_RC
5  TPM2_EC_Ephemeral(
6      EC_Ephemeral_In     *in,            // IN: input parameter list
7      EC_Ephemeral_Out    *out            // OUT: output parameter list
8      )
9  {
10     TPM2B_ECC_PARAMETER      r;
11     TPM_RC                   result;
12 //
13     do
14     {
15         // Get the random value that will be used in the point multiplications
16         // Note: this does not commit the count.
17         if(!CryptGenerateR(&r, NULL, in->curveID, NULL))
18             return TPM_RC_NO_RESULT;
19         // do a point multiply
20         result = CryptEccPointMultiply(&out->Q.point, in->curveID, NULL, &r,
21                                        NULL, NULL);
22         // commit the count value if either the r value results in the point at
23         // infinity or if the value is good. The commit on the r value for infinity
24         // is so that the r value will be skipped.
25         if((result == TPM_RC_SUCCESS) || (result == TPM_RC_NO_RESULT))
26             out->counter = CryptCommit();
27     } while(result == TPM_RC_NO_RESULT);
28
29     return TPM_RC_SUCCESS;
30 }
31 #endif // CC_EC_Ephemeral
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

20 Signing and Signature Verification

20.1 TPM2_VerifySignature

20.1.1 General Description

This command uses loaded keys to validate a signature on a message with the message digest passed to the TPM.

If the signature check succeeds, then the TPM will produce a TPMT_TK_VERIFIED. Otherwise, the TPM shall return TPM_RC_SIGNATURE.

If the key is in the NULL hierarchy, then *digest* in the ticket will be the Empty Buffer.

NOTE 1 A valid ticket may be used in subsequent commands to provide proof to the TPM that the TPM has validated the signature over the message using the key referenced by *keyHandle*.

If *keyHandle* references an asymmetric key, only the public portion of the key needs to be loaded. If *keyHandle* references a symmetric key, both the public and private portions need to be loaded.

NOTE 2 The sensitive area of the symmetric object is required to allow verification of the symmetric signature (the HMAC).

 Copyright © TCG 2006-2020

20.1.2 Command and Response

Table 104 — TPM2_VerifySignature Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_VerifySignature
TPMI_DH_OBJECT	keyHandle	handle of public key that will be used in the validation Auth Index: None
TPM2B_DIGEST	digest	digest of the signed message
TPMT_SIGNATURE	signature	signature to be tested

Table 105 — TPM2_VerifySignature Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMT_TK_VERIFIED	validation	

 Copyright © TCG 2006-2020

20.1.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "VerifySignature_fp.h"
3  #if CC_VerifySignature  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*keyHandle* does not reference a signing key
TPM_RC_SIGNATURE	signature is not genuine
TPM_RC_SCHEME	CryptValidateSignature()
TPM_RC_HANDLE	the input handle is references an HMAC key but the private portion is not loaded

```
4  TPM_RC
5  TPM2_VerifySignature(
6      VerifySignature_In      *in,            // IN: input parameter list
7      VerifySignature_Out     *out            // OUT: output parameter list
8      )
9  {
10     TPM_RC                   result;
11     OBJECT                  *signObject = HandleToObject(in->keyHandle);
12     TPMI_RH_HIERARCHY        hierarchy;
13
14 // Input Validation
15     // The object to validate the signature must be a signing key.
16     if(!IS_ATTRIBUTE(signObject->publicArea.objectAttributes, TPMA_OBJECT, sign))
17         return TPM_RCS_ATTRIBUTES + RC_VerifySignature_keyHandle;
18
19     // Validate Signature.  TPM_RC_SCHEME, TPM_RC_HANDLE or TPM_RC_SIGNATURE
20     // error may be returned by CryptCVerifySignatrue()
21     result = CryptValidateSignature(in->keyHandle, &in->digest, &in->signature);
22     if(result != TPM_RC_SUCCESS)
23         return RcSafeAddToResult(result, RC_VerifySignature_signature);
24
25 // Command Output
26
27     hierarchy = GetHeriarchy(in->keyHandle);
28     if(hierarchy == TPM_RH_NULL
29        || signObject->publicArea.nameAlg == TPM_ALG_NULL)
30     {
31         // produce empty ticket if hierarchy is TPM_RH_NULL or nameAlg is
32         // TPM_ALG_NULL
33         out->validation.tag = TPM_ST_VERIFIED;
34         out->validation.hierarchy = TPM_RH_NULL;
35         out->validation.digest.t.size = 0;
36     }
37     else
38     {
39         // Compute ticket
40         TicketComputeVerified(hierarchy, &in->digest, &signObject->name,
41                               &out->validation);
42     }
43
44     return TPM_RC_SUCCESS;
45 }
46 #endif // CC_VerifySignature
```

20.2 TPM2_Sign

20.2.1 General Description

This command causes the TPM to sign an externally provided hash with the specified symmetric or asymmetric signing key.

NOTE 1 If *keyhandle* references an unrestricted signing key, a digest can be signed using either this command or an HMAC command.

If *keyHandle* references a restricted signing key, then *validation* shall be provided, indicating that the TPM performed the hash of the data and *validation* shall indicate that hashed data did not start with TPM_GENERATED_VALUE.

NOTE 2 If the hashed data did start with TPM_GENERATED_VALUE, then the validation will be a NULL ticket.

The *x509sign* attribute of keyHandle may not be SET (TPM_RC_ATTRIBUTES).

If the scheme of *keyHandle* is not TPM_ALG_NULL, then *inScheme* shall either be the same scheme as *keyHandle* or TPM_ALG_NULL. If the *sign* attribute is not SET in the key referenced by *handle* then the TPM shall return TPM_RC_KEY.

If the scheme of *keyHandle* is TPM_ALG_NULL, the TPM will sign using *inScheme*; otherwise, it will sign using the scheme of *keyHandle*.

NOTE 3 When the signing scheme uses a hash algorithm, the algorithm is defined in the qualifying data of the scheme. This is the same algorithm that is required to be used in producing *digest*. The size of *digest* must match that of the hash algorithm in the scheme.

If *inScheme* is not a valid signing scheme for the type of keyHandle (or TPM_ALG_NULL), then the TPM shall return TPM_RC_SCHEME.

If the scheme of *keyHandle* is an anonymous *scheme*, then *inScheme* shall have the same scheme algorithm as *keyHandle* and *inScheme* will contain a counter value that will be used in the signing process.

EXAMPLE For ECDAA, *inScheme.details.ecdaa.count* will contain the count value.

If *validation* is provided, then the hash algorithm used in computing the digest is required to be the hash algorithm specified in the scheme of *keyHandle* (TPM_RC_TICKET).

If the *validation* parameter is not the Empty Buffer*,* then it will be checked even if the key referenced by *keyHandle* is not a restricted signing key.

NOTE 4 If *keyHandle* is both a sign and decrypt key, *keyHandle* will have a scheme of TPM_ALG_NULL. If *validation* is provided, then it must be a NULL validation ticket or the ticket validation will fail.

 Copyright © TCG 2006-2020

20.2.2 Command and Response

Table 106 — TPM2_Sign Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Sign
TPMI_DH_OBJECT	@keyHandle	Handle of key that will perform signing Auth Index: 1 Auth Role: USER
TPM2B_DIGEST	digest	digest to be signed
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *keyHandle* is TPM_ALG_NULL
TPMT_TK_HASHCHECK	validation	proof that digest was created by the TPM If *keyHandle* is not a restricted signing key, then this may be a NULL Ticket with *tag* = TPM_ST_CHECKHASH.

Table 107 — TPM2_Sign Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMT_SIGNATURE	signature	the signature

 Copyright © TCG 2006-2020

20.2.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "Sign_fp.h"
3  #if CC_Sign  // Conditional expansion of this file
4  #include "Attest_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_BINDING	The public and private portions of the key are not properly bound.
TPM_RC_KEY	*signHandle* does not reference a signing key;
TPM_RC_SCHEME	the scheme is not compatible with sign key type, or input scheme is not compatible with default scheme, or the chosen scheme is not a valid sign scheme
TPM_RC_TICKET	*validation* is not a valid ticket
TPM_RC_VALUE	the value to sign is larger than allowed for the type of *keyHandle*

```
5  TPM_RC
6  TPM2_Sign(
7      Sign_In          *in,            // IN: input parameter list
8      Sign_Out         *out            // OUT: output parameter list
9      )
10 {
11     TPM_RC                    result;
12     TPMT_TK_HASHCHECK         ticket;
13     OBJECT                   *signObject = HandleToObject(in->keyHandle);
14 //
15 // Input Validation
16     if(!IsSigningObject(signObject))
17         return TPM_RCS_KEY + RC_Sign_keyHandle;
18
19     // A key that will be used for x.509 signatures can't be used in TPM2_Sign().
20     if(IS_ATTRIBUTE(signObject->publicArea.objectAttributes, TPMA_OBJECT, x509sign))
21         return TPM_RCS_ATTRIBUTES + RC_Sign_keyHandle;
22
23     // pick a scheme for sign.  If the input sign scheme is not compatible with
24     // the default scheme, return an error.
25     if(!CryptSelectSignScheme(signObject, &in->inScheme))
26         return TPM_RCS_SCHEME + RC_Sign_inScheme;
27
28     // If validation is provided, or the key is restricted, check the ticket
29     if(in->validation.digest.t.size != 0
30        || IS_ATTRIBUTE(signObject->publicArea.objectAttributes,
31                        TPMA_OBJECT, restricted))
32     {
33         // Compute and compare ticket
34         TicketComputeHashCheck(in->validation.hierarchy,
35                                in->inScheme.details.any.hashAlg,
36                                &in->digest, &ticket);
37
38         if(!MemoryEqual2B(&in->validation.digest.b, &ticket.digest.b))
39             return TPM_RCS_TICKET + RC_Sign_validation;
40     }
41     else
42     // If we don't have a ticket, at least verify that the provided 'digest'
43     // is the size of the scheme hashAlg digest.
44     // NOTE: this does not guarantee that the 'digest' is actually produced using
45     // the indicated hash algorithm, but at least it might be.
46     {
47         if(in->digest.t.size
```


 Copyright © TCG 2006-2020

```
48             != CryptHashGetDigestSize(in->inScheme.details.any.hashAlg))
49              return TPM_RCS_SIZE + RC_Sign_digest;
50     }
51
52 // Command Output
53     // Sign the hash. A TPM_RC_VALUE or TPM_RC_SCHEME
54     // error may be returned at this point
55     result = CryptSign(signObject, &in->inScheme, &in->digest, &out->signature);
56
57     return result;
58 }
59 #endif // CC_Sign
```


 Copyright © TCG 2006-2020

21 Command Audit

21.1 Introduction

If a command has been selected for command audit, the command audit status will be updated when that command completes successfully. The digest is updated as:

$$commandAuditDigest_{new} \coloneqq \mathbf{H}_{auditAlg}(commandAuditDigest_{old} \mathbin{||} cpHash \mathbin{||} rpHash) \tag{5}$$

where

$\boldsymbol{H}_{auditAlg}$	hash function using the algorithm of the audit sequence
$commandAuditDigest$	accumulated digest
$cpHash$	the command parameter hash
$rpHash$	the response parameter hash

auditAlg, the hash algorithm, is set using TPM2_SetCommandCodeAuditStatus().

TPM2_Shutdown() cannot be audited but TPM2_Startup() can be audited. If the *cpHash* of the TPM2_Startup() is TPM_SU_STATE, that would indicate that a TPM2_Shutdown() had been successfully executed.

TPM2_SetCommandCodeAuditStatus() is always audited, except when it is used to change *auditAlg*.

If the TPM is in Failure mode, command audit is not functional.

 Copyright © TCG 2006-2020

21.2 TPM2_SetCommandCodeAuditStatus

21.2.1 General Description

This command may be used by the Privacy Administrator or platform to change the audit status of a command or to set the hash algorithm used for the audit digest, but not both at the same time.

If the *auditAlg* parameter is a supported hash algorithm and not the same as the current algorithm, then the TPM will check both *setList* and *clearList* are empty (zero length). If so, then the algorithm is changed, and the audit digest is cleared. If *auditAlg* is TPM_ALG_NULL or the same as the current algorithm, then the algorithm and audit digest are unchanged and the *setList* and *clearList* will be processed.

NOTE 1 Because the audit digest is cleared, the audit counter will increment the next time that an audited command is executed.

Use of TPM2_SetCommandCodeAuditStatus() to change the list of audited commands is an audited event. If TPM_CC_SetCommandCodeAuditStatus is in *clearList,* the fact that it is in *clearList* is ignored.

NOTE 2 Use of this command to change the audit hash algorithm is not audited and the digest is reset when the command completes. The change in the audit hash algorithm is the evidence that this command was used to change the algorithm.

The commands in *setList* indicate the commands to be added to the list of audited commands and the commands in *clearList* indicate the commands that will no longer be audited. It is not an error if a command in *setList* is already audited or is not implemented. It is not an error if a command in *clearList* is not currently being audited or is not implemented.

If a command code is in both *setList* and *clearList*, then it will not be audited (that is, *setList* shall be processed first).

 Copyright © TCG 2006-2020

21.2.2 Command and Response

Table 108 — TPM2_SetCommandCodeAuditStatus Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_SetCommandCodeAuditStatus {NV}
TPMI_RH_PROVISION	@auth	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPMI_ALG_HASH+	auditAlg	hash algorithm for the audit digest; if TPM_ALG_NULL, then the hash is not changed
TPML_CC	setList	list of commands that will be added to those that will be audited
TPML_CC	clearList	list of commands that will no longer be audited

Table 109 — TPM2_SetCommandCodeAuditStatus Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

21.2.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "SetCommandCodeAuditStatus_fp.h"
3  #if CC_SetCommandCodeAuditStatus  // Conditional expansion of this file
4  TPM_RC
5  TPM2_SetCommandCodeAuditStatus(
6      SetCommandCodeAuditStatus_In    *in             // IN: input parameter list
7      )
8  {
9
10     // The command needs NV update.  Check if NV is available.
11     // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
12     // this point
13     RETURN_IF_NV_IS_NOT_AVAILABLE;
14
15 // Internal Data Update
16
17     // Update hash algorithm
18     if(in->auditAlg != TPM_ALG_NULL && in->auditAlg != gp.auditHashAlg)
19     {
20         // Can't change the algorithm and command list at the same time
21         if(in->setList.count != 0 || in->clearList.count != 0)
22             return TPM_RCS_VALUE + RC_SetCommandCodeAuditStatus_auditAlg;
23
24         // Change the hash algorithm for audit
25         gp.auditHashAlg = in->auditAlg;
26
27         // Set the digest size to a unique value that indicates that the digest
28         // algorithm has been changed. The size will be cleared to zero in the
29         // command audit processing on exit.
30         gr.commandAuditDigest.t.size = 1;
31
32         // Save the change of command audit data (this sets g_updateNV so that NV
33         // will be updated on exit.)
34         NV_SYNC_PERSISTENT(auditHashAlg);
35     }
36     else
37     {
38         UINT32          i;
39         BOOL            changed = FALSE;
40
41         // Process set list
42         for(i = 0; i < in->setList.count; i++)
43
44             // If change is made in CommandAuditSet, set changed flag
45             if(CommandAuditSet(in->setList.commandCodes[i]))
46                 changed = TRUE;
47
48         // Process clear list
49         for(i = 0; i < in->clearList.count; i++)
50             // If change is made in CommandAuditClear, set changed flag
51             if(CommandAuditClear(in->clearList.commandCodes[i]))
52                 changed = TRUE;
53
54         // if change was made to command list, update NV
55         if(changed)
56             // this sets g_updateNV so that NV will be updated on exit.
57             NV_SYNC_PERSISTENT(auditCommands);
58     }
59
60     return TPM_RC_SUCCESS;
61 }
62 #endif // CC_SetCommandCodeAuditStatus
```

22 Integrity Collection (PCR)

22.1 Introduction

In TPM 1.2, an Event was hashed using SHA-1 and then the 20-octet digest was extended to a PCR using TPM_Extend(). This specification allows the use of multiple PCR at a given Index, each using a different hash algorithm. Rather than require that the external software generate multiple hashes of the Event with each being extended to a different PCR, the Event data may be sent to the TPM for hashing. This ensures that the resulting digests will properly reflect the algorithms chosen for the PCR even if the calling software is unable to implement the hash algorithm.

NOTE 1 There is continued support for software hashing of events with TPM2_PCR_Extend().

To support recording of an Event that is larger than the TPM input buffer, the caller may use the command sequence described in clause 1.

Change to a PCR requires authorization. The authorization may be with either an authorization value or an authorization policy. The platform-specific specifications determine which PCR may be controlled by policy. All other PCR are controlled by authorization.

If a PCR may be associated with a policy, then the algorithm ID of that policy determines whether the policy is to be applied. If the algorithm ID is not TPM_ALG_NULL, then the policy digest associated with the PCR must match the *policySession→policyDigest* in a policy session. If the algorithm ID is TPM_ALG_NULL, then no policy is present and the authorization requires an EmptyAuth.

If a platform-specific specification indicates that PCR are grouped, then all the PCR in the group use the same authorization policy or authorization value.

pcrUpdateCounter counter will be incremented on the successful completion of any command that modifies (Extends or resets) a PCR unless the platform-specific specification explicitly excludes the PCR from being counted.

NOTE 2 If a command causes PCR in multiple banks to change, the PCR Update Counter must be incremented once for each bank. The commands that extend PCR are: TPM2_PCR_Extend, TPM2_PCR_Event, and TPM2_EventSequenceComplete.

If a command resets PCR in multiple banks, the PCR Update Counter must be incremented only once. The commands that reset PCR are: TPM2_PCR_Reset, and TPM2_Startup.

A platform-specific specification may designate a set of PCR that are under control of the TCB. These PCR may not be modified without the proper authorization. Updates of these PCR shall not cause the PCR Update Counter to increment.

EXAMPLE Updates of the TCB PCR will not cause the PCR update counter to increment because these PCR are changed at the whim of the TCB and may not represent the trust state of the platform.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

22.2 TPM2_PCR_Extend

22.2.1 General Description

This command is used to cause an update to the indicated PCR. The *digests* parameter contains one or more tagged digest values identified by an algorithm ID. For each digest, the PCR associated with *pcrHandle* is Extended into the bank identified by the tag (*hashAlg*).

EXAMPLE A SHA1 digest would be Extended into the SHA1 bank and a SHA256 digest would be Extended into the SHA256 bank.

For each list entry, the TPM will check to see if *pcrNum* is implemented for that algorithm. If so, the TPM shall perform the following operation:

$$PCR.digest_{new}[pcrNum][alg] \coloneqq \mathbf{H}_{alg}(PCR.digest_{old}[pcrNum][alg] \mid\mid data[alg].buffer) \quad (6)$$

where

$\mathbf{H}_{alg}()$	hash function using the hash algorithm associated with the PCR instance
$PCR.digest$	the digest value in a PCR
$pcrNum$	the PCR numeric selector (*pcrHandle*)
alg	the PCR algorithm selector for the digest
$data[alg].buffer$	the bank-specific data to be extended

If no digest value is specified for a bank, then the PCR in that bank is not modified.

NOTE 1 This allows consistent operation of the digests list for all of the Event recording commands.

If a digest is present and the PCR in that bank is not implemented, the digest value is not used.

NOTE 2 If the caller includes digests for algorithms that are not implemented, then the TPM will fail the call because the unmarshalling of *digests* will fail. Each of the entries in the list is a TPMT_HA, which is a hash algorithm followed by a digest. If the algorithm is not implemented, unmarshalling of the *hashAlg* will fail and the TPM will return TPM_RC_HASH.

If the TPM unmarshals the *hashAlg* of a list entry and the unmarshaled value is not a hash algorithm implemented on the TPM, the TPM shall return TPM_RC_HASH.

The *pcrHandle* parameter is allowed to reference TPM_RH_NULL. If so, the input parameters are processed but no action is taken by the TPM. This permits the caller to probe for implemented hash algorithms as an alternative to TPM2_GetCapability.

NOTE 3 This command allows a list of digests so that PCR in all banks may be updated in a single command. While the semantics of this command allow multiple extends to a single PCR bank, this is not the preferred use and the limit on the number of entries in the list make this use somewhat impractical.

 Copyright © TCG 2006-2020

22.2.2 Command and Response

Table 110 — TPM2_PCR_Extend Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_Extend {NV}
TPMI_DH_PCR+	@pcrHandle	handle of the PCR Auth Handle: 1 Auth Role: USER
TPML_DIGEST_VALUES	digests	list of tagged digest values to be extended

Table 111 — TPM2_PCR_Extend Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

22.2.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PCR_Extend_fp.h"
3  #if CC_PCR_Extend  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_LOCALITY	current command locality is not allowed to extend the PCR referenced by *pcrHandle*

```
4  TPM_RC
5  TPM2_PCR_Extend(
6      PCR_Extend_In    *in              // IN: input parameter list
7      )
8  {
9      UINT32              i;
10
11 // Input Validation
12
13     // NOTE: This function assumes that the unmarshaling function for 'digests' will
14     // have validated that all of the indicated hash algorithms are valid. If the
15     // hash algorithms are correct, the unmarshaling code will unmarshal a digest
16     // of the size indicated by the hash algorithm. If the overall size is not
17     // consistent, the unmarshaling code will run out of input data or have input
18     // data left over. In either case, it will cause an unmarshaling error and this
19     // function will not be called.
20
21     // For NULL handle, do nothing and return success
22     if(in->pcrHandle == TPM_RH_NULL)
23         return TPM_RC_SUCCESS;
24
25     // Check if the extend operation is allowed by the current command locality
26     if(!PCRIsExtendAllowed(in->pcrHandle))
27         return TPM_RC_LOCALITY;
28
29     // If PCR is state saved and we need to update orderlyState, check NV
30     // availability
31     if(PCRIsStateSaved(in->pcrHandle))
32         RETURN_IF_ORDERLY;
33
34 // Internal Data Update
35
36     // Iterate input digest list to extend
37     for(i = 0; i < in->digests.count; i++)
38     {
39         PCRExtend(in->pcrHandle, in->digests.digests[i].hashAlg,
40                   CryptHashGetDigestSize(in->digests.digests[i].hashAlg),
41                   (BYTE *)&in->digests.digests[i].digest);
42     }
43
44     return TPM_RC_SUCCESS;
45 }
46 #endif // CC_PCR_Extend
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

22.3 TPM2_PCR_Event

22.3.1 General Description

This command is used to cause an update to the indicated PCR.

The data in *eventData* is hashed using the hash algorithm associated with each bank in which the indicated PCR has been allocated. After the data is hashed, the *digests* list is returned. If the *pcrHandle* references an implemented PCR and not TPM_RH_NULL, the *digests* list is processed as in TPM2_PCR_Extend().

A TPM shall support an *Event.size* of zero through 1,024 inclusive (*Event.size* is an octet count). An *Event.size* of zero indicates that there is no data but the indicated operations will still occur,

EXAMPLE 1 If the command implements PCR[2] in a SHA1 bank and a SHA256 bank, then an extend to PCR[2] will cause *eventData* to be hashed twice, once with SHA1 and once with SHA256. The SHA1 hash of *eventData* will be Extended to PCR[2] in the SHA1 bank and the SHA256 hash of *eventData* will be Extended to PCR[2] of the SHA256 bank.

On successful command completion, *digests* will contain the list of tagged digests of *eventData* that was computed in preparation for extending the data into the PCR. At the option of the TPM, the list may contain a digest for each bank, or it may only contain a digest for each bank in which *pcrHandle* is extant. If *pcrHandle* is TPM_RH_NULL, the TPM may return either an empty list or a digest for each bank.

EXAMPLE 2 Assume a TPM that implements a SHA1 bank and a SHA256 bank and that PCR[22] is only implemented in the SHA1 bank. If *pcrHandle* references PCR[22], then *digests* may contain either a SHA1 and a SHA256 digest or just a SHA1 digest.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

22.3.2 Command and Response

Table 112 — TPM2_PCR_Event Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_Event {NV}
TPMI_DH_PCR+	@pcrHandle	Handle of the PCR Auth Handle: 1 Auth Role: USER
TPM2B_EVENT	eventData	Event data in sized buffer

Table 113 — TPM2_PCR_Event Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPML_DIGEST_VALUES	digests	

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

22.3.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PCR_Event_fp.h"
3  #if CC_PCR_Event  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_LOCALITY	current command locality is not allowed to extend the PCR referenced by *pcrHandle*

```
4  TPM_RC
5  TPM2_PCR_Event(
6      PCR_Event_In    *in,            // IN: input parameter list
7      PCR_Event_Out   *out            // OUT: output parameter list
8      )
9  {
10     HASH_STATE          hashState;
11     UINT32              i;
12     UINT16              size;
13
14 // Input Validation
15
16     // If a PCR extend is required
17     if(in->pcrHandle != TPM_RH_NULL)
18     {
19         // If the PCR is not allow to extend, return error
20         if(!PCRIsExtendAllowed(in->pcrHandle))
21             return TPM_RC_LOCALITY;
22
23         // If PCR is state saved and we need to update orderlyState, check NV
24         // availability
25         if(PCRIsStateSaved(in->pcrHandle))
26             RETURN_IF_ORDERLY;
27     }
28
29 // Internal Data Update
30
31     out->digests.count = HASH_COUNT;
32
33     // Iterate supported PCR bank algorithms to extend
34     for(i = 0; i < HASH_COUNT; i++)
35     {
36         TPM_ALG_ID  hash = CryptHashGetAlgByIndex(i);
37         out->digests.digests[i].hashAlg = hash;
38         size = CryptHashStart(&hashState, hash);
39         CryptDigestUpdate2B(&hashState, &in->eventData.b);
40         CryptHashEnd(&hashState, size,
41                      (BYTE *)&out->digests.digests[i].digest);
42         if(in->pcrHandle != TPM_RH_NULL)
43             PCRExtend(in->pcrHandle, hash, size,
44                       (BYTE *)&out->digests.digests[i].digest);
45     }
46
47     return TPM_RC_SUCCESS;
48 }
49 #endif // CC_PCR_Event
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

22.4 TPM2_PCR_Read

22.4.1 General Description

This command returns the values of all PCR specified in *pcrSelectionIn*.

The TPM will process the list of TPMS_PCR_SELECTION in *pcrSelectionIn* in order. Within each TPMS_PCR_SELECTION, the TPM will process the bits in the *pcrSelect* array in ascending PCR order (see TPM 2.0 Part 1, *Selecting Multiple PCR*). If a bit is SET, and the indicated PCR is present, then the TPM will add the digest of the PCR to the list of values to be returned in *pcrValues.*

The TPM will continue processing bits until all have been processed or until *pcrValues* would be too large to fit into the output buffer if additional values were added.

The returned *pcrSelectionOut* will have a bit SET in its *pcrSelect* structures for each value present in *pcrValues*.

The current value of the PCR Update Counter is returned in *pcrUpdateCounter*.

The returned list may be empty if none of the selected PCR are implemented.

NOTE If no PCR are returned from a bank, the selector for the bank will be present in *pcrSelectionOut*.

No authorization is required to read a PCR and any implemented PCR may be read from any locality.

 Copyright © TCG 2006-2020

22.4.2 Command and Response

Table 114 — TPM2_PCR_Read Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_Read
TPML_PCR_SELECTION	pcrSelectionIn	The selection of PCR to read

Table 115 — TPM2_PCR_Read Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
UINT32	pcrUpdateCounter	the current value of the PCR update counter
TPML_PCR_SELECTION	pcrSelectionOut	the PCR in the returned list
TPML_DIGEST	pcrValues	the contents of the PCR indicated in *pcrSelectOut->pcrSelection[]* as tagged digests

 Copyright © TCG 2006-2020

22.4.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PCR_Read_fp.h"
3  #if CC_PCR_Read  // Conditional expansion of this file
4  TPM_RC
5  TPM2_PCR_Read(
6      PCR_Read_In     *in,            // IN: input parameter list
7      PCR_Read_Out    *out            // OUT: output parameter list
8      )
9  {
10 // Command Output
11
12     // Call PCR read function.  input pcrSelectionIn parameter could be changed
13     // to reflect the actual PCR being returned
14     PCRRead(&in->pcrSelectionIn, &out->pcrValues, &out->pcrUpdateCounter);
15
16     out->pcrSelectionOut = in->pcrSelectionIn;
17
18     return TPM_RC_SUCCESS;
19 }
20 #endif // CC_PCR_Read
```

 Copyright © TCG 2006-2020

22.5 TPM2_PCR_Allocate

22.5.1 General Description

This command is used to set the desired PCR allocation of PCR and algorithms. This command requires Platform Authorization.

The TPM will evaluate the request and, if sufficient memory is available for the requested allocation, the TPM will store the allocation request for use during the next _TPM_Init operation. The PCR allocation in place when this command is executed will be retained until the next _TPM_Init. If this command is received multiple times before a _TPM_Init, each one overwrites the previous stored allocation.

This command will only change the allocations of banks that are listed in *pcrAllocation*.

EXAMPLE 1 If a TPM supports SHA1 and SHA256, then it maintains an allocation for two banks (one of which could be empty). If *pcrAllocation* only has a selector for the SHA1 bank, then only the allocation of the SHA1 bank will be changed and the SHA256 bank will remain unchanged. To change the allocation of a TPM from 24 SHA1 PCR and no SHA256 PCR to 24 SHA256 PCR and no SHA1 PCR, the *pcrAllocation* would have to have two selections: one for the empty SHA1 bank and one for the SHA256 bank with 24 PCR.

If a bank is listed more than once, then the last selection in the *pcrAllocation* list is the one that the TPM will attempt to allocate.

NOTE 1 This does not mean to imply that *pcrAllocation.count* can exceed HASH_COUNT, the number of digests implemented in the TPM.

EXAMPLE 2 If HASH_COUNT is 2, *pcrAllocation* can specify SHA-256 twice, and the second one is used. However, if SHA_256 is specified three times, the unmarshaling may fail and the TPM may return an error.

This command shall not allocate more PCR in any bank than there are PCR attribute definitions. The PCR attribute definitions indicate how a PCR is to be managed – if it is resettable, the locality for update, etc. In the response to this command, the TPM returns the maximum number of PCR allowed for any bank.

When PCR are allocated, if DRTM_PCR is defined, the resulting allocation must have at least one bank with the D-RTM PCR allocated. If HCRTM_PCR is defined, the resulting allocation must have at least one bank with the HCRTM_PCR allocated. If not, the TPM returns TPM_RC_PCR.

The TPM may return TPM_RC_SUCCESS even though the request fails. This is to allow the TPM to return information about the size needed for the requested allocation and the size available. If the *sizeNeeded* parameter in the return is less than or equal to the *sizeAvailable* parameter, then the *allocationSuccess* parameter will be YES. Alternatively, if the request fails, The TPM may return TPM_RC_NO_RESULT.

NOTE 2 An example for this type of failure is a TPM that can only support one bank at a time and cannot support arbitrary distribution of PCR among banks.

After this command, TPM2_Shutdown() is only allowed to have a *startupType* equal to TPM_SU_CLEAR until after the next _TPM_Init.

NOTE 3 Even if this command does not cause the PCR allocation to change, the TPM cannot have its state saved. This is done in order to simplify the implementation. There is no need to optimize this command as it is not expected to be used more than once in the lifetime of the TPM (it can be used any number of times but there is no justification for optimization).

 Copyright © TCG 2006-2020

22.5.2 Command and Response

Table 116 — TPM2_PCR_Allocate Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_Allocate {NV}
TPMI_RH_PLATFORM	@authHandle	TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPML_PCR_SELECTION	pcrAllocation	the requested allocation

Table 117 — TPM2_PCR_Allocate Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_YES_NO	allocationSuccess	YES if the allocation succeeded
UINT32	maxPCR	maximum number of PCR that may be in a bank
UINT32	sizeNeeded	number of octets required to satisfy the request
UINT32	sizeAvailable	Number of octets available. Computed before the allocation.

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

22.5.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PCR_Allocate_fp.h"
3  #if CC_PCR_Allocate  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_PCR	the allocation did not have required PCR
TPM_RC_NV_UNAVAILABLE	NV is not accessible
TPM_RC_NV_RATE	NV is in a rate-limiting mode

```
4  TPM_RC
5  TPM2_PCR_Allocate(
6      PCR_Allocate_In     *in,            // IN: input parameter list
7      PCR_Allocate_Out    *out            // OUT: output parameter list
8      )
9  {
10     TPM_RC      result;
11
12     // The command needs NV update.  Check if NV is available.
13     // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
14     // this point.
15     // Note: These codes are not listed in the return values above because it is
16     // an implementation choice to check in this routine rather than in a common
17     // function that is called before these actions are called. These return values
18     // are described in the Response Code section of Part 3.
19     RETURN_IF_NV_IS_NOT_AVAILABLE;
20
21 // Command Output
22
23     // Call PCR Allocation function.
24     result = PCRAllocate(&in->pcrAllocation, &out->maxPCR,
25                          &out->sizeNeeded, &out->sizeAvailable);
26     if(result == TPM_RC_PCR)
27         return result;
28
29     //
30     out->allocationSuccess = (result == TPM_RC_SUCCESS);
31
32     // if re-configuration succeeds, set the flag to indicate PCR configuration is
33     // going to be changed in next boot
34     if(out->allocationSuccess == YES)
35         g_pcrReConfig = TRUE;
36
37     return TPM_RC_SUCCESS;
38 }
39 #endif // CC_PCR_Allocate
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

22.6 TPM2_PCR_SetAuthPolicy

22.6.1 General Description

This command is used to associate a policy with a PCR or group of PCR. The policy determines the conditions under which a PCR may be extended or reset.

A policy may only be associated with a PCR that has been defined by a platform-specific specification as allowing a policy. If the TPM implementation does not allow a policy for *pcrNum*, the TPM shall return TPM_RC_VALUE.

A platform-specific specification may group PCR so that they share a common policy. In such case, a *pcrNum* that selects any of the PCR in the group will change the policy for all PCR in the group.

The policy setting is persistent and may only be changed by TPM2_PCR_SetAuthPolicy() or by TPM2_ChangePPS().

Before this command is first executed on a TPM or after TPM2_ChangePPS(), the access control on the PCR will be set to the default value defined in the platform-specific specification.

NOTE 1 It is expected that the typical default will be with the policy hash set to TPM_ALG_NULL and an Empty Buffer for the *authPolicy* value. This will allow an *EmptyAuth* to be used as the authorization value.

If the size of the data buffer in *authPolicy* is not the size of a digest produced by *hashAlg*, the TPM shall return TPM_RC_SIZE.

NOTE 2 If *hashAlg* is TPM_ALG_NULL, then the size is required to be zero.

This command requires platformAuth/platformPolicy.

NOTE 3 If the PCR is in multiple policy sets, the policy will be changed in only one set. The set that is changed will be implementation dependent.

 Copyright © TCG 2006-2020

22.6.2 Command and Response

Table 118 — TPM2_PCR_SetAuthPolicy Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_SetAuthPolicy {NV}
TPMI_RH_PLATFORM	@authHandle	TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPM2B_DIGEST	authPolicy	the desired *authPolicy*
TPMI_ALG_HASH+	hashAlg	the hash algorithm of the policy
TPMI_DH_PCR	pcrNum	the PCR for which the policy is to be set

Table 119 — TPM2_PCR_SetAuthPolicy Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

22.6.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PCR_SetAuthPolicy_fp.h"
3  #if CC_PCR_SetAuthPolicy  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_SIZE	size of *authPolicy* is not the size of a digest produced by *policyDigest*
TPM_RC_VALUE	PCR referenced by *pcrNum* is not a member of a PCR policy group

```
4  TPM_RC
5  TPM2_PCR_SetAuthPolicy(
6      PCR_SetAuthPolicy_In    *in              // IN: input parameter list
7      )
8  {
9      UINT32      groupIndex;
10
11     // The command needs NV update.  Check if NV is available.
12     // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
13     // this point
14     RETURN_IF_NV_IS_NOT_AVAILABLE;
15
16 // Input Validation:
17
18     // Check the authPolicy consistent with hash algorithm
19     if(in->authPolicy.t.size != CryptHashGetDigestSize(in->hashAlg))
20         return TPM_RCS_SIZE + RC_PCR_SetAuthPolicy_authPolicy;
21
22     // If PCR does not belong to a policy group, return TPM_RC_VALUE
23     if(!PCRBelongsPolicyGroup(in->pcrNum, &groupIndex))
24         return TPM_RCS_VALUE + RC_PCR_SetAuthPolicy_pcrNum;
25
26 // Internal Data Update
27
28     // Set PCR policy
29     gp.pcrPolicies.hashAlg[groupIndex] = in->hashAlg;
30     gp.pcrPolicies.policy[groupIndex] = in->authPolicy;
31
32     // Save new policy to NV
33     NV_SYNC_PERSISTENT(pcrPolicies);
34
35     return TPM_RC_SUCCESS;
36 }
37 #endif // CC_PCR_SetAuthPolicy
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

22.7 TPM2_PCR_SetAuthValue

22.7.1 General Description

This command changes the *authValue* of a PCR or group of PCR.

An *authValue* may only be associated with a PCR that has been defined by a platform-specific specification as allowing an authorization value. If the TPM implementation does not allow an authorization for *pcrNum*, the TPM shall return TPM_RC_VALUE. A platform-specific specification may group PCR so that they share a common authorization value. In such case, a *pcrNum* that selects any of the PCR in the group will change the *authValue* value for all PCR in the group.

The authorization setting is set to EmptyAuth on each STARTUP(CLEAR) or by TPM2_Clear(). The authorization setting is preserved by SHUTDOWN(STATE).

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

22.7.2 Command and Response

Table 120 — TPM2_PCR_SetAuthValue Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_SetAuthValue
TPMI_DH_PCR	@pcrHandle	handle for a PCR that may have an authorization value set Auth Index: 1 Auth Role: USER
TPM2B_DIGEST	auth	the desired authorization value

Table 121 — TPM2_PCR_SetAuthValue Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Copyright © TCG 2006-2020

22.7.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PCR_SetAuthValue_fp.h"
3  #if CC_PCR_SetAuthValue  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_VALUE	PCR referenced by *pcrHandle* is not a member of a PCR authorization group

```
4  TPM_RC
5  TPM2_PCR_SetAuthValue(
6      PCR_SetAuthValue_In     *in              // IN: input parameter list
7      )
8  {
9      UINT32      groupIndex;
10 // Input Validation:
11
12     // If PCR does not belong to an auth group, return TPM_RC_VALUE
13     if(!PCRBelongsAuthGroup(in->pcrHandle, &groupIndex))
14         return TPM_RC_VALUE;
15
16     // The command may cause the orderlyState to be cleared due to the update of
17     // state clear data.  If this is the case, Check if NV is available.
18     // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
19     // this point
20     RETURN_IF_ORDERLY;
21
22 // Internal Data Update
23
24     // Set PCR authValue
25     MemoryRemoveTrailingZeros(&in->auth);
26     gc.pcrAuthValues.auth[groupIndex] = in->auth;
27
28     return TPM_RC_SUCCESS;
29 }
30 #endif // CC_PCR_SetAuthValue
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

22.8 TPM2_PCR_Reset

22.8.1 General Description

If the attribute of a PCR allows the PCR to be reset and proper authorization is provided, then this command may be used to set the PCR in all banks to zero. The attributes of the PCR may restrict the locality that can perform the reset operation.

NOTE 1 The definition of TPMI_DH_PCR in TPM 2.0 Part 2 indicates that if pcrHandle is out of the allowed range for PCR, then the appropriate return value is TPM_RC_VALUE.

If *pcrHandle* references a PCR that cannot be reset, the TPM shall return TPM_RC_LOCALITY.

NOTE 2 TPM_RC_LOCALITY is returned because the reset attributes are defined on a per-locality basis.

 Copyright © TCG 2006-2020

22.8.2 Command and Response

Table 122 — TPM2_PCR_Reset Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_Reset {NV}
TPMI_DH_PCR	@pcrHandle	the PCR to reset Auth Index: 1 Auth Role: USER

Table 123 — TPM2_PCR_Reset Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

22.8.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PCR_Reset_fp.h"
3  #if CC_PCR_Reset  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_LOCALITY	current command locality is not allowed to reset the PCR referenced by *pcrHandle*

```
4  TPM_RC
5  TPM2_PCR_Reset(
6      PCR_Reset_In     *in             // IN: input parameter list
7      )
8  {
9  // Input Validation
10
11     // Check if the reset operation is allowed by the current command locality
12     if(!PCRIsResetAllowed(in->pcrHandle))
13         return TPM_RC_LOCALITY;
14
15     // If PCR is state saved and we need to update orderlyState, check NV
16     // availability
17     if(PCRIsStateSaved(in->pcrHandle))
18         RETURN_IF_ORDERLY;
19
20 // Internal Data Update
21
22     // Reset selected PCR in all banks to 0
23     PCRSetValue(in->pcrHandle, 0);
24
25     // Indicate that the PCR changed so that pcrCounter will be incremented if
26     // necessary.
27     PCRChanged(in->pcrHandle);
28
29     return TPM_RC_SUCCESS;
30 }
31 #endif // CC_PCR_Reset
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

22.9 _TPM_Hash_Start

22.9.1 Description

This indication from the TPM interface indicates the start of an H-CRTM measurement sequence. On receipt of this indication, the TPM will initialize an H-CRTM Event Sequence context.

If no object memory is available for creation of the sequence context, the TPM will flush the context of an object so that creation of the sequence context will always succeed.

A platform-specific specification may allow this indication before TPM2_Startup().

NOTE If this indication occurs after TPM2_Startup(), it is the responsibility of software to ensure that an object context slot is available or to deal with the consequences of having the TPM select an arbitrary object to be flushed. If this indication occurs before TPM2_Startup() then all context slots are available.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

22.9.2 Detailed Actions

```
1   #include "Tpm.h"
```

This function is called to process a _TPM_Hash_Start() indication.

```
2   LIB_EXPORT void
3   _TPM_Hash_Start(
4       void
5       )
6   {
7       TPM_RC              result;
8       TPMI_DH_OBJECT      handle;
9
10      // If a DRTM sequence object exists, free it up
11      if(g_DRTMHandle != TPM_RH_UNASSIGNED)
12      {
13          FlushObject(g_DRTMHandle);
14          g_DRTMHandle = TPM_RH_UNASSIGNED;
15      }
16
17      // Create an event sequence object and store the handle in global
18      // g_DRTMHandle. A TPM_RC_OBJECT_MEMORY error may be returned at this point
19      // The NULL value for the first parameter will cause the sequence structure to
20      // be allocated without being set as present. This keeps the sequence from
21      // being left behind if the sequence is terminated early.
22      result = ObjectCreateEventSequence(NULL, &g_DRTMHandle);
23
24      // If a free slot was not available, then free up a slot.
25      if(result != TPM_RC_SUCCESS)
26      {
27          // An implementation does not need to have a fixed relationship between
28          // slot numbers and handle numbers. To handle the general case, scan for
29          // a handle that is assigned and free it for the DRTM sequence.
30          // In the reference implementation, the relationship between handles and
31          // slots is fixed. So, if the call to ObjectCreateEvenSequence()
32          // failed indicating that all slots are occupied, then the first handle we
33          // are going to check (TRANSIENT_FIRST) will be occupied. It will be freed
34          // so that it can be assigned for use as the DRTM sequence object.
35          for(handle = TRANSIENT_FIRST; handle < TRANSIENT_LAST; handle++)
36          {
37              // try to flush the first object
38              if(IsObjectPresent(handle))
39                  break;
40          }
41          // If the first call to find a slot fails but none of the slots is occupied
42          // then there's a big problem
43          pAssert(handle < TRANSIENT_LAST);
44
45          // Free the slot
46          FlushObject(handle);
47
48          // Try to create an event sequence object again.  This time, we must
49          // succeed.
50          result = ObjectCreateEventSequence(NULL, &g_DRTMHandle);
51          if(result != TPM_RC_SUCCESS)
52              FAIL(FATAL_ERROR_INTERNAL);
53      }
54
55      return;
56  }
```

22.10 _TPM_Hash_Data

22.10.1 Description

This indication from the TPM interface indicates arrival of one or more octets of data that are to be included in the H-CRTM Event Sequence sequence context created by the _TPM_Hash_Start indication. The context holds data for each hash algorithm for each PCR bank implemented on the TPM.

If no H-CRTM Event Sequence context exists, this indication is discarded and no other action is performed.

 Copyright © TCG 2006-2020

22.10.2 Detailed Actions

```
1  #include "Tpm.h"
```

This function is called to process a _TPM_Hash_Data() indication.

```
2  LIB_EXPORT void
3  _TPM_Hash_Data(
4      uint32_t         dataSize,       // IN: size of data to be extend
5      unsigned char   *data            // IN: data buffer
6      )
7  {
8      UINT32           i;
9      HASH_OBJECT     *hashObject;
10     TPMI_DH_PCR      pcrHandle = TPMIsStarted()
11         ? PCR_FIRST + DRTM_PCR : PCR_FIRST + HCRTM_PCR;
12
13 // If there is no DRTM sequence object, then _TPM_Hash_Start
14 // was not called so this function returns without doing
15 // anything.
16     if(g_DRTMHandle == TPM_RH_UNASSIGNED)
17         return;
18
19     hashObject = (HASH_OBJECT *)HandleToObject(g_DRTMHandle);
20     pAssert(hashObject->attributes.eventSeq);
21
22     // For each of the implemented hash algorithms, update the digest with the
23     // data provided.
24     for(i = 0; i < HASH_COUNT; i++)
25     {
26         // make sure that the PCR is implemented for this algorithm
27         if(PcrIsAllocated(pcrHandle,
28                           hashObject->state.hashState[i].hashAlg))
29           // Update sequence object
30             CryptDigestUpdate(&hashObject->state.hashState[i], dataSize, data);
31     }
32
33     return;
34 }
```

22.11 _TPM_Hash_End

22.11.1 Description

This indication from the TPM interface indicates the end of the H-CRTM measurement. This indication is discarded and no other action performed if the TPM does not contain an H-CRTM Event Sequence context.

NOTE 1 An H-CRTM Event Sequence context is created by _TPM_Hash_Start().

If the H-CRTM Event Sequence occurs after TPM2_Startup(), the TPM will set all of the PCR designated in the platform-specific specifications as resettable by this event to the value indicated in the platform specific specification and increment *restartCount*. The TPM will then Extend the Event Sequence digest/digests into the designated D-RTM PCR (PCR[17]).

$$\text{PCR}[17][hashAlg] \coloneqq \mathbf{H}_{hashAlg}(initial_value \,||\, \mathbf{H}_{hashAlg}(hash_data)) \quad (7)$$

where

$hashAlg$	hash algorithm associated with a bank of PCR
$initial_value$	initialization value specified in the platform-specific specification (should be 0…0)
$hash_data$	all the octets of data received in _TPM_Hash_Data indications

A _TPM_Hash_End indication that occurs after TPM2_Startup() will increment *pcrUpdateCounter* unless a platform-specific specification excludes modifications of PCR[DRTM] from causing an increment.

A platform-specific specification may allow an H-CRTM Event Sequence before TPM2_Startup(). If so, _TPM_Hash_End will complete the digest, initialize PCR[0] with a digest-size value of 4, and then extend the H-CRTM Event Sequence data into PCR[0].

$$\text{PCR}[0][hashAlg] \coloneqq \mathbf{H}_{hashAlg}(0\ldots04 \,||\, \mathbf{H}_{hashAlg}(hash_data)) \quad (8)$$

NOTE 2 The entire sequence of _TPM_Hash_Start, _TPM_Hash_Data, and _TPM_Hash_End are required to complete before TPM2_Startup() or the sequence will have no effect on the TPM.

NOTE 3 PCR[0] does not need to be updated according to (8) until the end of TPM2_Startup().

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

22.11.2 Detailed Actions

```
1    #include "Tpm.h"
```

This function is called to process a _TPM_Hash_End() indication.

```
2    LIB_EXPORT void
3    _TPM_Hash_End(
4        void
5        )
6    {
7        UINT32          i;
8        TPM2B_DIGEST    digest;
9        HASH_OBJECT    *hashObject;
10       TPMI_DH_PCR     pcrHandle;
11
12       // If the DRTM handle is not being used, then either _TPM_Hash_Start has not
13       // been called, _TPM_Hash_End was previously called, or some other command
14       // was executed and the sequence was aborted.
15       if(g_DRTMHandle == TPM_RH_UNASSIGNED)
16           return;
17
18       // Get DRTM sequence object
19       hashObject = (HASH_OBJECT *)HandleToObject(g_DRTMHandle);
20
21       // Is this _TPM_Hash_End after Startup or before
22       if(TPMIsStarted())
23       {
24           // After
25
26           // Reset the DRTM PCR
27           PCRResetDynamics();
28
29           // Extend the DRTM_PCR.
30           pcrHandle = PCR_FIRST + DRTM_PCR;
31
32           // DRTM sequence increments restartCount
33           gr.restartCount++;
34       }
35       else
36       {
37           pcrHandle = PCR_FIRST + HCRTM_PCR;
38           g_DrtmPreStartup = TRUE;
39       }
40
41       // Complete hash and extend PCR, or if this is an HCRTM, complete
42       // the hash, reset the H-CRTM register (PCR[0]) to 0...04, and then
43       // extend the H-CRTM data
44       for(i = 0; i < HASH_COUNT; i++)
45       {
46           TPMI_ALG_HASH       hash = CryptHashGetAlgByIndex(i);
47           // make sure that the PCR is implemented for this algorithm
48           if(PcrIsAllocated(pcrHandle,
49                             hashObject->state.hashState[i].hashAlg))
50           {
51               // Complete hash
52               digest.t.size = CryptHashGetDigestSize(hash);
53               CryptHashEnd2B(&hashObject->state.hashState[i], &digest.b);
54
55               PcrDrtm(pcrHandle, hash, &digest);
56           }
57       }
58
59       // Flush sequence object.
```

```
60        FlushObject(g_DRTMHandle);
61
62        g_DRTMHandle = TPM_RH_UNASSIGNED;
63
64        return;
65    }
```

 Copyright © TCG 2006-2020

23 Enhanced Authorization (EA) Commands

23.1 Introduction

The commands in this clause 1 are used for policy evaluation. When successful, each command will update the *policySession→policyDigest* in a policy session context in order to establish that the authorizations required to use an object have been provided. Many of the commands will also modify other parts of a policy context so that the caller may constrain the scope of the authorization that is provided.

NOTE 1 Many of the terms used in this clause are described in detail in TPM 2.0 Part 1 and are not redefined in this clause.

The *policySession* parameter of the command is the handle of the policy session context to be modified by the command.

If the *policySession* parameter indicates a trial policy session, then the *policySession→policyDigest* will be updated and the indicated validations are not performed. However, any authorizations required to perform the policy command will be checked and dictionary attack logic invoked as necessary.

NOTE 2 If software is used to create policies, no authorization values are used. For example, TPM_PolicySecret requires an authorization in a trial policy session, but not in a policy calculation outside the TPM.

NOTE 3 A policy session is set to a trial policy by TPM2_StartAuthSession(*sessionType* = TPM_SE_TRIAL).

NOTE 4 Unless there is an unmarshaling error in the parameters of the command, these commands will return TPM_RC_SUCCESS when *policySession* references a trial session.

NOTE 5 Policy context other than the *policySession→policyDigest* may be updated for a trial policy but it is not required.

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

23.2 Signed Authorization Actions

23.2.1 Introduction

The TPM2_PolicySigned, TPM_PolicySecret, and TPM2_PolicyTicket commands use many of the same functions. This clause consolidates those functions to simplify the document and to ensure uniformity of the operations.

23.2.2 Policy Parameter Checks

These parameter checks will be performed when indicated in the description of each of the commands:

a) *nonceTPM* – If this parameter is not the Empty Buffer, and it does not match *policySession→nonceTPM*, then the TPM shall return TPM_RC_VALUE.

b) *expiration* – If this parameter is not zero, then:

   1) if *nonceTPM* is not an Empty Buffer, then the absolute value of *expiration* is converted to milliseconds and added to *policySession→startTime* to create the *timeout* value and proceed to c).

   2) If *nonceTPM* is an Empty Buffer, then the absolute value of *expiration* is converted to milliseconds and used as the *timeout* value and proceed to c).

   However, *timeout* can only be changed to a smaller value.

c) *timeout* – If *timeout* is less than the current value of *Time*, or the current *timeEpoch* is not the same as *policySession→timeEpoch, the* TPM shall return TPM_RC_EXPIRED

d) *cpHashA* – If this parameter is not an Empty Buffer

   NOTE 2 *cpHashA* is the hash of the command to be executed using this policy session in the authorization. The algorithm used to compute this hash is required to be the algorithm of the policy session.

   1) the TPM shall return TPM_RC_CPHASH if *policySession→cpHash* is set and the contents of *policySession→cpHash* are not the same as *cpHashA*; or

      NOTE 3 cpHash is the expected cpHash value held in the policy session context.

   2) the TPM shall return TPM_RC_SIZE if *cpHashA* is not the same size as *policySession→policyDigest*.

      NOTE 4 policySession→policyDigest is the size of the digest produced by the hash algorithm used to compute policyDigest.

23.2.3 Policy Digest Update Function (PolicyUpdate())

This is the update process for *policySession→policyDigest* used by TPM2_PolicySigned(), TPM2_PolicySecret(), TPM2_PolicyTicket(), and TPM2_PolicyAuthorize(). The function prototype for the update function is:

$$\mathbf{PolicyUpdate}(commandCode, arg2, arg3) \tag{9}$$

where

$arg2$	a TPM2B_NAME
$arg3$	a TPM2B

These parameters are used to update *policySession→policyDigest* by

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mid\mid commandCode \mid\mid arg2.name) \tag{10}$$

followed by

$$policyDigest_{new+1} \coloneqq \mathbf{H}_{\mathrm{policyAlg}}(policyDigest_{new} \mid\mid arg3.buffer) \tag{11}$$

where

$\mathbf{H}_{policyAlg}()$	the hash algorithm chosen when the policy session was started

NOTE 1 If *arg3* is a TPM2B_NAME, then *arg3.buffer* will actually be an *arg3.name*.

NOTE 2 The *arg2.size* and *arg3.size* fields are not included in the hashes.

NOTE 3 **PolicyUpdate**() uses two hash operations because *arg2* and *arg3* are variable-sized and the concatenation of *arg2* and *arg3* in a single hash could produce the same digest even though *arg2* and *arg3* are different. For example, arg2 = 1 2 3 and arg3 = 4 5 6 would produce the same digest as arg2 = 1 2 and arg3 = 3 4 5 6. Processing of the arguments separately in different Extend operation ensures that the digest produced by **PolicyUpdate**() will be different if *arg2* and *arg3* are different.

 Copyright © TCG 2006-2020

23.2.4 Policy Context Updates

When a policy command modifies some part of the policy session context other than the *policySession→policyDigest*, the following rules apply.

- ***cpHash*** – this parameter may only be changed if it contains its initialization value (an Empty Buffer). If *cpHash* is not the Empty Buffer when a policy command attempts to update it, the TPM will return an error (TPM_RC_CPHASH) if the current and update values are not the same.
- ***timeOut*** – this parameter may only be changed to a smaller value. If a command attempts to update this value with a larger value (longer into the future), the TPM will discard the update value. This is not an error condition.
- ***commandCode*** – once set by a policy command, this value may not be changed except by TPM2_PolicyRestart(). If a policy command tries to change this to a different value, an error is returned (TPM_RC_POLICY_CC).
- ***pcrUpdateCounter*** – this parameter is updated by TPM2_PolicyPCR(). This value may only be set once during a policy. Each time TPM2_PolicyPCR() executes, it checks to see if *policySession→pcrUpdateCounter* has its default state, indicating that this is the first TPM2_PolicyPCR(). If it has its default value, then *policySession→pcrUpdateCounter* is set to the current value of *pcrUpdateCounter*. If *policySession→pcrUpdateCounter* does not have its default value and its value is not the same as *pcrUpdateCounter*, the TPM shall return TPM_RC_PCR_CHANGED.

  NOTE 1 If this parameter and *pcrUpdateCounter* are not the same, it indicates that PCR have changed since checked by the previous TPM2_PolicyPCR(). Since they have changed, the previous PCR validation is no longer valid.

- ***commandLocality*** – this parameter is the logical AND of all enabled localities. All localities are enabled for a policy when the policy session is created. TPM2_PolicyLocalities() selectively disables localities. Once use of a policy for a locality has been disabled, it cannot be enabled except by TPM2_PolicyRestart().
- ***isPPRequired*** – once SET, this parameter may only be CLEARed by TPM2_PolicyRestart().
- ***isAuthValueNeeded*** – once SET, this parameter may only be CLEARed by TPM2_PolicyPassword() or TPM2_PolicyRestart().
- ***isPasswordNeeded*** – once SET, this parameter may only be CLEARed by TPM2_PolicyAuthValue() or TPM2_PolicyRestart(),

NOTE 2 Both TPM2_PolicyAuthValue() and TPM2_PolicyPassword() change *policySession→policyDigest* in the same way. The different commands simply indicate to the TPM the format used for the *authValue* (HMAC or clear text). Both commands could be in the same policy. The final instance of these commands determines the format.

 Copyright © TCG 2006-2020

23.2.5 Policy Ticket Creation

For TPM2_PolicySigned() or TPM2_PolicySecret(), if the caller specified a negative value for *expiration*, then the TPM will return a ticket that includes a value indicating when the authorization expires. Otherwise, the TPM will return a NULL Ticket.

NOTE 1 If the *authHandle* in TPM2_PolicySecret() references a PIN Pass Index, then the command may succeed but a NULL Ticket will be returned.

The required computation for the digest in the authorization ticket is:

$$\mathbf{HMAC}_{contextAlg}(proof, (\text{TPM_ST_AUTH_xxx} \mid\mid cpHash \mid\mid policyRef \mid\mid authName \mid\mid timeout \mid\mid [timeEpoch] \mid\mid [resetCount])) \quad (12)$$

where

$\mathbf{HMAC}_{contextAlg}()$	an HMAC using the context integrity hash
$proof$	a TPM secret value associated with the hierarchy of the object associated with *authName*
TPM_ST_AUTH_xxx	either TPM_ST_AUTH_SIGNED or TPM_ST_AUTH_SECRET; used to ensure that the ticket is properly used
$cpHash$	optional hash of the authorized command
$policyRef$	optional reference to a policy value
$authName$	Name of the object that signed the authorization
$timeout$	implementation-specific value indicating when the authorization expires
$timeEpoch$	implementation-specific representation of the $timeEpoch$ at the time the ticket was created

NOTE 2 Not included if *timeout* is zero.

$resetCount$	implementation-specific representation of the TPM's *totalResetCount*

NOTE 3 Not included it *timeout* is zero or if *nonceTPM* was include in the authorization.

 Copyright © TCG 2006-2020

23.3 TPM2_PolicySigned

23.3.1 General Description

This command includes a signed authorization in a policy. The command ties the policy to a signing key by including the Name of the signing key in the *policyDigest*

If *policySession* is a trial session, the TPM will not check the signature and will update *policySession→policyDigest* as described in 23.2.3 as if a properly signed authorization was received, but no ticket will be produced.

If *policySession* is not a trial session, the TPM will validate *auth* and only perform the update if it is a valid signature over the fields of the command.

The authorizing entity will sign a digest of the authorization qualifiers: *nonceTPM*, *expiration*, *cpHashA*, and *policyRef*. The digest is computed as:

$$aHash \coloneqq \mathbf{H}_{authAlg}(nonceTPM \mid\mid expiration \mid\mid cpHashA \mid\mid policyRef) \qquad (13)$$

where

$\mathbf{H}_{authAlg}()$ the hash associated with the auth parameter of this command

NOTE 1 Each signature and key combination indicates the scheme and each scheme has an associated hash.

$nonceTPM$ the nonceTPM parameter from the TPM2_StartAuthSession() response. If the authorization is not limited to this session, the size of this value is zero.

$expiration$ time limit on authorization set by authorizing object. This 32-bit value is set to zero if the expiration time is not being set.

$cpHashA$ digest of the command parameters for the command being approved using the hash algorithm of the policy session. Set to an Empty Digest if the authorization is not limited to a specific command.

NOTE 3 This is not the *cpHash* of this TPM2_PolicySigned() command.

$policyRef$ an opaque value determined by the authorizing entity. Set to the Empty Buffer if no value is present.

NOTE 4 The *nonceTPM*, *cpHashA*, and *policyRef* qualifiers used to compute *aHash* use the TPM2B buffer but do not prepend the size.

EXAMPLE The computation for an *aHash* if there are no restrictions is:

$$aHash \coloneqq \mathbf{H}_{authAlg}(00\ 00\ 00\ 00_{16})$$

which is the hash of an expiration time of zero.

The *aHash* is signed by the key associated with a key whose handle is *authObject*. The signature and signing parameters are combined to create the *auth* parameter.

The TPM will perform the parameter checks listed in 23.2.2

If the parameter checks succeed, the TPM will construct a test digest (*tHash*) over the provided parameters using the same formulation as shown in equation (13) above.

If *tHash* does not match the digest of the signed *aHash,* then the authorization fails and the TPM shall return TPM_RC_POLICY_FAIL and make no change to *policySession→policyDigest*.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

When all validations have succeeded, *policySession→policyDigest* is updated by **PolicyUpdate**() (see 23.2.3).

$$\mathbf{PolicyUpdate}(\text{TPM_CC_PolicySigned}, authObject{\rightarrow}Name, policyRef) \tag{14}$$

authObject→Name is a TPM2B_NAME. *policySession* is updated as described in 23.2.4. The TPM will optionally produce a ticket as described in 23.2.5.

Authorization to use *authObject* is not required.

 Copyright © TCG 2006-2020

23.3.2 Command and Response

Table 124 — TPM2_PolicySigned Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit, encrypt, or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicySigned
TPMI_DH_OBJECT	authObject	handle for a key that will validate the signature Auth Index: None
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_NONCE	nonceTPM	the policy nonce for the session This can be the Empty Buffer.
TPM2B_DIGEST	cpHashA	digest of the command parameters to which this authorization is limited This is not the *cpHash* for this command but the *cpHash* for the command to which this policy session will be applied. If it is not limited, the parameter will be the Empty Buffer.
TPM2B_NONCE	policyRef	a reference to a policy relating to the authorization – may be the Empty Buffer Size is limited to be no larger than the nonce size supported on the TPM.
INT32	expiration	time when authorization will expire, measured in seconds from the time that *nonceTPM* was generated If *expiration* is non-negative, a NULL Ticket is returned. See 23.2.5.
TPMT_SIGNATURE	auth	signed authorization (not optional)

Table 125 — TPM2_PolicySigned Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_TIMEOUT	timeout	implementation-specific time value, used to indicate to the TPM when the ticket expires NOTE If *policyTicket* is a NULL Ticket, then this shall be the Empty Buffer.
TPMT_TK_AUTH	policyTicket	produced if the command succeeds and *expiration* in the command was non-zero; this ticket will use the TPMT_ST_AUTH_SIGNED structure tag. See 23.2.5

 Copyright © TCG 2006-2020

23.3.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "Policy_spt_fp.h"
3  #include "PolicySigned_fp.h"
4  #if CC_PolicySigned  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_CPHASH	*cpHash* was previously set to a different value
TPM_RC_EXPIRED	*expiration* indicates a time in the past or *expiration* is non-zero but no *nonceTPM* is present
TPM_RC_NONCE	*nonceTPM* is not the nonce associated with the *policySession*
TPM_RC_SCHEME	the signing scheme of *auth* is not supported by the TPM
TPM_RC_SIGNATURE	the signature is not genuine
TPM_RC_SIZE	input *cpHash* has wrong size

```
5  TPM_RC
6  TPM2_PolicySigned(
7      PolicySigned_In     *in,            // IN: input parameter list
8      PolicySigned_Out    *out            // OUT: output parameter list
9      )
10 {
11     TPM_RC                  result = TPM_RC_SUCCESS;
12     SESSION                *session;
13     TPM2B_NAME              entityName;
14     TPM2B_DIGEST            authHash;
15     HASH_STATE              hashState;
16     UINT64                  authTimeout = 0;
17 // Input Validation
18     // Set up local pointers
19     session = SessionGet(in->policySession);    // the session structure
20
21     // Only do input validation if this is not a trial policy session
22     if(session->attributes.isTrialPolicy == CLEAR)
23     {
24         authTimeout = ComputeAuthTimeout(session, in->expiration, &in->nonceTPM);
25
26         result = PolicyParameterChecks(session, authTimeout,
27                                        &in->cpHashA, &in->nonceTPM,
28                                        RC_PolicySigned_nonceTPM,
29                                        RC_PolicySigned_cpHashA,
30                                        RC_PolicySigned_expiration);
31         if(result != TPM_RC_SUCCESS)
32             return result;
33         // Re-compute the digest being signed
34
35         // Start hash
36         authHash.t.size = CryptHashStart(&hashState,
37                                          CryptGetSignHashAlg(&in->auth));
38         // If there is no digest size, then we don't have a verification function
39         // for this algorithm (e.g. TPM_ALG_ECDAA) so indicate that it is a
40         // bad scheme.
41         if(authHash.t.size == 0)
42             return TPM_RCS_SCHEME + RC_PolicySigned_auth;
43
44         //  nonceTPM
45         CryptDigestUpdate2B(&hashState, &in->nonceTPM.b);
46
```

 Copyright © TCG 2006-2020

```
47          //  expiration
48          CryptDigestUpdateInt(&hashState, sizeof(UINT32), in->expiration);
49
50          //  cpHashA
51          CryptDigestUpdate2B(&hashState, &in->cpHashA.b);
52
53          //  policyRef
54          CryptDigestUpdate2B(&hashState, &in->policyRef.b);
55
56          //  Complete digest
57          CryptHashEnd2B(&hashState, &authHash.b);
58
59          // Validate Signature.  A TPM_RC_SCHEME, TPM_RC_HANDLE or TPM_RC_SIGNATURE
60          // error may be returned at this point
61          result = CryptValidateSignature(in->authObject, &authHash, &in->auth);
62          if(result != TPM_RC_SUCCESS)
63              return RcSafeAddToResult(result, RC_PolicySigned_auth);
64      }
65  // Internal Data Update
66      // Update policy with input policyRef and name of authorization key
67      // These values are updated even if the session is a trial session
68      PolicyContextUpdate(TPM_CC_PolicySigned,
69                          EntityGetName(in->authObject, &entityName),
70                          &in->policyRef,
71                          &in->cpHashA, authTimeout, session);
72  // Command Output
73      // Create ticket and timeout buffer if in->expiration < 0 and this is not
74      // a trial session.
75      // NOTE: PolicyParameterChecks() makes sure that nonceTPM is present
76      // when expiration is non-zero.
77      if(in->expiration < 0
78         && session->attributes.isTrialPolicy == CLEAR)
79      {
80          BOOL        expiresOnReset = (in->nonceTPM.t.size == 0);
81          // Compute policy ticket
82          authTimeout &= ~EXPIRATION_BIT;
83
84          TicketComputeAuth(TPM_ST_AUTH_SIGNED, EntityGetHierarchy(in->authObject),
85                            authTimeout, expiresOnReset, &in->cpHashA, &in->policyRef,
86                            &entityName, &out->policyTicket);
87          // Generate timeout buffer.  The format of output timeout buffer is
88          // TPM-specific.
89          // Note: In this implementation, the timeout buffer value is computed after
90          // the ticket is produced so, when the ticket is checked, the expiration
91          // flag needs to be extracted before the ticket is checked.
92          // In the Windows compatible version, the least-significant bit of the
93          // timeout value is used as a flag to indicate if the authorization expires
94          // on reset. The flag is the MSb.
95          out->timeout.t.size = sizeof(authTimeout);
96          if(expiresOnReset)
97              authTimeout |= EXPIRATION_BIT;
98          UINT64_TO_BYTE_ARRAY(authTimeout, out->timeout.t.buffer);
99      }
100     else
101     {
102         // Generate a null ticket.
103         // timeout buffer is null
104         out->timeout.t.size = 0;
105
106         // authorization ticket is null
107         out->policyTicket.tag = TPM_ST_AUTH_SIGNED;
108         out->policyTicket.hierarchy = TPM_RH_NULL;
109         out->policyTicket.digest.t.size = 0;
110     }
111     return TPM_RC_SUCCESS;
112 }
```


 Copyright © TCG 2006-2020

```
113 #endif // CC_PolicySigned
```

23.4 TPM2_PolicySecret

23.4.1 General Description

This command includes a secret-based authorization to a policy. The caller proves knowledge of the secret value using an authorization session using the *authValue* associated with *authHandle*. A password session, an HMAC session, or a policy session containing TPM2_PolicyAuthValue() or TPM2_PolicyPassword() will satisfy this requirement.

If a policy session is used and use of the *authValue* of *authHandle* is not required, the TPM will return TPM_RC_MODE. That is, the session for *authHandle* must have either *isAuthValueNeeded* or *isPasswordNeeded* SET.

The secret is the *authValue* of the entity whose handle is *authHandle,* which may be any TPM entity with a handle and an associated *authValue*. This includes the reserved handles (for example, Platform, Storage, and Endorsement), NV Indexes, and loaded objects. *authEntity* is the entity referenced by *authHandle*. If *authEntity* references an Ordinary object, it must have *userWithAuth* SET.

NOTE 1 The *userWithAuth* requirement permits the implementation to use common authorization code.

If *authEntity* references a non-PIN Index, TPMA_NV_AUTHREAD is required to be SET in the Index. If *authEntity* references an NV PIN index, TPMA_NV_WRITTEN is required to be SET and *pinCount* must be less than *pinLimit*.

NOTE 2 The authorization value for a hierarchy cannot be used in this command if the hierarchy is disabled.

If the authorization check fails, then the normal dictionary attack logic is invoked.

If the authorization provided by the authorization session is valid, the command parameters are checked as described in 23.2.2.

When all validations have succeeded, *policySession→policyDigest* is updated by **PolicyUpdate**() (see 23.2.3).

$$\textbf{PolicyUpdate}(\text{TPM_CC_PolicySecret}, authEntity\rightarrow Name, policyRef) \quad (15)$$

authEntity→Name is a TPM2B_NAME. *policySession* is updated as described in 23.2.4. The TPM will optionally produce a ticket as described in 23.2.5.

If the session is a trial session, *policySession→policyDigest* is updated if the authorization is valid.

NOTE 2 If an HMAC is used to convey the authorization, a separate session is needed for the authorization. Because the HMAC in that authorization will include a nonce that prevents replay of the authorization, the value of the *nonceTPM* parameter in this command is limited. It is retained mostly to provide processing consistency with TPM2_PolicySigned().

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.4.2 Command and Response

Table 126 — TPM2_PolicySecret Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicySecret
TPMI_DH_ENTITY	@authHandle	handle for an entity providing the authorization Auth Index: 1 Auth Role: USER
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_NONCE	nonceTPM	the policy nonce for the session This can be the Empty Buffer.
TPM2B_DIGEST	cpHashA	digest of the command parameters to which this authorization is limited This not the *cpHash* for this command but the *cpHash* for the command to which this policy session will be applied. If it is not limited, the parameter will be the Empty Buffer.
TPM2B_NONCE	policyRef	a reference to a policy relating to the authorization – may be the Empty Buffer Size is limited to be no larger than the nonce size supported on the TPM.
INT32	expiration	time when authorization will expire, measured in seconds from the time that *nonceTPM* was generated If *expiration* is non-negative, a NULL Ticket is returned. See 23.2.5.

Table 127 — TPM2_PolicySecret Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_TIMEOUT	timeout	implementation-specific time value used to indicate to the TPM when the ticket expires
TPMT_TK_AUTH	policyTicket	produced if the command succeeds and *expiration* in the command was non-zero (See 23.2.5). This ticket will use the TPMT_ST_AUTH_SECRET structure tag

 Copyright © TCG 2006-2020

23.4.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "PolicySecret_fp.h"
3   #if CC_PolicySecret // Conditional expansion of this file
4   #include "Policy_spt_fp.h"
5   #include "NV_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_CPHASH	*cpHash* for policy was previously set to a value that is not the same as *cpHashA*
TPM_RC_EXPIRED	*expiration* indicates a time in the past
TPM_RC_NONCE	*nonceTPM* does not match the nonce associated with *policySession*
TPM_RC_SIZE	*cpHashA* is not the size of a digest for the hash associated with *policySession*

```
6   TPM_RC
7   TPM2_PolicySecret(
8       PolicySecret_In     *in,            // IN: input parameter list
9       PolicySecret_Out    *out            // OUT: output parameter list
10      )
11  {
12      TPM_RC                   result;
13      SESSION                 *session;
14      TPM2B_NAME               entityName;
15      UINT64                   authTimeout = 0;
16  // Input Validation
17      // Get pointer to the session structure
18      session = SessionGet(in->policySession);
19
20      //Only do input validation if this is not a trial policy session
21      if(session->attributes.isTrialPolicy == CLEAR)
22      {
23          authTimeout = ComputeAuthTimeout(session, in->expiration, &in->nonceTPM);
24
25          result = PolicyParameterChecks(session, authTimeout,
26                                         &in->cpHashA, &in->nonceTPM,
27                                         RC_PolicySecret_nonceTPM,
28                                         RC_PolicySecret_cpHashA,
29                                         RC_PolicySecret_expiration);
30          if(result != TPM_RC_SUCCESS)
31              return result;
32      }
33  // Internal Data Update
34      // Update policy context with input policyRef and name of authorizing key
35      // This value is computed even for trial sessions. Possibly update the cpHash
36      PolicyContextUpdate(TPM_CC_PolicySecret,
37                          EntityGetName(in->authHandle, &entityName), &in->policyRef,
38                          &in->cpHashA, authTimeout, session);
39  // Command Output
40      // Create ticket and timeout buffer if in->expiration < 0 and this is not
41      // a trial session.
42      // NOTE: PolicyParameterChecks() makes sure that nonceTPM is present
43      // when expiration is non-zero.
44      if(in->expiration < 0
45         && session->attributes.isTrialPolicy == CLEAR
46         && !NvIsPinPassIndex(in->authHandle))
47      {
48          BOOL        expiresOnReset = (in->nonceTPM.t.size == 0);
49          // Compute policy ticket
```

 Copyright © TCG 2006-2020

```
50          authTimeout &= ~EXPIRATION_BIT;
51          TicketComputeAuth(TPM_ST_AUTH_SECRET, EntityGetHierarchy(in->authHandle),
52                            authTimeout, expiresOnReset, &in->cpHashA, &in->policyRef,
53                            &entityName, &out->policyTicket);
54          // Generate timeout buffer.  The format of output timeout buffer is
55          // TPM-specific.
56          // Note: In this implementation, the timeout buffer value is computed after
57          // the ticket is produced so, when the ticket is checked, the expiration
58          // flag needs to be extracted before the ticket is checked.
59          out->timeout.t.size = sizeof(authTimeout);
60          // In the Windows compatible version, the least-significant bit of the
61          // timeout value is used as a flag to indicate if the authorization expires
62          // on reset. The flag is the MSb.
63          if(expiresOnReset)
64              authTimeout |= EXPIRATION_BIT;
65          UINT64_TO_BYTE_ARRAY(authTimeout, out->timeout.t.buffer);
66      }
67      else
68      {
69          // timeout buffer is null
70          out->timeout.t.size = 0;
71
72          // authorization ticket is null
73          out->policyTicket.tag = TPM_ST_AUTH_SECRET;
74          out->policyTicket.hierarchy = TPM_RH_NULL;
75          out->policyTicket.digest.t.size = 0;
76      }
77      return TPM_RC_SUCCESS;
78  }
79  #endif // CC_PolicySecret
```

23.5 TPM2_PolicyTicket

23.5.1 General Description

This command is similar to TPM2_PolicySigned() except that it takes a ticket instead of a signed authorization. The ticket represents a validated authorization that had an expiration time associated with it.

The parameters of this command are checked as described in 23.2.2.

If the checks succeed, the TPM uses the *timeout*, *cpHashA*, *policyRef*, and *authName* to construct a ticket to compare with the value in *ticket*. If these tickets match, then the TPM will create a TPM2B_NAME (*objectName*) using *authName* and update the context of *policySession* by **PolicyUpdate**() (see 23.2.3).

$$\mathbf{PolicyUpdate}(commandCode, authName, policyRef) \quad (16)$$

If the structure tag of ticket is TPM_ST_AUTH_SECRET, then *commandCode* will be TPM_CC_PolicySecret. If the structure tag of ticket is TPM_ST_AUTH_SIGNED, then *commandCode* will be TPM_CC_PolicySIgned.

policySession is updated as described in 23.2.4.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.5.2 Command and Response

Table 128 — TPM2_PolicyTicket Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyTicket
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_TIMEOUT	timeout	time when authorization will expire The contents are TPM specific. This shall be the value returned when ticket was produced.
TPM2B_DIGEST	cpHashA	digest of the command parameters to which this authorization is limited If it is not limited, the parameter will be the Empty Buffer.
TPM2B_NONCE	policyRef	reference to a qualifier for the policy – may be the Empty Buffer
TPM2B_NAME	authName	name of the object that provided the authorization
TPMT_TK_AUTH	ticket	an authorization ticket returned by the TPM in response to a TPM2_PolicySigned() or TPM2_PolicySecret()

Table 129 — TPM2_PolicyTicket Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

23.5.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PolicyTicket_fp.h"
3  #if CC_PolicyTicket  // Conditional expansion of this file
4  #include "Policy_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_CPHASH	policy's *cpHash* was previously set to a different value
TPM_RC_EXPIRED	*timeout* value in the ticket is in the past and the ticket has expired
TPM_RC_SIZE	*timeout* or *cpHash* has invalid size for the
TPM_RC_TICKET	*ticket* is not valid

```
5  TPM_RC
6  TPM2_PolicyTicket(
7      PolicyTicket_In     *in              // IN: input parameter list
8      )
9  {
10     TPM_RC                     result;
11     SESSION                   *session;
12     UINT64                     authTimeout;
13     TPMT_TK_AUTH               ticketToCompare;
14     TPM_CC                     commandCode = TPM_CC_PolicySecret;
15     BOOL                       expiresOnReset;
16
17 // Input Validation
18
19     // Get pointer to the session structure
20     session = SessionGet(in->policySession);
21
22     // NOTE: A trial policy session is not allowed to use this command.
23     // A ticket is used in place of a previously given authorization. Since
24     // a trial policy doesn't actually authenticate, the validated
25     // ticket is not necessary and, in place of using a ticket, one
26     // should use the intended authorization for which the ticket
27     // would be a substitute.
28     if(session->attributes.isTrialPolicy)
29         return TPM_RCS_ATTRIBUTES + RC_PolicyTicket_policySession;
30     // Restore timeout data.  The format of timeout buffer is TPM-specific.
31     // In this implementation, the most significant bit of the timeout value is
32     // used as the flag to indicate that the ticket expires on TPM Reset or
33     // TPM Restart. The flag has to be removed before the parameters and ticket
34     // are checked.
35     if(in->timeout.t.size != sizeof(UINT64))
36         return TPM_RCS_SIZE + RC_PolicyTicket_timeout;
37     authTimeout = BYTE_ARRAY_TO_UINT64(in->timeout.t.buffer);
38
39     // extract the flag
40     expiresOnReset = (authTimeout & EXPIRATION_BIT) != 0;
41     authTimeout &= ~EXPIRATION_BIT;
42
43     // Do the normal checks on the cpHashA and timeout values
44     result = PolicyParameterChecks(session, authTimeout,
45                                    &in->cpHashA,
46                                    NULL,                    // no nonce
47                                    0,                       // no bad nonce return
48                                    RC_PolicyTicket_cpHashA,
49                                    RC_PolicyTicket_timeout);
50     if(result != TPM_RC_SUCCESS)
51         return result;
```

 Copyright © TCG 2006-2020

```
52      // Validate Ticket
53      // Re-generate policy ticket by input parameters
54      TicketComputeAuth(in->ticket.tag, in->ticket.hierarchy,
55                        authTimeout, expiresOnReset, &in->cpHashA, &in->policyRef,
56                        &in->authName, &ticketToCompare);
57      // Compare generated digest with input ticket digest
58      if(!MemoryEqual2B(&in->ticket.digest.b, &ticketToCompare.digest.b))
59          return TPM_RCS_TICKET + RC_PolicyTicket_ticket;
60
61  // Internal Data Update
62
63      // Is this ticket to take the place of a TPM2_PolicySigned() or
64      // a TPM2_PolicySecret()?
65      if(in->ticket.tag == TPM_ST_AUTH_SIGNED)
66          commandCode = TPM_CC_PolicySigned;
67      else if(in->ticket.tag == TPM_ST_AUTH_SECRET)
68          commandCode = TPM_CC_PolicySecret;
69      else
70          // There could only be two possible tag values.  Any other value should
71          // be caught by the ticket validation process.
72          FAIL(FATAL_ERROR_INTERNAL);
73
74      // Update policy context
75      PolicyContextUpdate(commandCode, &in->authName, &in->policyRef,
76                          &in->cpHashA, authTimeout, session);
77
78      return TPM_RC_SUCCESS;
79  }
80  #endif // CC_PolicyTicket
```

23.6 TPM2_PolicyOR

23.6.1 General Description

This command allows options in authorizations without requiring that the TPM evaluate all of the options. If a policy may be satisfied by different sets of conditions, the TPM need only evaluate one set that satisfies the policy. This command will indicate that one of the required sets of conditions has been satisfied.

PolicySession→policyDigest is compared against the list of provided values. If the current *policySession→policyDigest* does not match any value in the list, the TPM shall return TPM_RC_VALUE. Otherwise, the TPM will reset *policySession→policyDigest* to a Zero Digest. Then *policySession→policyDigest* is extended by the concatenation of TPM_CC_PolicyOR and the concatenation of all of the digests.

If *policySession* is a trial session, the TPM will assume that *policySession→policyDigest* matches one of the list entries and compute the new value of *policyDigest*.

The algorithm for computing the new value for *policyDigest* of *policySession* is:

a) Concatenate all the digest values in *pHashList*:

$$digests := pHashList.digests[1].buffer \mathbin{||} \ldots \mathbin{||} pHashList.digests[n].buffer \quad (17)$$

NOTE 1 The TPM will not return an error if the size of an entry is not the same as the size of the digest of the policy. However, that entry cannot match *policyDigest*.

b) Reset *policyDigest* to a Zero Digest.

c) Extend the command code and the hashes computed in step a) above:

$$policyDigest_{new} := \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyOR} \mathbin{||} digests) \quad (18)$$

NOTE 2 The computation in b) and c) above is equivalent to:

$$policyDigest_{new} := \mathbf{H}_{policyAlg}(0\ldots0 \mathbin{||} \text{TPM_CC_PolicyOR} \mathbin{||} digests)$$

A TPM shall support a list with at least eight tagged digest values.

NOTE 3 If policies are to be portable between TPMs, then they should not use more than eight values.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.6.2 Command and Response

Table 130 — TPM2_PolicyOR Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyOR
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPML_DIGEST	pHashList	the list of hashes to check for a match

Table 131 — TPM2_PolicyOR Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

23.6.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PolicyOR_fp.h"
3  #if CC_PolicyOR  // Conditional expansion of this file
4  #include "Policy_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_VALUE	no digest in *pHashList* matched the current value of *policyDigest* for *policySession*

```
5  TPM_RC
6  TPM2_PolicyOR(
7      PolicyOR_In     *in              // IN: input parameter list
8      )
9  {
10     SESSION     *session;
11     UINT32       i;
12
13 // Input Validation and Update
14
15     // Get pointer to the session structure
16     session = SessionGet(in->policySession);
17
18     // Compare and Update Internal Session policy if match
19     for(i = 0; i < in->pHashList.count; i++)
20     {
21         if(session->attributes.isTrialPolicy == SET
22            || (MemoryEqual2B(&session->u2.policyDigest.b,
23                              &in->pHashList.digests[i].b)))
24         {
25             // Found a match
26             HASH_STATE      hashState;
27             TPM_CC          commandCode = TPM_CC_PolicyOR;
28
29             // Start hash
30             session->u2.policyDigest.t.size
31                 = CryptHashStart(&hashState, session->authHashAlg);
32             // Set policyDigest to 0 string and add it to hash
33             MemorySet(session->u2.policyDigest.t.buffer, 0,
34                       session->u2.policyDigest.t.size);
35             CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
36
37             // add command code
38             CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
39
40             // Add each of the hashes in the list
41             for(i = 0; i < in->pHashList.count; i++)
42             {
43                 // Extend policyDigest
44                 CryptDigestUpdate2B(&hashState, &in->pHashList.digests[i].b);
45             }
46             // Complete digest
47             CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
48
49             return TPM_RC_SUCCESS;
50         }
51     }
52     // None of the values in the list matched the current policyDigest
53     return TPM_RCS_VALUE + RC_PolicyOR_pHashList;
54 }
55 #endif // CC_PolicyOR
```


 Copyright © TCG 2006-2020

23.7 TPM2_PolicyPCR

23.7.1 General Description

This command is used to cause conditional gating of a policy based on PCR. This command together with TPM2_PolicyOR() allows one group of authorizations to occur when PCR are in one state and a different set of authorizations when the PCR are in a different state.

The TPM will modify the *pcrs* parameter so that bits that correspond to unimplemented PCR are CLEAR. If *policySession* is not a trial policy session, the TPM will use the modified value of pcrs to select PCR values to hash according to TPM 2.0 Part 1, *Selecting Multiple PCR.* The hash algorithm of the policy session is used to compute a digest (*digestTPM*) of the selected PCR. If *pcrDigest* does not have a length of zero, then it is compared to *digestTPM*; and if the values do not match, the TPM shall return TPM_RC_VALUE and make no change to *policySession→policyDigest*. If the values match, or if the length of *pcrDigest* is zero, then *policySession→policyDigest* is extended by:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyPCR} \mathbin{||} pcrs \mathbin{||} digestTPM) \qquad (19)$$

where

$pcrs$	the pcrs parameter with bits corresponding to unimplemented PCR set to 0
$digestTPM$	the digest of the selected PCR using the hash algorithm of the policy session

NOTE 1 If the caller provides the expected PCR value, the intention is that the policy evaluation stop at that point if the PCR do not match. If the caller does not provide the expected PCR value, then the validity of the settings will not be determined until an attempt is made to use the policy for authorization. If the policy is constructed such that the PCR check comes before user authorization checks, this early termination would allow software to avoid unnecessary prompts for user input to satisfy a policy that would fail later due to incorrect PCR values.

After this command completes successfully, the TPM shall return TPM_RC_PCR_CHANGED if the policy session is used for authorization and the PCR are not known to be correct.

The TPM uses a "generation" number (*pcrUpdateCounter*) that is incremented each time PCR are updated (unless the PCR being changed is specified not to cause a change to this counter). The value of this counter is stored in the policy session context (*policySession→pcrUpdateCounter*) when this command is executed. When the policy is used for authorization, the current value of the counter is compared to the value in the policy session context and the authorization will fail if the values are not the same.

When this command is executed, *policySession→pcrUpdateCounter* is checked to see if it has been previously set (in the reference implementation, it has a value of zero if not previously set). If it has been set, it will be compared with the current value of *pcrUpdateCounter* to determine if any PCR changes have occurred. If the values are different, the TPM shall return TPM_RC_PCR_CHANGED.

NOTE 2 Since the pcrUpdateCounter is updated if any PCR is extended (except those specified not to do so), this means that the command will fail even if a PCR not specified in the policy is updated. This is an optimization for the purposes of conserving internal TPM memory. This would be a rare occurrence, and, if this should occur, the policy could be reset using the TPM2_PolicyRestart command and rerun.

If *policySession→pcrUpdateCounter* has not been set, then it is set to the current value of *pcrUpdateCounter*.

If this command is used for a trial *policySession*, *policySession→policyDigest* will be updated using the values from the command rather than the values from a digest of the TPM PCR. If the caller does not provide PCR settings (*pcrDigest* has a length of zero), the TPM may (and it is preferred to) use the current TPM PCR settings (*digestTPM)* in the calculation for the new *policyDigest*. The TPM may return

 Copyright © TCG 2006-2020

an error if the caller does not provide a PCR digest for a trial policy session but this is not the preferred behavior.

The TPM will not check any PCR and will compute:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyPCR} \mathbin{||} pcrs \mathbin{||} pcrDigest) \qquad (20)$$

In this computation, pcrs is the input parameter without modification.

NOTE 3 The pcrs parameter is expected to match the configuration of the TPM for which the policy is being computed which may not be the same as the TPM on which the trial policy is being computed.

NOTE 4 Although no PCR are checked in a trial policy session, *pcrDigest* is expected to correspond to some useful PCR values. It is legal, but pointless, to have the TPM aid in calculating a *policyDigest* corresponding to PCR values that are not useful in practice.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.7.2 Command and Response

Table 132 — TPM2_PolicyPCR Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyPCR
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_DIGEST	pcrDigest	expected digest value of the selected PCR using the hash algorithm of the session; may be zero length
TPML_PCR_SELECTION	pcrs	the PCR to include in the check digest

Table 133 — TPM2_PolicyPCR Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.7.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "PolicyPCR_fp.h"
3   #if CC_PolicyPCR  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_VALUE	if provided, *pcrDigest* does not match the current PCR settings
TPM_RC_PCR_CHANGED	a previous TPM2_PolicyPCR() set *pcrCounter* and it has changed

```
4   TPM_RC
5   TPM2_PolicyPCR(
6       PolicyPCR_In    *in             // IN: input parameter list
7       )
8   {
9       SESSION         *session;
10      TPM2B_DIGEST     pcrDigest;
11      BYTE             pcrs[sizeof(TPML_PCR_SELECTION)];
12      UINT32           pcrSize;
13      BYTE            *buffer;
14      TPM_CC           commandCode = TPM_CC_PolicyPCR;
15      HASH_STATE       hashState;
16
17  // Input Validation
18
19      // Get pointer to the session structure
20      session = SessionGet(in->policySession);
21
22      // Compute current PCR digest
23      PCRComputeCurrentDigest(session->authHashAlg, &in->pcrs, &pcrDigest);
24
25      // Do validation for non trial session
26      if(session->attributes.isTrialPolicy == CLEAR)
27      {
28          // Make sure that this is not going to invalidate a previous PCR check
29          if(session->pcrCounter != 0 && session->pcrCounter != gr.pcrCounter)
30              return TPM_RC_PCR_CHANGED;
31
32          // If the caller specified the PCR digest and it does not
33          // match the current PCR settings, return an error..
34          if(in->pcrDigest.t.size != 0)
35          {
36              if(!MemoryEqual2B(&in->pcrDigest.b, &pcrDigest.b))
37                  return TPM_RCS_VALUE + RC_PolicyPCR_pcrDigest;
38          }
39      }
40      else
41      {
42          // For trial session, just use the input PCR digest if one provided
43          // Note: It can't be too big because it is a TPM2B_DIGEST and the size
44          // would have been checked during unmarshaling
45          if(in->pcrDigest.t.size != 0)
46              pcrDigest = in->pcrDigest;
47      }
48  // Internal Data Update
49      // Update policy hash
50      // policyDigestnew = hash(   policyDigestold || TPM_CC_PolicyPCR
51      //                        || PCRS || pcrDigest)
52      //  Start hash
53      CryptHashStart(&hashState, session->authHashAlg);
54
```

 Copyright © TCG 2006-2020

```
55        //  add old digest
56        CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
57
58        //  add commandCode
59        CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
60
61        //  add PCRS
62        buffer = pcrs;
63        pcrSize = TPML_PCR_SELECTION_Marshal(&in->pcrs, &buffer, NULL);
64        CryptDigestUpdate(&hashState, pcrSize, pcrs);
65
66        //  add PCR digest
67        CryptDigestUpdate2B(&hashState, &pcrDigest.b);
68
69        //  complete the hash and get the results
70        CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
71
72        //  update pcrCounter in session context for non trial session
73        if(session->attributes.isTrialPolicy == CLEAR)
74        {
75            session->pcrCounter = gr.pcrCounter;
76        }
77
78        return TPM_RC_SUCCESS;
79    }
80    #endif // CC_PolicyPCR
```

 Copyright © TCG 2006-2020

23.8 TPM2_PolicyLocality

23.8.1 General Description

This command indicates that the authorization will be limited to a specific locality.

policySession→commandLocality is a parameter kept in the session context. When the policy session is started, this parameter is initialized to a value that allows the policy to apply to any locality.

If *locality* has a value greater than 31, then an extended locality is indicated. For an extended locality, the TPM will validate that *policySession→commandLocality* has not previously been set or that the current value of *policySession→commandLocality* is the same as *locality* (TPM_RC_RANGE).

When *locality* is not an extended locality, the TPM will validate that the *policySession→commandLocality* is not set to an extended locality value (TPM_RC_RANGE). If not the TPM will disable any locality not SET in the *locality* parameter. If the result of disabling localities results in no locality being enabled, the TPM will return TPM_RC_RANGE.

If no error occurred in the validation of *locality*, *policySession→policyDigest* is extended with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyLocality} \mathbin{||} locality) \quad (21)$$

Then *policySession→commandLocality* is updated to indicate which localities are still allowed after execution of TPM2_PolicyLocality().

When the policy session is used to authorize a command, the authorization will fail if the locality used for the command is not one of the enabled localities in *policySession→commandLocality*.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.8.2 Command and Response

Table 134 — TPM2_PolicyLocality Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyLocality
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPMA_LOCALITY	locality	the allowed localities for the policy

Table 135 — TPM2_PolicyLocality Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.8.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "PolicyLocality_fp.h"
3   #if CC_PolicyLocality  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_RANGE	all the locality values selected by *locality* have been disabled by previous TPM2_PolicyLocality() calls.

```
4   TPM_RC
5   TPM2_PolicyLocality(
6       PolicyLocality_In   *in             // IN: input parameter list
7       )
8   {
9       SESSION     *session;
10      BYTE         marshalBuffer[sizeof(TPMA_LOCALITY)];
11      BYTE         prevSetting[sizeof(TPMA_LOCALITY)];
12      UINT32       marshalSize;
13      BYTE        *buffer;
14      TPM_CC       commandCode = TPM_CC_PolicyLocality;
15      HASH_STATE   hashState;
16
17  // Input Validation
18
19      // Get pointer to the session structure
20      session = SessionGet(in->policySession);
21
22      // Get new locality setting in canonical form
23      marshalBuffer[0] = 0;   // Code analysis says that this is not initialized
24      buffer = marshalBuffer;
25      marshalSize = TPMA_LOCALITY_Marshal(&in->locality, &buffer, NULL);
26
27      // Its an error if the locality parameter is zero
28      if(marshalBuffer[0] == 0)
29          return TPM_RCS_RANGE + RC_PolicyLocality_locality;
30
31      // Get existing locality setting in canonical form
32      prevSetting[0] = 0;     // Code analysis says that this is not initialized
33      buffer = prevSetting;
34      TPMA_LOCALITY_Marshal(&session->commandLocality, &buffer, NULL);
35
36      // If the locality has previously been set
37      if(prevSetting[0] != 0
38          // then the current locality setting and the requested have to be the same
39          // type (that is, either both normal or both extended
40          && ((prevSetting[0] < 32) != (marshalBuffer[0] < 32)))
41          return TPM_RCS_RANGE + RC_PolicyLocality_locality;
42
43      // See if the input is a regular or extended locality
44      if(marshalBuffer[0] < 32)
45      {
46          // if there was no previous setting, start with all normal localities
47          // enabled
48          if(prevSetting[0] == 0)
49              prevSetting[0] = 0x1F;
50
51          // AND the new setting with the previous setting and store it in prevSetting
52          prevSetting[0] &= marshalBuffer[0];
53
54          // The result setting can not be 0
55          if(prevSetting[0] == 0)
```

 Copyright © TCG 2006-2020

```
56              return TPM_RCS_RANGE + RC_PolicyLocality_locality;
57      }
58      else
59      {
60          // for extended locality
61          // if the locality has already been set, then it must match the
62          if(prevSetting[0] != 0 && prevSetting[0] != marshalBuffer[0])
63              return TPM_RCS_RANGE + RC_PolicyLocality_locality;
64
65          // Setting is OK
66          prevSetting[0] = marshalBuffer[0];
67      }
68
69  // Internal Data Update
70
71      // Update policy hash
72      // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyLocality || locality)
73      // Start hash
74      CryptHashStart(&hashState, session->authHashAlg);
75
76      // add old digest
77      CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
78
79      // add commandCode
80      CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
81
82      // add input locality
83      CryptDigestUpdate(&hashState, marshalSize, marshalBuffer);
84
85      // complete the digest
86      CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
87
88      // update session locality by unmarshal function.  The function must succeed
89      // because both input and existing locality setting have been validated.
90      buffer = prevSetting;
91      TPMA_LOCALITY_Unmarshal(&session->commandLocality, &buffer,
92                              (INT32 *)&marshalSize);
93
94      return TPM_RC_SUCCESS;
95  }
96  #endif // CC_PolicyLocality
```

23.9 TPM2_PolicyNV

23.9.1 General Description

This command is used to cause conditional gating of a policy based on the contents of an NV Index. It is an immediate assertion. The NV index is validated during the TPM2_PolicyNV() command, not when the session is used for authorization.

The authorization to read the NV Index must succeed even if *policySession* is a trial policy session.

If *policySession* is a trial policy session, the TPM will update *policySession→policyDigest* as shown in equations (22) and (23) below and return TPM_RC_SUCCESS. It will not perform any further validation. The remainder of this general description would apply only if *policySession* is not a trial policy session.

An authorization session providing authorization to read the NV Index shall be provided.

If TPMA_NV_WRITTEN is not SET in the NV Index, the TPM shall return TPM_RC_NV_UNINITIALIZED. If TPMA_NV_READLOCKED of the NV Index is SET, then the TPM shall return TPM_RC_NV_LOCKED.

For an NV Index with the TPM_NT_COUNTER or TPM_NT_BITS attribute SET, the TPM may ignore the *offset* parameter and use an offset of 0. Therefore, it is recommended that the caller set the *offset* parameter to 0 for interoperability.

If *offset* and the *size* field of *data* add to a value that is greater than the *dataSize* field of the NV Index referenced by *nvIndex*, the TPM shall return an error (TPM_RC_NV_RANGE). The implementation may return an error (TPM_RC_VALUE) if it performs an additional check and determines that *offset* is greater than the *dataSize* field of the NV Index.

operandA begins at *offset* into the NV index contents and has a size equal to the size of *operandB*. The TPM will perform the indicated arithmetic check using *operandA* and *operandB*. If the check fails, the TPM shall return TPM_RC_POLICY and not change *policySession→policyDigest*. If the check succeeds, the TPM will hash the arguments:

$$args \coloneqq \mathbf{H}_{policyAlg}(operandB.buffer \mathbin{||} offset \mathbin{||} operation) \tag{22}$$

where

$\mathbf{H}_{policyAlg}()$	hash function using the algorithm of the policy session
$operandB$	the value used for the comparison
$offset$	offset from the start of the NV Index data to start the comparison
$operation$	the operation parameter indicating the comparison being performed

The value of args and the Name of the NV Index are extended to *policySession→policyDigest* by

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyNV} \mathbin{||} args \mathbin{||} nvIndex{\rightarrow}Name) \tag{23}$$

where

$\mathbf{H}_{policyAlg}()$	hash function using the algorithm of the policy session
$args$	value computed in equation (22)
$nvIndex{\rightarrow}Name$	the Name of the NV Index

The signed arithmetic operations are performed using twos-compliment.

Magnitude comparisons assume that the octet at offset zero in the referenced NV location and in *operandB* contain the most significant octet of the data.

 Copyright © TCG 2006-2020

23.9.2 Command and Response

Table 136 — TPM2_PolicyNV Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyNV
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index of the area to read Auth Index: None
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_OPERAND	operandB	the second operand
UINT16	offset	the octet offset in the NV Index for the start of operand A
TPM_EO	operation	the comparison to make

Table 137 — TPM2_PolicyNV Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.9.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PolicyNV_fp.h"
3  #if CC_PolicyNV  // Conditional expansion of this file
4  #include "Policy_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_AUTH_TYPE	NV index authorization type is not correct
TPM_RC_NV_LOCKED	NV index read locked
TPM_RC_NV_UNINITIALIZED	the NV index has not been initialized
TPM_RC_POLICY	the comparison to the NV contents failed
TPM_RC_SIZE	the size of *nvIndex* data starting at *offset* is less than the size of *operandB*
TPM_RC_VALUE	*offset* is too large

```
5  TPM_RC
6  TPM2_PolicyNV(
7      PolicyNV_In     *in             // IN: input parameter list
8      )
9  {
10     TPM_RC                result;
11     SESSION              *session;
12     NV_REF                locator;
13     NV_INDEX             *nvIndex;
14     BYTE                  nvBuffer[sizeof(in->operandB.t.buffer)];
15     TPM2B_NAME            nvName;
16     TPM_CC                commandCode = TPM_CC_PolicyNV;
17     HASH_STATE            hashState;
18     TPM2B_DIGEST          argHash;
19
20 // Input Validation
21
22     // Get pointer to the session structure
23     session = SessionGet(in->policySession);
24
25     //If this is a trial policy, skip all validations and the operation
26     if(session->attributes.isTrialPolicy == CLEAR)
27     {
28         // No need to access the actual NV index information for a trial policy.
29         nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
30
31         // Common read access checks. NvReadAccessChecks() may return
32         // TPM_RC_NV_AUTHORIZATION, TPM_RC_NV_LOCKED, or TPM_RC_NV_UNINITIALIZED
33         result = NvReadAccessChecks(in->authHandle,
34                                     in->nvIndex,
35                                     nvIndex->publicArea.attributes);
36         if(result != TPM_RC_SUCCESS)
37             return result;
38
39         // Make sure that offset is withing range
40         if(in->offset > nvIndex->publicArea.dataSize)
41             return TPM_RCS_VALUE + RC_PolicyNV_offset;
42
43         // Valid NV data size should not be smaller than input operandB size
44         if((nvIndex->publicArea.dataSize - in->offset) < in->operandB.t.size)
45             return TPM_RCS_SIZE + RC_PolicyNV_operandB;
46
```

 Copyright © TCG 2006-2020

```
47          // Get NV data.  The size of NV data equals the input operand B size
48          NvGetIndexData(nvIndex, locator, in->offset, in->operandB.t.size, nvBuffer);
49
50          // Check to see if the condition is valid
51          if(!PolicySptCheckCondition(in->operation, nvBuffer,
52                                      in->operandB.t.buffer, in->operandB.t.size))
53              return TPM_RC_POLICY;
54      }
55  // Internal Data Update
56
57      // Start argument hash
58      argHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
59
60      //  add operandB
61      CryptDigestUpdate2B(&hashState, &in->operandB.b);
62
63      //  add offset
64      CryptDigestUpdateInt(&hashState, sizeof(UINT16), in->offset);
65
66      //  add operation
67      CryptDigestUpdateInt(&hashState, sizeof(TPM_EO), in->operation);
68
69      //  complete argument digest
70      CryptHashEnd2B(&hashState, &argHash.b);
71
72      // Update policyDigest
73      //  Start digest
74      CryptHashStart(&hashState, session->authHashAlg);
75
76      //  add old digest
77      CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
78
79      //  add commandCode
80      CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
81
82      //  add argument digest
83      CryptDigestUpdate2B(&hashState, &argHash.b);
84
85      // Adding nvName
86      CryptDigestUpdate2B(&hashState, &EntityGetName(in->nvIndex, &nvName)->b);
87
88      // complete the digest
89      CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
90
91      return TPM_RC_SUCCESS;
92  }
93  #endif // CC_PolicyNV
```

 Copyright © TCG 2006-2020

23.10 TPM2_PolicyCounterTimer

23.10.1 General Description

This command is used to cause conditional gating of a policy based on the contents of the TPMS_TIME_INFO structure.

If *policySession* is a trial policy session, the TPM will update *policySession→policyDigest* as shown in equations (24) and (25) below and return TPM_RC_SUCCESS. It will not perform any validation. The remainder of this general description would apply only if *policySession* is not a trial policy session.

The TPM will perform the indicated arithmetic check on the indicated portion of the TPMS_TIME_INFO structure. If the check fails, the TPM shall return TPM_RC_POLICY and not change *policySession→policyDigest*. If the check succeeds, the TPM will hash the arguments:

$$args \coloneqq \mathbf{H}_{policyAlg}(operandB.buffer \mathbin{||} offset \mathbin{||} operation) \tag{24}$$

where

$\mathbf{H}_{policyAlg}()$	hash function using the algorithm of the policy session
$operandB.buffer$	the value used for the comparison
$offset$	offset from the start of the TPMS_TIME_INFO structure at which the comparison starts
$operation$	the operation parameter indicating the comparison being performed

NOTE There is no security related reason for the double hash.

The value of *args* is extended to *policySession→policyDigest* by

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyCounterTimer} \mathbin{||} args) \tag{25}$$

where

$\mathbf{H}_{policyAlg}()$	hash function using the algorithm of the policy session
$args$	value computed in equation (24)

The signed arithmetic operations are performed using twos-compliment. The indicated portion of the TPMS_TIME_INFO structure begins at *offset* and has a length of *operandB.size*. If the number of octets to be compared overflows the TPMS_TIME_INFO structure, the TPM returns TPM_RC_RANGE. If *offset* is greater than the size of the marshaled TPMS_TIME_INFO structure, the TPM returns TPM_RC_VALUE. The structure is marshaled into its canonical form with no padding. The TPM does not check for alignment of the offset with a TPMS_TIME_INFO structure member.

Magnitude comparisons assume that the octet at offset zero in the referenced location and in *operandB* contain the most significant octet of the data.

 Copyright © TCG 2006-2020

23.10.2 Command and Response

Table 138 — TPM2_PolicyCounterTimer Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyCounterTimer
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_OPERAND	operandB	the second operand
UINT16	offset	the octet offset in the TPMS_TIME_INFO structure for the start of operand A
TPM_EO	operation	the comparison to make

Table 139 — TPM2_PolicyCounterTimer Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

23.10.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "PolicyCounterTimer_fp.h"
3   #if CC_PolicyCounterTimer  // Conditional expansion of this file
4   #include "Policy_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_POLICY	the comparison of the selected portion of the TPMS_TIME_INFO with *operandB* failed
TPM_RC_RANGE	*offset* + *size* exceed size of TPMS_TIME_INFO structure

```
5   TPM_RC
6   TPM2_PolicyCounterTimer(
7       PolicyCounterTimer_In   *in             // IN: input parameter list
8       )
9   {
10      SESSION                 *session;
11      TIME_INFO                infoData;          // data buffer of  TPMS_TIME_INFO
12      BYTE                    *pInfoData = (BYTE *)&infoData;
13      UINT16                   infoDataSize;
14      TPM_CC                   commandCode = TPM_CC_PolicyCounterTimer;
15      HASH_STATE               hashState;
16      TPM2B_DIGEST             argHash;
17
18  // Input Validation
19      // Get a marshaled time structure
20      infoDataSize = TimeGetMarshaled(&infoData);
21      // Make sure that the referenced stays within the bounds of the structure.
22      // NOTE: the offset checks are made even for a trial policy because the policy
23      // will not make any sense if the references are out of bounds of the timer
24      // structure.
25      if(in->offset > infoDataSize)
26          return TPM_RCS_VALUE + RC_PolicyCounterTimer_offset;
27      if((UINT32)in->offset + (UINT32)in->operandB.t.size > infoDataSize)
28          return TPM_RCS_RANGE;
29      // Get pointer to the session structure
30      session = SessionGet(in->policySession);
31
32      //If this is a trial policy, skip the check to see if the condition is met.
33      if(session->attributes.isTrialPolicy == CLEAR)
34      {
35          // If the command is going to use any part of the counter or timer, need
36          // to verify that time is advancing.
37          // The time and clock vales are the first two 64-bit values in the clock
38          if(in->offset < sizeof(UINT64) + sizeof(UINT64))
39          {
40              // Using Clock or Time so see if clock is running. Clock doesn't
41              // run while NV is unavailable.
42              // TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned here.
43              RETURN_IF_NV_IS_NOT_AVAILABLE;
44          }
45          // offset to the starting position
46          pInfoData = (BYTE *)infoData;
47          // Check to see if the condition is valid
48          if(!PolicySptCheckCondition(in->operation, pInfoData + in->offset,
49                                      in->operandB.t.buffer, in->operandB.t.size))
50              return TPM_RC_POLICY;
51      }
52  // Internal Data Update
53      // Start argument list hash
```

 Copyright © TCG 2006-2020

```
54      argHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
55      //  add operandB
56      CryptDigestUpdate2B(&hashState, &in->operandB.b);
57      //  add offset
58      CryptDigestUpdateInt(&hashState, sizeof(UINT16), in->offset);
59      //  add operation
60      CryptDigestUpdateInt(&hashState, sizeof(TPM_EO), in->operation);
61      //  complete argument hash
62      CryptHashEnd2B(&hashState, &argHash.b);
63
64      // update policyDigest
65      //  start hash
66      CryptHashStart(&hashState, session->authHashAlg);
67
68      //  add old digest
69      CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
70
71      //  add commandCode
72      CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
73
74      //  add argument digest
75      CryptDigestUpdate2B(&hashState, &argHash.b);
76
77      // complete the digest
78      CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
79
80      return TPM_RC_SUCCESS;
81  }
82  #endif // CC_PolicyCounterTimer
```

 Copyright © TCG 2006-2020

23.11 TPM2_PolicyCommandCode

23.11.1 General Description

This command indicates that the authorization will be limited to a specific command code.

If *policySession→commandCode* has its default value, then it will be set to *code.* If *policySession→commandCode* does not have its default value, then the TPM will return TPM_RC_VALUE if the two values are not the same.

If *code* is not implemented, the TPM will return TPM_RC_POLICY_CC.

If the TPM does not return an error, it will update *policySession→policyDigest* by

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM_CC_PolicyCommandCode} \,||\, code) \quad (26)$$

NOTE 1 If a previous TPM2_PolicyCommandCode() had been executed, then it is probable that the policy expression is improperly formed but the TPM does not return an error if *code* is the same.

NOTE 2 A TPM2_PolicyOR() would be used to allow an authorization to be used for multiple commands.

When the policy session is used to authorize a command, the TPM will fail the command if the *commandCode* of that command does not match *policySession→commandCode*.

This command, or TPM2_PolicyDuplicationSelect(), is required to enable the policy to be used for ADMIN role authorization.

EXAMPLE Before TPM2_Certify() can be executed, TPM2_PolicyCommandCode() with code set to TPM_CC_Certify is required.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.11.2 Command and Response

Table 140 — TPM2_PolicyCommandCode Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyCommandCode
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM_CC	code	the allowed *commandCode*

Table 141 — TPM2_PolicyCommandCode Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

23.11.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "PolicyCommandCode_fp.h"
3   #if CC_PolicyCommandCode  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_VALUE	*commandCode* of *policySession* previously set to a different value

```
4   TPM_RC
5   TPM2_PolicyCommandCode(
6       PolicyCommandCode_In    *in              // IN: input parameter list
7       )
8   {
9       SESSION     *session;
10      TPM_CC      commandCode = TPM_CC_PolicyCommandCode;
11      HASH_STATE  hashState;
12
13  // Input validation
14
15      // Get pointer to the session structure
16      session = SessionGet(in->policySession);
17
18      if(session->commandCode != 0 && session->commandCode != in->code)
19              return TPM_RCS_VALUE + RC_PolicyCommandCode_code;
20      if(CommandCodeToCommandIndex(in->code) == UNIMPLEMENTED_COMMAND_INDEX)
21          return TPM_RCS_POLICY_CC + RC_PolicyCommandCode_code;
22
23  // Internal Data Update
24      // Update policy hash
25      // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyCommandCode || code)
26      //  Start hash
27      CryptHashStart(&hashState, session->authHashAlg);
28
29      //  add old digest
30      CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
31
32      //  add commandCode
33      CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
34
35      //  add input commandCode
36      CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), in->code);
37
38      //  complete the hash and get the results
39      CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
40
41      // update commandCode value in session context
42      session->commandCode = in->code;
43
44      return TPM_RC_SUCCESS;
45  }
46  #endif // CC_PolicyCommandCode
```

 Copyright © TCG 2006-2020

23.12 TPM2_PolicyPhysicalPresence

23.12.1 General Description

This command indicates that physical presence will need to be asserted at the time the authorization is performed.

If this command is successful, *policySession→isPPRequired* will be SET to indicate that this check is required when the policy is used for authorization. Additionally, *policySession→policyDigest* is extended with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyPhysicalPresence}) \qquad (27)$$

 Copyright © TCG 2006-2020

23.12.2 Command and Response

Table 142 — TPM2_PolicyPhysicalPresence Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyPhysicalPresence
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None

Table 143 — TPM2_PolicyPhysicalPresence Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.12.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PolicyPhysicalPresence_fp.h"
3  #if CC_PolicyPhysicalPresence  // Conditional expansion of this file
4  TPM_RC
5  TPM2_PolicyPhysicalPresence(
6      PolicyPhysicalPresence_In   *in            // IN: input parameter list
7      )
8  {
9      SESSION     *session;
10     TPM_CC      commandCode = TPM_CC_PolicyPhysicalPresence;
11     HASH_STATE  hashState;
12
13 // Internal Data Update
14
15     // Get pointer to the session structure
16     session = SessionGet(in->policySession);
17
18     // Update policy hash
19     // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyPhysicalPresence)
20     //  Start hash
21     CryptHashStart(&hashState, session->authHashAlg);
22
23     //  add old digest
24     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
25
26     //  add commandCode
27     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
28
29     //  complete the digest
30     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
31
32     // update session attribute
33     session->attributes.isPPRequired = SET;
34
35     return TPM_RC_SUCCESS;
36 }
37 #endif // CC_PolicyPhysicalPresence
```

23.13 TPM2_PolicyCpHash

23.13.1 General Description

This command is used to allow a policy to be bound to a specific command and command parameters.

TPM2_PolicySigned(), TPM2_PolicySecret(), and TPM2_PolicyTIcket() are designed to allow an authorizing entity to execute an arbitrary command as the *cpHashA* parameter of those commands is not included in *policySession→policyDigest*. TPM2_PolicyCommandCode() allows the policy to be bound to a specific Command Code so that only certain entities may authorize specific command codes. This command allows the policy to be restricted such that an entity may only authorize a command with a specific set of parameters.

If *policySession→cpHash* is already set and not the same as *cpHashA*, then the TPM shall return TPM_RC_CPHASH. If *cpHashA* does not have the size of the *policySession→policyDigest*, the TPM shall return TPM_RC_SIZE.

NOTE 1 If a previous TPM2_PolicyCpHash() had been executed, then it is probable that the policy expression is improperly formed but the TPM does not return an error if *cpHash* is the same.

If the *cpHashA* checks succeed, *policySession→cpHash* is set to *cpHashA* and *policySession→policyDigest* is updated with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM_CC_PolicyCpHash} \,||\, cpHashA) \quad (28)$$

 Copyright © TCG 2006-2020

23.13.2 Command and Response

Table 144 — TPM2_PolicyCpHash Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyCpHash
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_DIGEST	cpHashA	the *cpHash* added to the policy

Table 145 — TPM2_PolicyCpHash Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

23.13.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PolicyCpHash_fp.h"
3  #if CC_PolicyCpHash  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_CPHASH	*cpHash* of *policySession* has previously been set to a different value
TPM_RC_SIZE	*cpHashA* is not the size of a digest produced by the hash algorithm associated with *policySession*

```
4  TPM_RC
5  TPM2_PolicyCpHash(
6      PolicyCpHash_In     *in             // IN: input parameter list
7      )
8  {
9      SESSION     *session;
10     TPM_CC      commandCode = TPM_CC_PolicyCpHash;
11     HASH_STATE  hashState;
12
13 // Input Validation
14
15     // Get pointer to the session structure
16     session = SessionGet(in->policySession);
17
18     // A valid cpHash must have the same size as session hash digest
19     // NOTE: the size of the digest can't be zero because TPM_ALG_NULL
20     // can't be used for the authHashAlg.
21     if(in->cpHashA.t.size != CryptHashGetDigestSize(session->authHashAlg))
22         return TPM_RCS_SIZE + RC_PolicyCpHash_cpHashA;
23
24     // error if the cpHash in session context is not empty and is not the same
25     // as the input or is not a cpHash
26     if((session->u1.cpHash.t.size != 0)
27        && (!session->attributes.isCpHashDefined
28            || !MemoryEqual2B(&in->cpHashA.b, &session->u1.cpHash.b)))
29         return TPM_RC_CPHASH;
30
31 // Internal Data Update
32
33     // Update policy hash
34     // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyCpHash || cpHashA)
35     //  Start hash
36     CryptHashStart(&hashState, session->authHashAlg);
37
38     //  add old digest
39     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
40
41     //  add commandCode
42     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
43
44     //  add cpHashA
45     CryptDigestUpdate2B(&hashState, &in->cpHashA.b);
46
47     //  complete the digest and get the results
48     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
49
50     // update cpHash in session context
51     session->u1.cpHash = in->cpHashA;
52     session->attributes.isCpHashDefined = SET;
53
```

 Copyright © TCG 2006-2020

```
54      return TPM_RC_SUCCESS;
55  }
56  #endif // CC_PolicyCpHash
```

23.14 TPM2_PolicyNameHash

23.14.1 General Description

This command allows a policy to be bound to a specific set of TPM entities without being bound to the parameters of the command. This is most useful for commands such as TPM2_Duplicate() and for TPM2_PCR_Event() when the referenced PCR requires a policy.

The *nameHash* parameter should contain the digest of the Names associated with the handles to be used in the authorized command.

EXAMPLE For the TPM2_Duplicate() command, two handles are provided. One is the handle of the object being duplicated and the other is the handle of the new parent. For that command, *nameHash* would contain:

$$nameHash \coloneqq \mathbf{H}_{policyAlg}(objectHandle{\rightarrow}Name \mid\mid newParentHandle{\rightarrow}Name)$$

If *policySession→cpHash* is already set, the TPM shall return TPM_RC_CPHASH. If the size of *nameHash* is not the size of *policySession→policyDigest*, the TPM shall return TPM_RC_SIZE. Otherwise, *policySession→cpHash* is set to *nameHash*.

If this command completes successfully, the *cpHash* of the authorized command will not be used for validation. Only the digest of the Names associated with the handles in the command will be used.

NOTE 1 This allows the space normally used to hold *policySession→cpHash* to be used for *policySession→nameHash* instead.

The *policySession→policyDigest* will be updated with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mid\mid \text{TPM_CC_PolicyNameHash} \mid\mid nameHash) \quad (29)$$

NOTE 2 This command can only be used with TPM2_PolicyAuthorize() or TPM2_PolicyAuthorizeNV. The owner of the object being duplicated provides approval for their object to be migrated to a specific new parent.

Without this approval, the Name of the Object would need to be known at the time that Object's policy is created. However, since the Name of the Object includes its policy, the Name is not known. The Name can be known by the authorizing entity.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.14.2 Command and Response

Table 146 — TPM2_PolicyNameHash Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyNameHash
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_DIGEST	nameHash	the digest to be added to the policy

Table 147 — TPM2_PolicyNameHash Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

23.14.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PolicyNameHash_fp.h"
3  #if CC_PolicyNameHash  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_CPHASH	*nameHash* has been previously set to a different value
TPM_RC_SIZE	*nameHash* is not the size of the digest produced by the hash algorithm associated with *policySession*

```
4  TPM_RC
5  TPM2_PolicyNameHash(
6      PolicyNameHash_In   *in             // IN: input parameter list
7      )
8  {
9      SESSION             *session;
10     TPM_CC               commandCode = TPM_CC_PolicyNameHash;
11     HASH_STATE           hashState;
12
13 // Input Validation
14
15     // Get pointer to the session structure
16     session = SessionGet(in->policySession);
17
18     // A valid nameHash must have the same size as session hash digest
19     // Since the authHashAlg for a session cannot be TPM_ALG_NULL, the digest size
20     // is always non-zero.
21     if(in->nameHash.t.size != CryptHashGetDigestSize(session->authHashAlg))
22         return TPM_RCS_SIZE + RC_PolicyNameHash_nameHash;
23
24     // u1 in the policy session context cannot otherwise be occupied
25     if(session->u1.cpHash.b.size != 0
26        || session->attributes.isBound
27        || session->attributes.isCpHashDefined
28        || session->attributes.isTemplateSet)
29         return TPM_RC_CPHASH;
30
31 // Internal Data Update
32
33     // Update policy hash
34     // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyNameHash || nameHash)
35     //  Start hash
36     CryptHashStart(&hashState, session->authHashAlg);
37
38     //  add old digest
39     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
40
41     //  add commandCode
42     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
43
44     //  add nameHash
45     CryptDigestUpdate2B(&hashState, &in->nameHash.b);
46
47     //  complete the digest
48     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
49
50     // update nameHash in session context
51     session->u1.cpHash = in->nameHash;
52
53     return TPM_RC_SUCCESS;
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

```
54  }
55  #endif // CC_PolicyNameHash
```

23.15 TPM2_PolicyDuplicationSelect

23.15.1 General Description

This command allows qualification of duplication to allow duplication to a selected new parent.

If this command not used in conjunction with a PolicyAuthorize Command, then only the new parent is selected and *includeObject* should be CLEAR.

EXAMPLE When an object is created when the list of allowed duplication targets is known, the policy would be created with *includeObject* CLEAR.

NOTE 1 Only the new parent may be selected because, without TPM2_PolicyAuthorize(), the Name of the Object to be duplicated would need to be known at the time that Object's policy is created. However, since the Name of the Object includes its policy, the Name is not known. The Name can be known by the authorizing entity (a PolicyAuthorize Command) in which case *includeObject* may be SET.

If used in conjunction with TPM2_PolicyAuthorize(), then the authorizer of the new policy has the option of selecting just the new parent or of selecting both the new parent and the duplication Object.

NOTE 2 If the authorizing entity for an TPM2_PolicyAuthorize() only specifies the new parent, then that authorization may be applied to the duplication of any number of other Objects. If the authorizing entity specifies both a new parent and the duplicated Object, then the authorization only applies to that pairing of Object and new parent.

If either *policySession→cpHash* or *policySession→nameHash* has been previously set, the TPM shall return TPM_RC_CPHASH. Otherwise, *policySession→nameHash* will be set to:

$$nameHash \coloneqq \mathbf{H}_{policyAlg}(objectName.name \mid\mid newParentName.name) \quad (30)$$

NOTE 3 It is allowed that policySesion→nameHash and policySession→cpHash share the same memory space.

NOTE 4 The Name in these equations uses Name.name, indicating that the UINT16 size is not included in the hash.

The *policySession→policyDigest* will be updated according to the setting of *includeObject*. If equal to YES, *policySession→policyDigest* is updated by:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mid\mid \text{TPM_CC_PolicyDuplicationSelect} \mid\mid objectName.name \mid\mid newParentName.name \mid\mid includeObject) \quad (31)$$

If includeObject is NO, *policySession→policyDigest* is updated by:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mid\mid \text{TPM_CC_PolicyDuplicationSelect} \mid\mid newParentName.name \mid\mid includeObject) \quad (32)$$

NOTE 5 *policySession→nameHash* receives the digest of both Names so that the check performed in TPM2_Duplicate() may be the same regardless of which Names are included in *policySession→policyDigest*. This means that, when TPM2_PolicyDuplicationSelect() is executed, it is only valid for a specific pair of duplication object and new parent.

If the command succeeds, *policySession→commandCode* is set to TPM_CC_Duplicate.

NOTE 6 The normal use of this command is before a TPM2_PolicyAuthorize(). An authorized entity would approve a *policyDigest* that allowed duplication to a specific new parent. The authorizing entity may want to limit the authorization so that the approval allows only a specific object to be duplicated to the new parent. In that case, the authorizing entity would approve the *policyDigest* of *equation* (31).

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.15.2 Command and Response

Table 148 — TPM2_PolicyDuplicationSelect Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyDuplicationSelect
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_NAME	objectName	the Name of the object to be duplicated
TPM2B_NAME	newParentName	the Name of the new parent
TPMI_YES_NO	includeObject	if YES, the *objectName* will be included in the value in *policySession→policyDigest*

Table 149 — TPM2_PolicyDuplicationSelect Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

23.15.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PolicyDuplicationSelect_fp.h"
3  #if CC_PolicyDuplicationSelect  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_COMMAND_CODE	*commandCode* of '*policySession*; is not empty
TPM_RC_CPHASH	*cpHash* of *policySession* is not empty

```
4  TPM_RC
5  TPM2_PolicyDuplicationSelect(
6      PolicyDuplicationSelect_In  *in              // IN: input parameter list
7      )
8  {
9      SESSION         *session;
10     HASH_STATE      hashState;
11     TPM_CC          commandCode = TPM_CC_PolicyDuplicationSelect;
12
13 // Input Validation
14
15     // Get pointer to the session structure
16     session = SessionGet(in->policySession);
17
18     // cpHash in session context must be empty
19     if(session->u1.cpHash.t.size != 0)
20         return TPM_RC_CPHASH;
21
22     // commandCode in session context must be empty
23     if(session->commandCode != 0)
24         return TPM_RC_COMMAND_CODE;
25
26 // Internal Data Update
27
28     // Update name hash
29     session->u1.cpHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
30
31     //  add objectName
32     CryptDigestUpdate2B(&hashState, &in->objectName.b);
33
34     //  add new parent name
35     CryptDigestUpdate2B(&hashState, &in->newParentName.b);
36
37     //  complete hash
38     CryptHashEnd2B(&hashState, &session->u1.cpHash.b);
39
40     // update policy hash
41     // Old policyDigest size should be the same as the new policyDigest size since
42     // they are using the same hash algorithm
43     session->u2.policyDigest.t.size
44         = CryptHashStart(&hashState, session->authHashAlg);
45 //  add old policy
46     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
47
48     //  add command code
49     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
50
51     //  add objectName
52     if(in->includeObject == YES)
53         CryptDigestUpdate2B(&hashState, &in->objectName.b);
54
```

 Copyright © TCG 2006-2020

```
55      //  add new parent name
56      CryptDigestUpdate2B(&hashState, &in->newParentName.b);
57
58      //  add includeObject
59      CryptDigestUpdateInt(&hashState, sizeof(TPMI_YES_NO), in->includeObject);
60
61      //  complete digest
62      CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
63
64      // set commandCode in session context
65      session->commandCode = TPM_CC_Duplicate;
66
67      return TPM_RC_SUCCESS;
68  }
69  #endif // CC_PolicyDuplicationSelect
```

 Copyright © TCG 2006-2020

23.16 TPM2_PolicyAuthorize

23.16.1 General Description

This command allows policies to change. If a policy were static, then it would be difficult to add users to a policy. This command lets a policy authority sign a new policy so that it may be used in an existing policy.

The authorizing entity signs a structure that contains

$$aHash \coloneqq \mathbf{H}_{aHashAlg}(approvedPolicy \,||\, policyRef) \quad (33)$$

The *aHashAlg* is required to be the *nameAlg* of the key used to sign the *aHash*. The *aHash* value is then signed (symmetric or asymmetric) by *keySign*. That signature is then checked by the TPM in 20.1 TPM2_VerifySignature() which produces a ticket by

$$\mathbf{HMAC}(proof, (\text{TPM_ST_VERIFIED} \,||\, aHash \,||\, keySign \rightarrow Name)) \quad (34)$$

NOTE 1 The reason for the validation is because of the expectation that the policy will be used multiple times and it is more efficient to check a ticket than to load an object each time to check a signature.

The ticket is then used in TPM2_PolicyAuthorize() to validate the parameters.

The *keySign* parameter is required to be a valid object name using nameAlg other than TPM_ALG_NULL. If the first two octets of *keySign* are not a valid hash algorithm, the TPM shall return TPM_RC_HASH. If the remainder of the Name is not the size of the indicated digest, the TPM shall return TPM_RC_SIZE.

The TPM validates that the *approvedPolicy* matches the current value of *policySession→policyDigest* and if not, shall return TPM_RC_VALUE.

The TPM then validates that the parameters to TPM2_PolicyAuthorize() match the values used to generate the ticket. If so, the TPM will reset *policySession→policyDigest* to a Zero Digest. Then it will update *policySession→policyDigest* with **PolicyUpdate**() (see 23.2.3).

$$\mathbf{PolicyUpdate}(\text{TPM_CC_PolicyAuthorize}, keySign, policyRef) \quad (35)$$

If the ticket is not valid, the TPM shall return TPM_RC_POLICY.

If *policySession* is a trial session, *policySession→policyDigest* is extended as if the ticket is valid without actual verification.

NOTE 2 The unmarshaling process requires that a proper TPMT_TK_VERIFIED be provided for *checkTicket* but it may be a NULL Ticket. A NULL ticket is useful in a trial policy, where the caller uses the TPM to perform policy calculations but does not have a valid authorization ticket.

 Copyright © TCG 2006-2020

23.16.2 Command and Response

Table 150 — TPM2_PolicyAuthorize Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyAuthorize
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_DIGEST	approvedPolicy	digest of the policy being approved
TPM2B_NONCE	policyRef	a policy qualifier
TPM2B_NAME	keySign	Name of a key that can sign a policy addition
TPMT_TK_VERIFIED	checkTicket	ticket validating that *approvedPolicy* and *policyRef* were signed by *keySign*

Table 151 — TPM2_PolicyAuthorize Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.16.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PolicyAuthorize_fp.h"
3  #if CC_PolicyAuthorize  // Conditional expansion of this file
4  #include "Policy_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_HASH	hash algorithm in *keyName* is not supported
TPM_RC_SIZE	*keyName* is not the correct size for its hash algorithm
TPM_RC_VALUE	the current *policyDigest* of *policySession* does not match *approvedPolicy*; or *checkTicket* doesn't match the provided values

```
5  TPM_RC
6  TPM2_PolicyAuthorize(
7      PolicyAuthorize_In  *in            // IN: input parameter list
8      )
9  {
10     SESSION                  *session;
11     TPM2B_DIGEST              authHash;
12     HASH_STATE                hashState;
13     TPMT_TK_VERIFIED          ticket;
14     TPM_ALG_ID                hashAlg;
15     UINT16                    digestSize;
16
17 // Input Validation
18
19     // Get pointer to the session structure
20     session = SessionGet(in->policySession);
21
22     // Extract from the Name of the key, the algorithm used to compute it's Name
23     hashAlg = BYTE_ARRAY_TO_UINT16(in->keySign.t.name);
24
25     // 'keySign' parameter needs to use a supported hash algorithm, otherwise
26     // can't tell how large the digest should be
27     if(!CryptHashIsValidAlg(hashAlg, FALSE))
28         return TPM_RCS_HASH + RC_PolicyAuthorize_keySign;
29
30     digestSize = CryptHashGetDigestSize(hashAlg);
31     if(digestSize != (in->keySign.t.size - 2))
32         return TPM_RCS_SIZE + RC_PolicyAuthorize_keySign;
33
34     //If this is a trial policy, skip all validations
35     if(session->attributes.isTrialPolicy == CLEAR)
36     {
37         // Check that "approvedPolicy" matches the current value of the
38         // policyDigest in policy session
39         if(!MemoryEqual2B(&session->u2.policyDigest.b,
40                           &in->approvedPolicy.b))
41             return TPM_RCS_VALUE + RC_PolicyAuthorize_approvedPolicy;
42
43         // Validate ticket TPMT_TK_VERIFIED
44         // Compute aHash.  The authorizing object sign a digest
45         //  aHash := hash(approvedPolicy || policyRef).
46         // Start hash
47         authHash.t.size = CryptHashStart(&hashState, hashAlg);
48
49         // add approvedPolicy
50         CryptDigestUpdate2B(&hashState, &in->approvedPolicy.b);
51
```

 Copyright © TCG 2006-2020

```
52          // add policyRef
53          CryptDigestUpdate2B(&hashState, &in->policyRef.b);
54
55          // complete hash
56          CryptHashEnd2B(&hashState, &authHash.b);
57
58          // re-compute TPMT_TK_VERIFIED
59          TicketComputeVerified(in->checkTicket.hierarchy, &authHash,
60                                &in->keySign, &ticket);
61
62          // Compare ticket digest.  If not match, return error
63          if(!MemoryEqual2B(&in->checkTicket.digest.b, &ticket.digest.b))
64              return TPM_RCS_VALUE + RC_PolicyAuthorize_checkTicket;
65      }
66
67  // Internal Data Update
68
69      // Set policyDigest to zero digest
70      PolicyDigestClear(session);
71
72      // Update policyDigest
73      PolicyContextUpdate(TPM_CC_PolicyAuthorize, &in->keySign, &in->policyRef,
74                          NULL, 0, session);
75
76      return TPM_RC_SUCCESS;
77  }
78  #endif // CC_PolicyAuthorize
```

23.17 TPM2_PolicyAuthValue

23.17.1 General Description

This command allows a policy to be bound to the authorization value of the authorized entity.

When this command completes successfully, *policySession→isAuthValueNeeded* is SET to indicate that the *authValue* will be included in *hmacKey* when the authorization HMAC is computed for the command being authorized using this session. Additionally, *policySession→isPasswordNeeded* will be CLEAR.

NOTE If a policy does not use this command, then the *hmacKey* for the authorized command would only use *sessionKey.* If *sessionKey* is not present, then the *hmacKey* is an Empty Buffer and no HMAC would be computed.

If successful, *policySession→policyDigest* will be updated with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyAuthValue}) \quad (36)$$

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.17.2 Command and Response

Table 152 — TPM2_PolicyAuthValue Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyAuthValue
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None

Table 153 — TPM2_PolicyAuthValue Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

23.17.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "PolicyAuthValue_fp.h"
3   #if CC_PolicyAuthValue  // Conditional expansion of this file
4   #include "Policy_spt_fp.h"
5   TPM_RC
6   TPM2_PolicyAuthValue(
7       PolicyAuthValue_In  *in              // IN: input parameter list
8       )
9   {
10      SESSION             *session;
11      TPM_CC               commandCode = TPM_CC_PolicyAuthValue;
12      HASH_STATE           hashState;
13
14  // Internal Data Update
15
16      // Get pointer to the session structure
17      session = SessionGet(in->policySession);
18
19      // Update policy hash
20      // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyAuthValue)
21      //   Start hash
22      CryptHashStart(&hashState, session->authHashAlg);
23
24      //  add old digest
25      CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
26
27      //  add commandCode
28      CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
29
30      //  complete the hash and get the results
31      CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
32
33      // update isAuthValueNeeded bit in the session context
34      session->attributes.isAuthValueNeeded = SET;
35      session->attributes.isPasswordNeeded = CLEAR;
36
37      return TPM_RC_SUCCESS;
38  }
39  #endif // CC_PolicyAuthValue
```


 Copyright © TCG 2006-2020

23.18 TPM2_PolicyPassword

23.18.1 General Description

This command allows a policy to be bound to the authorization value of the authorized object.

When this command completes successfully, *policySession→isPasswordNeeded* is SET to indicate that *authValue* of the authorized object will be checked when the session is used for authorization. The caller will provide the *authValue* in clear text in the *hmac* parameter of the authorization. The comparison of *hmac* to *authValue* is performed as if the authorization is a password.

NOTE 1 The parameter field in the policy session where the authorization value is provided is called *hmac*. If TPM2_PolicyPassword() is part of the sequence, then the field will contain a password and not an HMAC.

If successful, *policySession→policyDigest* will be updated with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM_CC_PolicyAuthValue}) \quad (37)$$

NOTE 2 This is the same extend value as used with TPM2_PolicyAuthValue so that the evaluation may be done using either an HMAC or a password with no change to the *authPolicy* of the object. The reason that two commands are present is to indicate to the TPM if the *hmac* field in the authorization will contain an HMAC or a password value.

When this command is successful, *policySession→isAuthValueNeeded* will be CLEAR.

 Copyright © TCG 2006-2020

23.18.2 Command and Response

Table 154 — TPM2_PolicyPassword Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyPassword
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None

Table 155 — TPM2_PolicyPassword Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.18.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PolicyPassword_fp.h"
3  #if CC_PolicyPassword  // Conditional expansion of this file
4  #include "Policy_spt_fp.h"
5  TPM_RC
6  TPM2_PolicyPassword(
7      PolicyPassword_In   *in              // IN: input parameter list
8      )
9  {
10     SESSION             *session;
11     TPM_CC               commandCode = TPM_CC_PolicyAuthValue;
12     HASH_STATE           hashState;
13
14 // Internal Data Update
15
16     // Get pointer to the session structure
17     session = SessionGet(in->policySession);
18
19     // Update policy hash
20     // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyAuthValue)
21     //  Start hash
22     CryptHashStart(&hashState, session->authHashAlg);
23
24     //  add old digest
25     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
26
27     //  add commandCode
28     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
29
30     //  complete the digest
31     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
32
33     //  Update isPasswordNeeded bit
34     session->attributes.isPasswordNeeded = SET;
35     session->attributes.isAuthValueNeeded = CLEAR;
36
37     return TPM_RC_SUCCESS;
38 }
39 #endif // CC_PolicyPassword
```

23.19 TPM2_PolicyGetDigest

23.19.1 General Description

This command returns the current *policyDigest* of the session. This command allows the TPM to be used to perform the actions required to pre-compute the *authPolicy* for an object.

 Copyright © TCG 2006-2020

23.19.2 Command and Response

Table 156 — TPM2_PolicyGetDigest Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyGetDigest
TPMI_SH_POLICY	policySession	handle for the policy session Auth Index: None

Table 157 — TPM2_PolicyGetDigest Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	policyDigest	the current value of the *policySession→policyDigest*

 Copyright © TCG 2006-2020

23.19.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "PolicyGetDigest_fp.h"
3   #if CC_PolicyGetDigest  // Conditional expansion of this file
4   TPM_RC
5   TPM2_PolicyGetDigest(
6       PolicyGetDigest_In      *in,            // IN: input parameter list
7       PolicyGetDigest_Out     *out            // OUT: output parameter list
8       )
9   {
10      SESSION     *session;
11
12  // Command Output
13
14      // Get pointer to the session structure
15      session = SessionGet(in->policySession);
16
17      out->policyDigest = session->u2.policyDigest;
18
19      return TPM_RC_SUCCESS;
20  }
21  #endif // CC_PolicyGetDigest
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.20 TPM2_PolicyNvWritten

23.20.1 General Description

This command allows a policy to be bound to the TPMA_NV_WRITTEN attributes. This is a deferred assertion. Values are stored in the policy session context and checked when the policy is used for authorization.

If *policySession→checkNVWritten* is CLEAR, it is SET and *policySession→nvWrittenState* is set to *writtenSet*. If *policySession→checkNVWritten* is SET, the TPM will return TPM_RC_VALUE if *policySession→nvWrittenState* and *writtenSet* are not the same.

If the TPM does not return an error, it will update *policySession→policyDigest* by

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyNvWritten} \mathbin{||} writtenSet) \quad (38)$$

When the policy session is used to authorize a command, the TPM will fail the command if *policySession→checkNVWritten* is SET and *nvIndex→attributes→TPMA_NV_WRITTEN does not match policySession→nvWrittenState.*

NOTE 1 A typical use case is a simple policy for the first write during manufacturing provisioning that would require TPMA_NV_WRITTEN CLEAR and a more complex policy for later use that would require TPMA_NV_WRITTEN SET.

NOTE 2 When an Index is written, it has a different authorization name than an Index that has not been written. It is possible to use this change in the NV Index to create a write-once Index.

 Copyright © TCG 2006-2020

23.20.2 Command and Response

Table 158 — TPM2_PolicyNvWritten Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyNvWritten
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPMI_YES_NO	writtenSet	YES if NV Index is required to have been written NO if NV Index is required not to have been written

Table 159 — TPM2_PolicyNvWritten Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.20.3 Detailed Actions

```
1 #include "Tpm.h"
2 #include "PolicyNvWritten_fp.h"
3 #if CC_PolicyNvWritten // Conditional expansion of this file
```

Make an NV Index policy dependent on the state of the TPMA_NV_WRITTEN attribute of the index.

Error Returns	Meaning
TPM_RC_VALUE	a conflicting request for the attribute has already been processed

```
4 TPM_RC
5 TPM2_PolicyNvWritten(
6     PolicyNvWritten_In  *in              // IN: input parameter list
7     )
8 {
9     SESSION     *session;
10    TPM_CC       commandCode = TPM_CC_PolicyNvWritten;
11    HASH_STATE   hashState;
12
13 // Input Validation
14
15    // Get pointer to the session structure
16    session = SessionGet(in->policySession);
17
18    // If already set is this a duplicate (the same setting)? If it
19    // is a conflicting setting, it is an error
20    if(session->attributes.checkNvWritten == SET)
21    {
22        if(((session->attributes.nvWrittenState == SET)
23            != (in->writtenSet == YES)))
24            return TPM_RCS_VALUE + RC_PolicyNvWritten_writtenSet;
25    }
26
27 // Internal Data Update
28
29    // Set session attributes so that the NV Index needs to be checked
30    session->attributes.checkNvWritten = SET;
31    session->attributes.nvWrittenState = (in->writtenSet == YES);
32
33    // Update policy hash
34    // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyNvWritten
35    //                          || writtenSet)
36    // Start hash
37    CryptHashStart(&hashState, session->authHashAlg);
38
39    // add old digest
40    CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
41
42    // add commandCode
43    CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
44
45    // add the byte of writtenState
46    CryptDigestUpdateInt(&hashState, sizeof(TPMI_YES_NO), in->writtenSet);
47
48    // complete the digest
49    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
50
51    return TPM_RC_SUCCESS;
52 }
53 #endif // CC_PolicyNvWritten
```

23.21 TPM2_PolicyTemplate

23.21.1 General Description

This command allows a policy to be bound to a specific creation template. This is most useful for an object creation command such as TPM2_Create(), TPM2_CreatePrimary(), or TPM2_CreateLoaded().

The *templateHash* parameter should contain the digest of the template that will be required for the *inPublic* parameter of an Object creation command.

If *policySession→isTemplateHash* is SET and *policySession→cpHash* is not equal to *templateHash,* the TPM shall return TPM_RC_VALUE.

NOTE 1 Revision 01.38 of this specification permitted the TPM to return TPM_RC_CPHASH.

Otherwise, if *policySession→cpHash* is already set, the TPM shall return TPM_RC_CPHASH.

NOTE 2 Revision 01.38 of this specification permitted the TPM to return TPM_RC_VALUE.

If the size of *templateHash* is not the size of *policySession→policyDigest*, the TPM shall return TPM_RC_SIZE. Otherwise, *policySession→cpHash* is set to *templateHash*.

NOTE 3 The digest calculation includes the TPM2B buffer but not the TPM2B size.

If this command completes successfully, the *cpHash* of the authorized command will not be used for validation. Only the digest of the *inPublic* parameter will be used.

NOTE 4 This allows the space normally used to hold *policySession→cpHash* to be used for *policySession→templateHash* instead.

The *policySession→policyDigest* will be updated with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyTemplate} \mathbin{||} templateHash) \qquad (39)$$

 Copyright © TCG 2006-2020

23.21.2 Command and Response

Table 160 — TPM2_PolicyTemplate Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyTemplate
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_DIGEST	templateHash	the digest to be added to the policy

Table 161 — TPM2_PolicyTemplate Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

23.21.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PolicyTemplate_fp.h"
3  #if CC_PolicyTemplate  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_CPHASH	*cpHash* of *policySession* has previously been set to a different value
TPM_RC_SIZE	*templateHash* is not the size of a digest produced by the hash algorithm associated with *policySession*

```
4  TPM_RC
5  TPM2_PolicyTemplate(
6      PolicyTemplate_In     *in              // IN: input parameter list
7      )
8  {
9      SESSION     *session;
10     TPM_CC      commandCode = TPM_CC_PolicyTemplate;
11     HASH_STATE  hashState;
12
13 // Input Validation
14
15     // Get pointer to the session structure
16     session = SessionGet(in->policySession);
17
18     // If the template is set, make sure that it is the same as the input value
19     if(session->attributes.isTemplateSet)
20     {
21         if(!MemoryEqual2B(&in->templateHash.b, &session->u1.cpHash.b))
22             return TPM_RCS_VALUE + RC_PolicyTemplate_templateHash;
23     }
24     // error if cpHash contains something that is not a template
25     else if(session->u1.templateHash.t.size != 0)
26         return TPM_RC_CPHASH;
27
28     // A valid templateHash must have the same size as session hash digest
29     if(in->templateHash.t.size != CryptHashGetDigestSize(session->authHashAlg))
30         return TPM_RCS_SIZE + RC_PolicyTemplate_templateHash;
31
32 // Internal Data Update
33     // Update policy hash
34     // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyCpHash
35     //  || cpHashA.buffer)
36     //  Start hash
37     CryptHashStart(&hashState, session->authHashAlg);
38
39     //  add old digest
40     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
41
42     //  add commandCode
43     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
44
45     //  add cpHashA
46     CryptDigestUpdate2B(&hashState, &in->templateHash.b);
47
48     //  complete the digest and get the results
49     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
50
51     // update cpHash in session context
52     session->u1.templateHash = in->templateHash;
53     session->attributes.isTemplateSet = SET;
```

 Copyright © TCG 2006-2020

```
54
55      return TPM_RC_SUCCESS;
56  }
57  #endif // CC_PolicyTemplateHash
```

 Copyright © TCG 2006-2020

23.22 TPM2_PolicyAuthorizeNV

23.22.1 General Description

This command provides a capability that is the equivalent of a revocable policy. With TPM2_PolicyAuthorize(), the authorization ticket never expires, so the authorization may not be withdrawn. With this command, the approved policy is kept in an NV Index location so that the policy may be changed as needed to render the old policy unusable.

NOTE 1 This command is useful for Objects but of limited value for other policies that are persistently stored in TPM NV, such as the OwnerPolicy.

An authorization session providing authorization to read the NV Index shall be provided.

The authorization to read the NV Index must succeed even if *policySession* is a trial policy session.

If *policySession* is a trial policy session, the TPM will update *policySession→policyDigest* as shown in equation (40) below and return TPM_RC_SUCCESS. It will not perform any further validation. The remainder of this general description would apply only if *policySession* is not a trial policy session.

NOTE 2 If read access is controlled by policy, the policy should include a branch that authorizes a TPM2_PolicyAuthorizeNV().

If TPMA_NV_WRITTEN is not SET in the Index referenced by *nvIndex*, the TPM shall return TPM_RC_NV_UNINITIALIZED. If TPMA_NV_READLOCKED of the NV Index is SET, then the TPM shall return TPM_RC_NV_LOCKED.

The *dataSize* of the NV Index referenced by *nvIndex* is required to be at least large enough to hold a properly formatted TPMT_HA (TPM_RC_INSUFFICIENT).

NOTE 3 A TPMT_HA contains a TPM_ALG_ID followed a digest that is consistent in size with the hash algorithm indicated by the TPM_ALG_ID.

It is an error (TPM_RC_HASH) if the first two octets of the Index are not a TPM_ALG_ID for a hash algorithm implemented on the TPM or if the indicated hash algorithm does not match *policySession→authHash*.

NOTE 4 The TPM_ALG_ID is stored in the first two octets in big endian format.

The TPM will compare *policySession→policyDigest* to the contents of the NV Index, starting at the first octet after the TPM_ALG_ID (the third octet) and return TPM_RC_VALUE if they are not the same.

NOTE 5 If the Index does not contain enough bytes for the compare, then TPM_RC_INSUFFICENT is generated as indicated above.

NOTE 6 The *dataSize* of the Index may be larger than is required for this command. This permits the Index to include metadata.

If the comparison is successful, the TPM will reset *policySession→policyDigest* to a Zero Digest. Then it will update *policySession→policyDigest* with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyAuthorizeNV} \mathbin{||} nvIndex{\rightarrow}Name) \quad (40)$$

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.22.2 Command and Response

Table 162 — TPM2_PolicyAuthorizeNV Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyAuthorizeNV
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index of the area to read Auth Index: None
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None

Table 163 — TPM2_PolicyAuthorizeNV Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

23.22.3 Detailed Actions

```
1   #include "Tpm.h"
2   #if CC_PolicyAuthorizeNV  // Conditional expansion of this file
3   #include "PolicyAuthorizeNV_fp.h"
4   #include "Policy_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_HASH	hash algorithm in *keyName* is not supported or is not the same as the hash algorithm of the policy session
TPM_RC_SIZE	*keyName* is not the correct size for its hash algorithm
TPM_RC_VALUE	the current *policyDigest* of *policySession* does not match *approvedPolicy*; or *checkTicket* doesn't match the provided values

```
5   TPM_RC
6   TPM2_PolicyAuthorizeNV(
7       PolicyAuthorizeNV_In    *in
8       )
9   {
10      SESSION                 *session;
11      TPM_RC                   result;
12      NV_REF                   locator;
13      NV_INDEX                *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
14      TPM2B_NAME               name;
15      TPMT_HA                  policyInNv;
16      BYTE                     nvTemp[sizeof(TPMT_HA)];
17      BYTE                    *buffer = nvTemp;
18      INT32                    size;
19
20  // Input Validation
21      // Get pointer to the session structure
22      session = SessionGet(in->policySession);
23
24      // Skip checks if this is a trial policy
25      if(!session->attributes.isTrialPolicy)
26      {
27          // Check the authorizations for reading
28          // Common read access checks. NvReadAccessChecks() returns
29          // TPM_RC_NV_AUTHORIZATION, TPM_RC_NV_LOCKED, or TPM_RC_NV_UNINITIALIZED
30          // error may be returned at this point
31          result = NvReadAccessChecks(in->authHandle, in->nvIndex,
32                                      nvIndex->publicArea.attributes);
33          if(result != TPM_RC_SUCCESS)
34              return result;
35
36          // Read the contents of the index into a temp buffer
37          size = MIN(nvIndex->publicArea.dataSize, sizeof(TPMT_HA));
38          NvGetIndexData(nvIndex, locator, 0, (UINT16)size, nvTemp);
39
40          // Unmarshal the contents of the buffer into the internal format of a
41          // TPMT_HA so that the hash and digest elements can be accessed from the
42          // structure rather than the byte array that is in the Index (written by
43          // user of the Index).
44          result = TPMT_HA_Unmarshal(&policyInNv, &buffer, &size, FALSE);
45          if(result != TPM_RC_SUCCESS)
46              return result;
47
48          // Verify that the hash is the same
49          if(policyInNv.hashAlg != session->authHashAlg)
50              return TPM_RC_HASH;
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

```
51
52          // See if the contents of the digest in the Index matches the value
53          // in the policy
54          if(!MemoryEqual(&policyInNv.digest, &session->u2.policyDigest.t.buffer,
55                          session->u2.policyDigest.t.size))
56              return TPM_RC_VALUE;
57      }
58
59  // Internal Data Update
60
61      // Set policyDigest to zero digest
62      PolicyDigestClear(session);
63
64      // Update policyDigest
65      PolicyContextUpdate(TPM_CC_PolicyAuthorizeNV, EntityGetName(in->nvIndex, &name),
66                          NULL, NULL, 0, session);
67
68      return TPM_RC_SUCCESS;
69  }
70  #endif // CC_PolicyAuthorize
```

24 Hierarchy Commands

24.1 TPM2_CreatePrimary

24.1.1 General Description

This command is used to create a Primary Object under one of the Primary Seeds or a Temporary Object under TPM_RH_NULL. The command uses a TPM2B_PUBLIC as a template for the object to be created. The size of the *unique* field shall not be checked for consistency with the other object parameters. The command will create and load a Primary Object. The sensitive area is not returned.

NOTE 1 Since the sensitive data is not returned, the key cannot be reloaded. It can either be made persistent or it can be recreated.

NOTE 2 For interoperability, the *unique* field should not be set to a value that is larger than allowed by object parameters, so that the unmarshaling will not fail.

NOTE 3 An Empty Buffer is a legal *unique* field value.

EXAMPLE 1 A TPM_ALG_RSA object with a *keyBits* of 2048 in the objects parameters should have a *unique* field that is no larger than 256 bytes.

EXAMPLE 2 A TPM_ALG_KEYEDHASH or a TPM_ALG_SYMCIPHER object should have a *unique* field this is no larger than the digest produced by the object's *nameAlg*.

Any type of object and attributes combination that is allowed by TPM2_Create() may be created by this command. The constraints on templates and parameters are the same as TPM2_Create() except that a Primary Storage Key and a Temporary Storage Key are not constrained to use the algorithms of their parents.

For setting of the attributes of the created object, *fixedParent*, *fixedTPM*, decrypt, and restricted are implied to be SET in the parent (a Permanent Handle). The remaining attributes are implied to be CLEAR.

The TPM will derive the object from the Primary Seed indicated in *primaryHandle* using an approved KDF. All of the bits of the template are used in the creation of the Primary Key. Methods for creating a Primary Object from a Primary Seed are described in TPM 2.0 Part 1 and implemented in TPM 2.0 Part 4.

If this command is called multiple times with the same *inPublic* parameter, *inSensitive.data*, and Primary Seed, the TPM shall produce the same Primary Object.

NOTE 4 If the Primary Seed is changed, the Primary Objects generated with the new seed shall be statistically unique even if the parameters of the call are the same.

This command requires authorization. Authorization for a Primary Object attached to the Platform Primary Seed (PPS) shall be provided by *platformAuth* or *platformPolicy*. Authorization for a Primary Object attached to the Storage Primary Seed (SPS) shall be provided by *ownerAuth* or *ownerPolicy*. Authorization for a Primary Key attached to the Endorsement Primary Seed (EPS) shall be provided by *endorsementAuth* or *endorsementPolicy*.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

24.1.2 Command and Response

Table 164 — TPM2_CreatePrimary Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_CreatePrimary
TPMI_RH_HIERARCHY+	@primaryHandle	TPM_RH_ENDORSEMENT, TPM_RH_OWNER, TPM_RH_PLATFORM+{PP}, or TPM_RH_NULL Auth Index: 1 Auth Role: USER
TPM2B_SENSITIVE_CREATE	inSensitive	the sensitive data, see TPM 2.0 Part 1 Sensitive Values
TPM2B_PUBLIC	inPublic	the public template
TPM2B_DATA	outsideInfo	data that will be included in the creation data for this object to provide permanent, verifiable linkage between this object and some object owner data
TPML_PCR_SELECTION	creationPCR	PCR that will be used in creation data

Table 165 — TPM2_CreatePrimary Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM_HANDLE	objectHandle	handle of type TPM_HT_TRANSIENT for created Primary Object
TPM2B_PUBLIC	outPublic	the public portion of the created object
TPM2B_CREATION_DATA	creationData	contains a TPMT_CREATION_DATA
TPM2B_DIGEST	creationHash	digest of *creationData* using *nameAlg* of *outPublic*
TPMT_TK_CREATION	creationTicket	ticket used by TPM2_CertifyCreation() to validate that the creation data was produced by the TPM
TPM2B_NAME	name	the name of the created object

 Copyright © TCG 2006-2020

24.1.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "CreatePrimary_fp.h"
3  #if CC_CreatePrimary  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*sensitiveDataOrigin* is CLEAR when sensitive.data is an Empty Buffer *fixedTPM*, *fixedParent*, or *encryptedDuplication* attributes are inconsistent between themselves or with those of the parent object; inconsistent *restricted*, *decrypt* and *sign* attributes attempt to inject sensitive data for an asymmetric key;
TPM_RC_KDF	incorrect KDF specified for decrypting keyed hash object
TPM_RC_KEY	a provided symmetric key value is not allowed
TPM_RC_OBJECT_MEMORY	there is no free slot for the object
TPM_RC_SCHEME	inconsistent attributes *decrypt*, *sign*, *restricted* and key's scheme ID; or hash algorithm is inconsistent with the scheme ID for keyed hash object
TPM_RC_SIZE	size of public authorization policy or sensitive authorization value does not match digest size of the name algorithm; or sensitive data size for the keyed hash object is larger than is allowed for the scheme
TPM_RC_SYMMETRIC	a storage key with no symmetric algorithm specified; or non-storage key with symmetric algorithm different from TPM_ALG_NULL
TPM_RC_TYPE	unknown object type

```
4  TPM_RC
5  TPM2_CreatePrimary(
6      CreatePrimary_In    *in,            // IN: input parameter list
7      CreatePrimary_Out   *out            // OUT: output parameter list
8      )
9  {
10     TPM_RC               result = TPM_RC_SUCCESS;
11     TPMT_PUBLIC         *publicArea;
12     DRBG_STATE           rand;
13     OBJECT              *newObject;
14     TPM2B_NAME           name;
15
16 // Input Validation
17     // Will need a place to put the result
18     newObject = FindEmptyObjectSlot(&out->objectHandle);
19     if(newObject == NULL)
20         return TPM_RC_OBJECT_MEMORY;
21     // Get the address of the public area in the new object
22     // (this is just to save typing)
23     publicArea = &newObject->publicArea;
24
25     *publicArea = in->inPublic.publicArea;
26
27     // Check attributes in input public area. CreateChecks() checks the things that
28     // are unique to creation and then validates the attributes and values that are
29     // common to create and load.
30     result = CreateChecks(NULL, publicArea,
31                           in->inSensitive.sensitive.data.t.size);
32     if(result != TPM_RC_SUCCESS)
33         return RcSafeAddToResult(result, RC_CreatePrimary_inPublic);
34     // Validate the sensitive area values
```


 Copyright © TCG 2006-2020

```
35      if(!AdjustAuthSize(&in->inSensitive.sensitive.userAuth,
36                         publicArea->nameAlg))
37          return TPM_RCS_SIZE + RC_CreatePrimary_inSensitive;
38  // Command output
39      // Compute the name using out->name as a scratch area (this is not the value
40      // that ultimately will be returned, then instantiate the state that will be
41      // used as a random number generator during the object creation.
42      // The caller does not know the seed values so the actual name does not have
43      // to be over the input, it can be over the unmarshaled structure.
44      result = DRBG_InstantiateSeeded(&rand,
45                              &HierarchyGetPrimarySeed(in->primaryHandle)->b,
46                              PRIMARY_OBJECT_CREATION,
47                              (TPM2B *)PublicMarshalAndComputeName(publicArea, &name),
48                              &in->inSensitive.sensitive.data.b);
49      if(result == TPM_RC_SUCCESS)
50      {
51          newObject->attributes.primary = SET;
52          if(in->primaryHandle == TPM_RH_ENDORSEMENT)
53              newObject->attributes.epsHierarchy = SET;
54
55          // Create the primary object.
56          result = CryptCreateObject(newObject, &in->inSensitive.sensitive,
57              (RAND_STATE *)&rand);
58      }
59      if(result != TPM_RC_SUCCESS)
60          return result;
61
62      // Set the publicArea and name from the computed values
63      out->outPublic.publicArea = newObject->publicArea;
64      out->name = newObject->name;
65
66      // Fill in creation data
67      FillInCreationData(in->primaryHandle, publicArea->nameAlg,
68                         &in->creationPCR, &in->outsideInfo, &out->creationData,
69                         &out->creationHash);
70
71      // Compute creation ticket
72      TicketComputeCreation(EntityGetHierarchy(in->primaryHandle), &out->name,
73                            &out->creationHash, &out->creationTicket);
74
75      // Set the remaining attributes for a loaded object
76      ObjectSetLoadedAttributes(newObject, in->primaryHandle);
77      return result;
78  }
79  #endif // CC_CreatePrimary
```

 Copyright © TCG 2006-2020

24.2 TPM2_HierarchyControl

24.2.1 General Description

This command enables and disables use of a hierarchy and its associated NV storage. The command allows *phEnable*, *phEnableNV*, *shEnable*, and *ehEnable* to be changed when the proper authorization is provided.

This command may be used to CLEAR *phEnable* and *phEnableNV* if *platformAuth/platformPolicy* is provided. *phEnable* may not be SET using this command.

This command may be used to CLEAR *shEnable* if either *platformAuth/platformPolicy* or *ownerAuth/ownerPolicy* is provided. *shEnable* may be SET if *platformAuth*/*platformPolicy* is provided.

This command may be used to CLEAR *ehEnable* if either *platformAuth*/*platformPolicy* or *endorsementAuth*/*endorsementPolicy* is provided. *ehEnable* may be SET if *platformAuth*/*platformPolicy* is provided.

When this command is used to CLEAR *phEnable*, *shEnable*, or *ehEnable*, the TPM will disable use of any persistent entity associated with the disabled hierarchy and will flush any transient objects associated with the disabled hierarchy.

When this command is used to CLEAR *shEnable,* the TPM will disable access to any NV index that has TPMA_NV_PLATFORMCREATE CLEAR (indicating that the NV Index was defined using Owner Authorization). As long as *shEnable* is CLEAR, the TPM will return an error in response to any command that attempts to operate upon an NV index that has TPMA_NV_PLATFORMCREATE CLEAR.

When this command is used to CLEAR *phEnableNV*, the TPM will disable access to any NV index that has TPMA_NV_PLATFORMCREATE SET (indicating that the NV Index was defined using Platform Authorization). As long as *phEnableNV* is CLEAR, the TPM will return an error in response to any command that attempts to operate upon an NV index that has TPMA_NV_PLATFORMCREATE SET.

 Copyright © TCG 2006-2020

24.2.2 Command and Response

Table 166 — TPM2_HierarchyControl Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_HierarchyControl {NV E}
TPMI_RH_HIERARCHY	@authHandle	TPM_RH_ENDORSEMENT, TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPMI_RH_ENABLES	enable	the enable being modified TPM_RH_ENDORSEMENT, TPM_RH_OWNER, TPM_RH_PLATFORM, or TPM_RH_PLATFORM_NV
TPMI_YES_NO	state	YES if the enable should be SET, NO if the enable should be CLEAR

Table 167 — TPM2_HierarchyControl Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

24.2.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "HierarchyControl_fp.h"
3   #if CC_HierarchyControl  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_AUTH_TYPE	*authHandle* is not applicable to *hierarchy* in its current state

```
4   TPM_RC
5   TPM2_HierarchyControl(
6       HierarchyControl_In     *in             // IN: input parameter list
7       )
8   {
9       BOOL        select = (in->state == YES);
10      BOOL        *selected = NULL;
11
12  // Input Validation
13      switch(in->enable)
14      {
15          // Platform hierarchy has to be disabled by PlatformAuth
16          // If the platform hierarchy has already been disabled, only a reboot
17          // can enable it again
18          case TPM_RH_PLATFORM:
19          case TPM_RH_PLATFORM_NV:
20              if(in->authHandle != TPM_RH_PLATFORM)
21                  return TPM_RC_AUTH_TYPE;
22              break;
23
24          // ShEnable may be disabled if PlatformAuth/PlatformPolicy or
25          // OwnerAuth/OwnerPolicy is provided.  If ShEnable is disabled, then it
26          // may only be enabled if PlatformAuth/PlatformPolicy is provided.
27          case TPM_RH_OWNER:
28              if(in->authHandle != TPM_RH_PLATFORM
29                 && in->authHandle != TPM_RH_OWNER)
30                  return TPM_RC_AUTH_TYPE;
31              if(gc.shEnable == FALSE && in->state == YES
32                 && in->authHandle != TPM_RH_PLATFORM)
33                  return TPM_RC_AUTH_TYPE;
34              break;
35
36          // EhEnable may be disabled if either PlatformAuth/PlatformPolicy or
37          // EndosementAuth/EndorsementPolicy is provided.  If EhEnable is disabled,
38          // then it may only be enabled if PlatformAuth/PlatformPolicy is
39          // provided.
40          case TPM_RH_ENDORSEMENT:
41              if(in->authHandle != TPM_RH_PLATFORM
42                 && in->authHandle != TPM_RH_ENDORSEMENT)
43                  return TPM_RC_AUTH_TYPE;
44              if(gc.ehEnable == FALSE && in->state == YES
45                 && in->authHandle != TPM_RH_PLATFORM)
46                  return TPM_RC_AUTH_TYPE;
47              break;
48          default:
49              FAIL(FATAL_ERROR_INTERNAL);
50              break;
51      }
52
53  // Internal Data Update
54
55      // Enable or disable the selected hierarchy
56      // Note: the authorization processing for this command may keep these
```

 Copyright © TCG 2006-2020

```
 57        // command actions from being executed. For example, if phEnable is
 58        // CLEAR, then platformAuth cannot be used for authorization. This
 59        // means that would not be possible to use platformAuth to change the
 60        // state of phEnable from CLEAR to SET.
 61        // If it is decided that platformPolicy can still be used when phEnable
 62        // is CLEAR, then this code could SET phEnable when proper platform
 63        // policy is provided.
 64        switch(in->enable)
 65        {
 66            case TPM_RH_OWNER:
 67                selected = &gc.shEnable;
 68                break;
 69            case TPM_RH_ENDORSEMENT:
 70                selected = &gc.ehEnable;
 71                break;
 72            case TPM_RH_PLATFORM:
 73                selected = &g_phEnable;
 74                break;
 75            case TPM_RH_PLATFORM_NV:
 76                selected = &gc.phEnableNV;
 77                break;
 78            default:
 79                FAIL(FATAL_ERROR_INTERNAL);
 80                break;
 81        }
 82        if(selected != NULL && *selected != select)
 83        {
 84            // Before changing the internal state, make sure that NV is available.
 85            // Only need to update NV if changing the orderly state
 86            RETURN_IF_ORDERLY;
 87
 88            // state is changing and NV is available so modify
 89            *selected = select;
 90            // If a hierarchy was just disabled, flush it
 91            if(select == CLEAR && in->enable != TPM_RH_PLATFORM_NV)
 92            // Flush hierarchy
 93                ObjectFlushHierarchy(in->enable);
 94
 95            // orderly state should be cleared because of the update to state clear data
 96            // This gets processed in ExecuteCommand() on the way out.
 97            g_clearOrderly = TRUE;
 98        }
 99        return TPM_RC_SUCCESS;
100    }
101    #endif // CC_HierarchyControl
```

 Copyright © TCG 2006-2020

24.3 TPM2_SetPrimaryPolicy

24.3.1 General Description

This command allows setting of the authorization policy for the lockout (*lockoutPolicy*), the platform hierarchy (*platformPolicy*), the storage hierarchy (*ownerPolicy*), and the endorsement hierarchy (*endorsementPolicy*). On TPMs implementing Authenticated Countdown Timers (ACT), this command may also be used to set the authorization policy for an ACT.

The command requires an authorization session. The session shall use the current *authValue* or satisfy the current *authPolicy* for the referenced hierarchy, or the ACT.

The policy that is changed is the policy associated with *authHandle*.

If the enable associated with *authHandle* is not SET, then the associated authorization values (*authValue* or *authPolicy*) may not be used, and the TPM returns TPM_RC_HIERARCHY.

When *hashAlg* is not TPM_ALG_NULL, if the size of *authPolicy* is not consistent with the hash algorithm, the TPM returns TPM_RC_SIZE.

 Copyright © TCG 2006-2020

24.3.2 Command and Response

Table 168 — TPM2_SetPrimaryPolicy Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_SetPrimaryPolicy {NV}
TPMI_RH_HIERARCHY_POLICY	@authHandle	TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_OWNER, TPMI_RH_ACT or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPM2B_DIGEST	authPolicy	an authorization policy digest; may be the Empty Buffer If *hashAlg* is TPM_ALG_NULL, then this shall be an Empty Buffer.
TPMI_ALG_HASH+	hashAlg	the hash algorithm to use for the policy If the *authPolicy* is an Empty Buffer, then this field shall be TPM_ALG_NULL.

Table 169 — TPM2_SetPrimaryPolicy Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

24.3.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "SetPrimaryPolicy_fp.h"
3  #if CC_SetPrimaryPolicy  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_SIZE	size of input *authPolicy* is not consistent with input hash algorithm

```
4  TPM_RC
5  TPM2_SetPrimaryPolicy(
6      SetPrimaryPolicy_In     *in              // IN: input parameter list
7      )
8  {
9  // Input Validation
10
11     // Check the authPolicy consistent with hash algorithm. If the policy size is
12     // zero, then the algorithm is required to be TPM_ALG_NULL
13     if(in->authPolicy.t.size != CryptHashGetDigestSize(in->hashAlg))
14         return TPM_RCS_SIZE + RC_SetPrimaryPolicy_authPolicy;
15
16     // The command need NV update for OWNER and ENDORSEMENT hierarchy, and
17     // might need orderlyState update for PLATFROM hierarchy.
18     // Check if NV is available.  A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE
19     // error may be returned at this point
20     RETURN_IF_NV_IS_NOT_AVAILABLE;
21
22 // Internal Data Update
23
24     // Set hierarchy policy
25     switch(in->authHandle)
26     {
27         case TPM_RH_OWNER:
28             gp.ownerAlg = in->hashAlg;
29             gp.ownerPolicy = in->authPolicy;
30             NV_SYNC_PERSISTENT(ownerAlg);
31             NV_SYNC_PERSISTENT(ownerPolicy);
32             break;
33         case TPM_RH_ENDORSEMENT:
34             gp.endorsementAlg = in->hashAlg;
35             gp.endorsementPolicy = in->authPolicy;
36             NV_SYNC_PERSISTENT(endorsementAlg);
37             NV_SYNC_PERSISTENT(endorsementPolicy);
38             break;
39         case TPM_RH_PLATFORM:
40             gc.platformAlg = in->hashAlg;
41             gc.platformPolicy = in->authPolicy;
42             // need to update orderly state
43             g_clearOrderly = TRUE;
44             break;
45         case TPM_RH_LOCKOUT:
46             gp.lockoutAlg = in->hashAlg;
47             gp.lockoutPolicy = in->authPolicy;
48             NV_SYNC_PERSISTENT(lockoutAlg);
49             NV_SYNC_PERSISTENT(lockoutPolicy);
50             break;
51
52 #define SET_ACT_POLICY(N)                                                        \
53         case TPM_RH_ACT_##N:                                                     \
54             go.ACT_##N.hashAlg = in->hashAlg;                                    \
55             go.ACT_##N.authPolicy = in->authPolicy;                              \
56             g_clearOrderly = TRUE;                                               \
```

 Copyright © TCG 2006-2020

```
57              break;
58
59              FOR_EACH_ACT(SET_ACT_POLICY)
60
61          default:
62              FAIL(FATAL_ERROR_INTERNAL);
63              break;
64      }
65
66      return TPM_RC_SUCCESS;
67  }
68  #endif // CC_SetPrimaryPolicy
```

24.4 TPM2_ChangePPS

24.4.1 General Description

This replaces the current platform primary seed (PPS) with a value from the RNG and sets *platformPolicy* to the default initialization value (the Empty Buffer).

NOTE 1 A policy that is the Empty Buffer can match no policy.

NOTE 2 Platform Authorization is not changed.

All resident transient and persistent objects in the Platform hierarchy are flushed.

Saved contexts in the Platform hierarchy that were created under the old PPS will no longer be able to be loaded.

The policy hash algorithm for PCR is reset to TPM_ALG_NULL.

This command does not clear any NV Index values.

NOTE 3 Index values belonging to the Platform are preserved because the indexes may have configuration information that will be the same after the PPS changes. The Platform may remove the indexes that are no longer needed using TPM2_NV_UndefineSpace().

This command requires Platform Authorization.

 Copyright © TCG 2006-2020

24.4.2 Command and Response

Table 170 — TPM2_ChangePPS Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ChangePPS {NV E}
TPMI_RH_PLATFORM	@authHandle	TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER

Table 171 — TPM2_ChangePPS Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

24.4.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "ChangePPS_fp.h"
3   #if CC_ChangePPS  // Conditional expansion of this file
4   TPM_RC
5   TPM2_ChangePPS(
6       ChangePPS_In    *in             // IN: input parameter list
7       )
8   {
9       UINT32          i;
10
11      // Check if NV is available.  A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE
12      // error may be returned at this point
13      RETURN_IF_NV_IS_NOT_AVAILABLE;
14
15      // Input parameter is not reference in command action
16      NOT_REFERENCED(in);
17
18  // Internal Data Update
19
20      // Reset platform hierarchy seed from RNG
21      CryptRandomGenerate(sizeof(gp.PPSeed.t.buffer), gp.PPSeed.t.buffer);
22
23      // Create a new phProof value from RNG to prevent the saved platform
24      // hierarchy contexts being loaded
25      CryptRandomGenerate(sizeof(gp.phProof.t.buffer), gp.phProof.t.buffer);
26
27      // Set platform authPolicy to null
28      gc.platformAlg = TPM_ALG_NULL;
29      gc.platformPolicy.t.size = 0;
30
31      // Flush loaded object in platform hierarchy
32      ObjectFlushHierarchy(TPM_RH_PLATFORM);
33
34      // Flush platform evict object and index in NV
35      NvFlushHierarchy(TPM_RH_PLATFORM);
36
37      // Save hierarchy changes to NV
38      NV_SYNC_PERSISTENT(PPSeed);
39      NV_SYNC_PERSISTENT(phProof);
40
41      // Re-initialize PCR policies
42  #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
43      for(i = 0; i < NUM_POLICY_PCR_GROUP; i++)
44      {
45          gp.pcrPolicies.hashAlg[i] = TPM_ALG_NULL;
46          gp.pcrPolicies.policy[i].t.size = 0;
47      }
48      NV_SYNC_PERSISTENT(pcrPolicies);
49  #endif
50
51      // orderly state should be cleared because of the update to state clear data
52      g_clearOrderly = TRUE;
53
54      return TPM_RC_SUCCESS;
55  }
56  #endif // CC_ChangePPS
```


Copyright © TCG 2006-2020

24.5 TPM2_ChangeEPS

24.5.1 General Description

This replaces the current endorsement primary seed (EPS) with a value from the RNG and sets the Endorsement hierarchy controls to their default initialization values: *ehEnable* is SET, *endorsementAuth* and *endorsementPolicy* are both set to the Empty Buffer. It will flush any resident objects (transient or persistent) in the Endorsement hierarchy and not allow objects in the hierarchy associated with the previous EPS to be loaded.

NOTE In the reference implementation, *ehProof* is a non-volatile value from the RNG. It is allowed that the *ehProof* be generated by a KDF using both the EPS and SPS as inputs. If generated with a KDF, the ehProof can be generated on an as-needed basis or made a non-volatile value.

This command requires Platform Authorization.

 Copyright © TCG 2006-2020

24.5.2 Command and Response

Table 172 — TPM2_ChangeEPS Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ChangeEPS {NV E}
TPMI_RH_PLATFORM	@authHandle	TPM_RH_PLATFORM+{PP} Auth Handle: 1 Auth Role: USER

Table 173 — TPM2_ChangeEPS Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

24.5.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "ChangeEPS_fp.h"
3  #if CC_ChangeEPS  // Conditional expansion of this file
4  TPM_RC
5  TPM2_ChangeEPS(
6      ChangeEPS_In    *in             // IN: input parameter list
7      )
8  {
9      // The command needs NV update.  Check if NV is available.
10     // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
11     // this point
12     RETURN_IF_NV_IS_NOT_AVAILABLE;
13
14     // Input parameter is not reference in command action
15     NOT_REFERENCED(in);
16
17 // Internal Data Update
18
19     // Reset endorsement hierarchy seed from RNG
20     CryptRandomGenerate(sizeof(gp.EPSeed.t.buffer), gp.EPSeed.t.buffer);
21
22     // Create new ehProof value from RNG
23     CryptRandomGenerate(sizeof(gp.ehProof.t.buffer), gp.ehProof.t.buffer);
24
25     // Enable endorsement hierarchy
26     gc.ehEnable = TRUE;
27
28     // set authValue buffer to zeros
29     MemorySet(gp.endorsementAuth.t.buffer, 0, gp.endorsementAuth.t.size);
30     // Set endorsement authValue to null
31     gp.endorsementAuth.t.size = 0;
32
33     // Set endorsement authPolicy to null
34     gp.endorsementAlg = TPM_ALG_NULL;
35     gp.endorsementPolicy.t.size = 0;
36
37     // Flush loaded object in endorsement hierarchy
38     ObjectFlushHierarchy(TPM_RH_ENDORSEMENT);
39
40     // Flush evict object of endorsement hierarchy stored in NV
41     NvFlushHierarchy(TPM_RH_ENDORSEMENT);
42
43     // Save hierarchy changes to NV
44     NV_SYNC_PERSISTENT(EPSeed);
45     NV_SYNC_PERSISTENT(ehProof);
46     NV_SYNC_PERSISTENT(endorsementAuth);
47     NV_SYNC_PERSISTENT(endorsementAlg);
48     NV_SYNC_PERSISTENT(endorsementPolicy);
49
50     // orderly state should be cleared because of the update to state clear data
51     g_clearOrderly = TRUE;
52
53     return TPM_RC_SUCCESS;
54 }
55 #endif // CC_ChangeEPS
```


 Copyright © TCG 2006-2020

24.6 TPM2_Clear

24.6.1 General Description

This command removes all TPM context associated with a specific Owner.

The clear operation will:

- flush resident objects (persistent and volatile) in the Storage and Endorsement hierarchies;
- delete any NV Index with TPMA_NV_PLATFORMCREATE == CLEAR;
- change the storage primary seed (SPS) to a new value from the TPM's random number generator (RNG),
- change *shProof* and *ehProof*,

NOTE 1 The proof values may be set from the RNG or derived from the associated new Primary Seed. If derived from the Primary Seeds, the derivation of *ehProof* shall use both the SPS and EPS. The computation shall use the SPS as an HMAC key and the derived value may then be a parameter in a second HMAC in which the EPS is the HMAC key. The reference design uses values from the RNG.

- SET *shEnable* and *ehEnable*;
- set *ownerAuth*, *endorsementAuth*, and *lockoutAuth* to the Empty Buffer;
- set *ownerPolicy, endorsementPolicy,* and *lockoutPolicy* to the Empty Buffer;
- set *Clock* to zero;
- set *resetCount* to zero;
- set *restartCount* to zero; and
- set *Safe* to YES.
- increment *pcrUpdateCounter*

NOTE 2 This permits an application to create a policy session that is invalidated on TPM2_Clear. The policy needs, ideally as the first term, TPM2_PolicyPCR(). The session is invalidated even if the PCR selection is empty.

This command requires Platform Authorization or Lockout Authorization. If TPM2_ClearControl() has disabled this command, the TPM shall return TPM_RC_DISABLED.

If this command is authorized using *lockoutAuth*, the HMAC in the response shall use the new *lockoutAuth* value (that is, the Empty Buffer) when computing the response HMAC.

24.6.2 Command and Response

Table 174 — TPM2_Clear Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Clear {NV E}
TPMI_RH_CLEAR	@authHandle	TPM_RH_LOCKOUT or TPM_RH_PLATFORM+{PP} Auth Handle: 1 Auth Role: USER

Table 175 — TPM2_Clear Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

24.6.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "Clear_fp.h"
3  #if CC_Clear  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_DISABLED	Clear command has been disabled

```
4  TPM_RC
5  TPM2_Clear(
6      Clear_In        *in             // IN: input parameter list
7      )
8  {
9      // Input parameter is not reference in command action
10     NOT_REFERENCED(in);
11
12     // The command needs NV update.  Check if NV is available.
13     // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
14     // this point
15     RETURN_IF_NV_IS_NOT_AVAILABLE;
16
17 // Input Validation
18
19     // If Clear command is disabled, return an error
20     if(gp.disableClear)
21         return TPM_RC_DISABLED;
22
23 // Internal Data Update
24
25     // Reset storage hierarchy seed from RNG
26     CryptRandomGenerate(sizeof(gp.SPSeed.t.buffer), gp.SPSeed.t.buffer);
27
28     // Create new shProof and ehProof value from RNG
29     CryptRandomGenerate(sizeof(gp.shProof.t.buffer), gp.shProof.t.buffer);
30     CryptRandomGenerate(sizeof(gp.ehProof.t.buffer), gp.ehProof.t.buffer);
31
32     // Enable storage and endorsement hierarchy
33     gc.shEnable = gc.ehEnable = TRUE;
34
35     // set the authValue buffers to zero
36     MemorySet(&gp.ownerAuth, 0, sizeof(gp.ownerAuth));
37     MemorySet(&gp.endorsementAuth, 0, sizeof(gp.endorsementAuth));
38     MemorySet(&gp.lockoutAuth, 0, sizeof(gp.lockoutAuth));
39
40     // Set storage, endorsement, and lockout authPolicy to null
41     gp.ownerAlg = gp.endorsementAlg = gp.lockoutAlg = TPM_ALG_NULL;
42     MemorySet(&gp.ownerPolicy, 0, sizeof(gp.ownerPolicy));
43     MemorySet(&gp.endorsementPolicy, 0, sizeof(gp.endorsementPolicy));
44     MemorySet(&gp.lockoutPolicy, 0, sizeof(gp.lockoutPolicy));
45
46     // Flush loaded object in storage and endorsement hierarchy
47     ObjectFlushHierarchy(TPM_RH_OWNER);
48     ObjectFlushHierarchy(TPM_RH_ENDORSEMENT);
49
50     // Flush owner and endorsement object and owner index in NV
51     NvFlushHierarchy(TPM_RH_OWNER);
52     NvFlushHierarchy(TPM_RH_ENDORSEMENT);
53
54     // Initialize dictionary attack parameters
55     DAPreInstall_Init();
56
```

 Copyright © TCG 2006-2020

```
57      // Reset clock
58      go.clock = 0;
59      go.clockSafe = YES;
60      NvWrite(NV_ORDERLY_DATA, sizeof(ORDERLY_DATA), &go);
61
62      // Reset counters
63      gp.resetCount = gr.restartCount = gr.clearCount = 0;
64      gp.auditCounter = 0;
65
66      // Save persistent data changes to NV
67      // Note: since there are so many changes to the persistent data structure, the
68      // entire PERSISTENT_DATA structure is written as a unit
69      NvWrite(NV_PERSISTENT_DATA, sizeof(PERSISTENT_DATA), &gp);
70
71      // Reset the PCR authValues (this does not change the PCRs)
72      PCR_ClearAuth();
73
74      // Bump the PCR counter
75      PCRChanged(0);
76
77      // orderly state should be cleared because of the update to state clear data
78      g_clearOrderly = TRUE;
79
80      return TPM_RC_SUCCESS;
81  }
82  #endif // CC_Clear
```

24.7 TPM2_ClearControl

24.7.1 General Description

TPM2_ClearControl() disables and enables the execution of TPM2_Clear().

The TPM will SET the TPM's TPMA_PERMANENT.*disableClear* attribute if *disable* is YES and will CLEAR the attribute if *disable* is NO. When the attribute is SET, TPM2_Clear() may not be executed.

NOTE This is to simplify the logic of TPM2_Clear(). TPM2_ClearControl() can be called using Platform Authorization to CLEAR the *disableClear* attribute and then execute TPM2_Clear().

Lockout Authorization may be used to SET *disableClear* but not to CLEAR it.

Platform Authorization may be used to SET or CLEAR *disableClear*.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

24.7.2 Command and Response

Table 176 — TPM2_ClearControl Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ClearControl {NV}
TPMI_RH_CLEAR	@auth	TPM_RH_LOCKOUT or TPM_RH_PLATFORM+{PP} Auth Handle: 1 Auth Role: USER
TPMI_YES_NO	disable	YES if the *disableOwnerClear* flag is to be SET, NO if the flag is to be CLEAR.

Table 177 — TPM2_ClearControl Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

24.7.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "ClearControl_fp.h"
3  #if CC_ClearControl  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_AUTH_FAIL	authorization is not properly given

```
4  TPM_RC
5  TPM2_ClearControl(
6      ClearControl_In     *in               // IN: input parameter list
7      )
8  {
9      // The command needs NV update.
10     RETURN_IF_NV_IS_NOT_AVAILABLE;
11
12 // Input Validation
13
14     // LockoutAuth may be used to set disableLockoutClear to TRUE but not to FALSE
15     if(in->auth == TPM_RH_LOCKOUT && in->disable == NO)
16         return TPM_RC_AUTH_FAIL;
17
18 // Internal Data Update
19
20     if(in->disable == YES)
21         gp.disableClear = TRUE;
22     else
23         gp.disableClear = FALSE;
24
25     // Record the change to NV
26     NV_SYNC_PERSISTENT(disableClear);
27
28     return TPM_RC_SUCCESS;
29 }
30 #endif // CC_ClearControl
```

 Copyright © TCG 2006-2020

24.8 TPM2_HierarchyChangeAuth

24.8.1 General Description

This command allows the authorization secret for a hierarchy or lockout to be changed using the current authorization value as the command authorization.

If *authHandle* is TPM_RH_PLATFORM, then *platformAuth* is changed. If *authHandle* is TPM_RH_OWNER, then o*wnerAuth* is changed. If *authHandle* is TPM_RH_ENDORSEMENT, then e*ndorsementAuth* is changed. If *authHandle* is TPM_RH_LOCKOUT, then *lockoutAuth* is changed. The HMAC in the response shall use the new authorization value when computing the response HMAC.

If *authHandle* is TPM_RH_PLATFORM, then Physical Presence may need to be asserted for this command to succeed (see 26.2, *TPM2_PP_Commands*).

The authorization value may be no larger than the digest produced by the hash algorithm used for context integrity.

EXAMPLE If SHA384 is used in the computation of the integrity values for saved contexts, then the largest authorization value is 48 octets.

 Copyright © TCG 2006-2020

24.8.2 Command and Response

Table 178 — TPM2_HierarchyChangeAuth Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_HierarchyChangeAuth {NV}
TPMI_RH_HIERARCHY_AUTH	@authHandle	TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPM2B_AUTH	newAuth	new authorization value

Table 179 — TPM2_HierarchyChangeAuth Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

24.8.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "HierarchyChangeAuth_fp.h"
3  #if CC_HierarchyChangeAuth  // Conditional expansion of this file
4  #include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_SIZE	*newAuth* size is greater than that of integrity hash digest

```
5  TPM_RC
6  TPM2_HierarchyChangeAuth(
7      HierarchyChangeAuth_In  *in              // IN: input parameter list
8      )
9  {
10     // The command needs NV update.
11     RETURN_IF_NV_IS_NOT_AVAILABLE;
12
13     // Make sure that the authorization value is a reasonable size (not larger than
14     // the size of the digest produced by the integrity hash. The integrity
15     // hash is assumed to produce the longest digest of any hash implemented
16     // on the TPM. This will also remove trailing zeros from the authValue.
17     if(MemoryRemoveTrailingZeros(&in->newAuth) > CONTEXT_INTEGRITY_HASH_SIZE)
18         return TPM_RCS_SIZE + RC_HierarchyChangeAuth_newAuth;
19
20     // Set hierarchy authValue
21     switch(in->authHandle)
22     {
23         case TPM_RH_OWNER:
24             gp.ownerAuth = in->newAuth;
25             NV_SYNC_PERSISTENT(ownerAuth);
26             break;
27         case TPM_RH_ENDORSEMENT:
28             gp.endorsementAuth = in->newAuth;
29             NV_SYNC_PERSISTENT(endorsementAuth);
30             break;
31         case TPM_RH_PLATFORM:
32             gc.platformAuth = in->newAuth;
33             // orderly state should be cleared
34             g_clearOrderly = TRUE;
35             break;
36         case TPM_RH_LOCKOUT:
37             gp.lockoutAuth = in->newAuth;
38             NV_SYNC_PERSISTENT(lockoutAuth);
39             break;
40         default:
41             FAIL(FATAL_ERROR_INTERNAL);
42             break;
43     }
44
45     return TPM_RC_SUCCESS;
46 }
47 #endif // CC_HierarchyChangeAuth
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

25 Dictionary Attack Functions

25.1 Introduction

A TPM is required to have support for logic that will help prevent a dictionary attack on an authorization value. The protection is provided by a counter that increments when a password authorization or an HMAC authorization fails. When the counter reaches a predefined value, the TPM will not accept, for some time interval, further requests that require authorization and the TPM is in Lockout mode. While the TPM is in Lockout mode, the TPM will return TPM_RC_LOCKOUT if the command requires use of an object's or Index's *authValue* unless the authorization applies to an entry in the Platform hierarchy.

NOTE 1 Authorizations for objects and NV Index values in the Platform hierarchy are never locked out. However, a command that requires multiple authorizations will not be accepted when the TPM is in Lockout mode unless all of the authorizations reference objects and indexes in the Platform hierarchy.

If the TPM is continuously powered for the duration of *newRecoveryTime* and no authorization failures occur, the authorization failure counter will be decremented by one. This property is called "self-healing." Self-healing shall not cause the count of failed attempts to decrement below zero.

The count of failed attempts, the lockout interval, and self-healing interval are settable using TPM2_DictionaryAttackParameters(). The lockout parameters and the current value of the lockout counter can be read with TPM2_GetCapability().

Dictionary attack protection does not apply to an entity associated with a permanent handle (handle type == TPM_HT_PERMANENT) other than TPM_RH_LOCKOUT

25.2 TPM2_DictionaryAttackLockReset

25.2.1 General Description

This command cancels the effect of a TPM lockout due to a number of successive authorization failures. If this command is properly authorized, the lockout counter is set to zero.

Only one *lockoutAuth* authorization failure is allowed for this command during a *lockoutRecovery* interval (set using TPM2_DictionaryAttackParameters().

 Copyright © TCG 2006-2020

25.2.2 Command and Response

Table 180 — TPM2_DictionaryAttackLockReset Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_DictionaryAttackLockReset {NV}
TPMI_RH_LOCKOUT	@lockHandle	TPM_RH_LOCKOUT Auth Index: 1 Auth Role: USER

Table 181 — TPM2_DictionaryAttackLockReset Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

25.2.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "DictionaryAttackLockReset_fp.h"
3  #if CC_DictionaryAttackLockReset  // Conditional expansion of this file
4  TPM_RC
5  TPM2_DictionaryAttackLockReset(
6      DictionaryAttackLockReset_In    *in             // IN: input parameter list
7      )
8  {
9      // Input parameter is not reference in command action
10     NOT_REFERENCED(in);
11
12     // The command needs NV update.
13     RETURN_IF_NV_IS_NOT_AVAILABLE;
14
15 // Internal Data Update
16
17     // Set failed tries to 0
18     gp.failedTries = 0;
19
20     // Record the changes to NV
21     NV_SYNC_PERSISTENT(failedTries);
22
23     return TPM_RC_SUCCESS;
24 }
25 #endif // CC_DictionaryAttackLockReset
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

25.3 TPM2_DictionaryAttackParameters

25.3.1 General Description

This command changes the lockout parameters.

The command requires Lockout Authorization.

The timeout parameters (*newRecoveryTime* and *lockoutRecovery*) indicate values that are measured with respect to the *Time* and not *Clock*.

NOTE Use of *Time* means that the TPM shall be continuously powered for the duration of a timeout.

If *newRecoveryTime* is zero, then DA protection is disabled. Authorizations are checked but authorization failures will not cause the TPM to enter lockout.

If *newMaxTries* is zero, the TPM will be in lockout and use of DA protected entities will be disabled.

If *lockoutRecovery* is zero, then the recovery interval is _TPM_Init followed by TPM2_Startup().

Only one *lockoutAuth* authorization failure is allowed for this command during a *lockoutRecovery* interval.

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

25.3.2 Command and Response

Table 182 — TPM2_DictionaryAttackParameters Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_DictionaryAttackParameters {NV}
TPMI_RH_LOCKOUT	@lockHandle	TPM_RH_LOCKOUT Auth Index: 1 Auth Role: USER
UINT32	newMaxTries	count of authorization failures before the lockout is imposed
UINT32	newRecoveryTime	time in seconds before the authorization failure count is automatically decremented A value of zero indicates that DA protection is disabled.
UINT32	lockoutRecovery	time in seconds after a *lockoutAuth* failure before use of *lockoutAuth* is allowed A value of zero indicates that a reboot is required.

Table 183 — TPM2_DictionaryAttackParameters Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

25.3.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "DictionaryAttackParameters_fp.h"
3  #if CC_DictionaryAttackParameters  // Conditional expansion of this file
4  TPM_RC
5  TPM2_DictionaryAttackParameters(
6      DictionaryAttackParameters_In   *in             // IN: input parameter list
7      )
8  {
9      // The command needs NV update.
10     RETURN_IF_NV_IS_NOT_AVAILABLE;
11
12 // Internal Data Update
13
14     // Set dictionary attack parameters
15     gp.maxTries = in->newMaxTries;
16     gp.recoveryTime = in->newRecoveryTime;
17     gp.lockoutRecovery = in->lockoutRecovery;
18
19 #if 0   // Errata eliminates this code
20     // This functionality has been disabled. The preferred implementation is now
21     // to leave failedTries unchanged when the parameters are changed. This could
22     // have the effect of putting the TPM into DA lockout if in->newMaxTries is
23     // not greater than the current value of gp.failedTries.
24     // Set failed tries to 0
25     gp.failedTries = 0;
26 #endif
27
28     // Record the changes to NV
29     NV_SYNC_PERSISTENT(failedTries);
30     NV_SYNC_PERSISTENT(maxTries);
31     NV_SYNC_PERSISTENT(recoveryTime);
32     NV_SYNC_PERSISTENT(lockoutRecovery);
33
34     return TPM_RC_SUCCESS;
35 }
36 #endif // CC_DictionaryAttackParameters
```

26 Miscellaneous Management Functions

26.1 Introduction

This clause contains commands that do not logically group with any other commands.

26.2 TPM2_PP_Commands

26.2.1 General Description

This command is used to determine which commands require assertion of Physical Presence (PP) in addition to *platformAuth*/*platformPolicy*.

This command requires that *auth* is TPM_RH_PLATFORM and that Physical Presence be asserted.

After this command executes successfully, the commands listed in *setList* will be added to the list of commands that require that Physical Presence be asserted when the handle associated with the authorization is TPM_RH_PLATFORM. The commands in *clearList* will no longer require assertion of Physical Presence in order to authorize a command.

If a command is not in either list, its state is not changed. If a command is in both lists, then it will no longer require Physical Presence (for example, *setList* is processed first).

Only commands with handle types of TPMI_RH_PLATFORM, TPMI_RH_PROVISION, TPMI_RH_CLEAR, or TPMI_RH_HIERARCHY can be gated with Physical Presence. If any other command is in either list, it is discarded.

When a command requires that Physical Presence be provided, then Physical Presence shall be asserted for either an HMAC or a Policy authorization.

NOTE 1 Physical Presence may be made a requirement of any policy.

NOTE 2 If the TPM does not implement this command, the command list is vendor specific. A platform-specific specification may require that the command list be initialized in a specific way.

TPM2_PP_Commands() always requires assertion of Physical Presence.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

26.2.2 Command and Response

Table 184 — TPM2_PP_Commands Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PP_Commands {NV}
TPMI_RH_PLATFORM	@auth	TPM_RH_PLATFORM+PP Auth Index: 1 Auth Role: USER + Physical Presence
TPML_CC	setList	list of commands to be added to those that will require that Physical Presence be asserted
TPML_CC	clearList	list of commands that will no longer require that Physical Presence be asserted

Table 185 — TPM2_PP_Commands Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

26.2.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "PP_Commands_fp.h"
3  #if CC_PP_Commands  // Conditional expansion of this file
4  TPM_RC
5  TPM2_PP_Commands(
6      PP_Commands_In  *in             // IN: input parameter list
7      )
8  {
9      UINT32          i;
10
11     // The command needs NV update.  Check if NV is available.
12     // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
13     // this point
14     RETURN_IF_NV_IS_NOT_AVAILABLE;
15
16 // Internal Data Update
17
18     // Process set list
19     for(i = 0; i < in->setList.count; i++)
20         // If command is implemented, set it as PP required.  If the input
21         // command is not a PP command, it will be ignored at
22         // PhysicalPresenceCommandSet().
23         // Note: PhysicalPresenceCommandSet() checks if the command is implemented.
24         PhysicalPresenceCommandSet(in->setList.commandCodes[i]);
25
26     // Process clear list
27     for(i = 0; i < in->clearList.count; i++)
28         // If command is implemented, clear it as PP required.  If the input
29         // command is not a PP command, it will be ignored at
30         // PhysicalPresenceCommandClear().  If the input command is
31         // TPM2_PP_Commands, it will be ignored as well
32         PhysicalPresenceCommandClear(in->clearList.commandCodes[i]);
33
34     // Save the change of PP list
35     NV_SYNC_PERSISTENT(ppList);
36
37     return TPM_RC_SUCCESS;
38 }
39 #endif // CC_PP_Commands
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

26.3 TPM2_SetAlgorithmSet

26.3.1 General Description

This command allows the platform to change the set of algorithms that are used by the TPM. The *algorithmSet* setting is a vendor-dependent value.

If the changing of the algorithm set results in a change of the algorithms of PCR banks, then the TPM will need to be reset (_TPM_Init and TPM2_Startup(TPM_SU_CLEAR)) before the new PCR settings take effect. After this command executes successfully, if *startupType* in the next TPM2_Startup() is not TPM_SU_CLEAR, the TPM shall return TPM_RC_VALUE and may enter Failure mode.

Other than PCR, when an algorithm is no longer supported, the behavior of this command is vendor-dependent.

EXAMPLE Entities may remain resident. Persistent objects, transient objects, or sessions may be flushed. NV Indexes may be undefined. Policies may be erased.

NOTE The reference implementation does not have support for this command. In particular, it does not support use of this command to selectively disable algorithms. Proper support would require modification of the unmarshaling code so that each time an algorithm is unmarshaled, it would be verified as being enabled.

 Copyright © TCG 2006-2020

26.3.2 Command and Response

Table 186 — TPM2_SetAlgorithmSet Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_SetAlgorithmSet {NV}
TPMI_RH_PLATFORM	@authHandle	TPM_RH_PLATFORM Auth Index: 1 Auth Role: USER
UINT32	algorithmSet	a TPM vendor-dependent value indicating the algorithm set selection

Table 187 — TPM2_SetAlgorithmSet Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

26.3.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "SetAlgorithmSet_fp.h"
3  #if CC_SetAlgorithmSet  // Conditional expansion of this file
4  TPM_RC
5  TPM2_SetAlgorithmSet(
6      SetAlgorithmSet_In  *in             // IN: input parameter list
7      )
8  {
9      // The command needs NV update.  Check if NV is available.
10     // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
11     // this point
12     RETURN_IF_NV_IS_NOT_AVAILABLE;
13
14 // Internal Data Update
15     gp.algorithmSet = in->algorithmSet;
16
17     // Write the algorithm set changes to NV
18     NV_SYNC_PERSISTENT(algorithmSet);
19
20     return TPM_RC_SUCCESS;
21 }
22 #endif // CC_SetAlgorithmSet
```

 Copyright © TCG 2006-2020

27 Field Upgrade

27.1 Introduction

This clause contains the commands for managing field upgrade of the firmware in the TPM. The field upgrade scheme may be used for replacement or augmentation of the firmware installed in the TPM.

EXAMPLE 1 If an algorithm is found to be flawed, a patch of that algorithm might be installed using the firmware upgrade process. The patch might be a replacement of a portion of the code or a complete replacement of the firmware.

EXAMPLE 2 If an additional set of ECC parameters is needed, the firmware process may be used to add the parameters to the TPM data set.

The field upgrade process uses two commands (TPM2_FieldUpgradeStart() and TPM2_FieldUpgradeData()). TPM2_FieldUpgradeStart() validates that a signature on the provided digest is from the TPM manufacturer and that proper authorization is provided using *platformPolicy*.

NOTE 1 The *platformPolicy* for field upgraded is defined by the PM and may include requirements that the upgrade be signed by the PM or the TPM owner and include any other constraints that are desired by the PM.

If the proper authorization is given, the TPM will retain the signed digest and enter the Field Upgrade mode (FUM). While in FUM, the TPM will accept TPM2_FieldUpgradeData() commands. It may accept other commands if it is able to complete them using the previously installed firmware. Otherwise, it will return TPM_RC_UPGRADE.

Each block of the field upgrade shall contain the digest of the next block of the field upgrade data. That digest shall be included in the digest of the previous block. The digest of the first block is signed by the TPM manufacturer. That signature and first block digest are the parameters for TPM2_FieldUpgradeStart(). The digest is saved in the TPM as the required digest for the next field upgrade data block and as the identifier of the field upgrade sequence.

For each field upgrade data block that is sent to the TPM by TPM2_FieldUpgradeData(), the TPM shall validate that the digest matches the required digest and if not, shall return TPM_RC_VALUE. The TPM shall extract the digest of the next expected block and return that value to the caller, along with the digest of the first data block of the update sequence.

The system may attempt to abandon the firmware upgrade by using a zero-length buffer in TPM2_FieldUpdateData(). If the TPM is able to resume operation using the firmware present when the upgrade started, then the TPM will indicate that it has abandon the update by setting the digest of the next block to the Empty Buffer. If the TPM cannot abandon the update, it will return the expected next digest.

The system may also attempt to abandon the update because of a power interruption. If the TPM is able to resume normal operations, then it will respond normally to TPM2_Startup(). If the TPM is not able to resume normal operations, then it will respond to any command but TPM2_FieldUpgradeData() with TPM_RC_UPGRADE.

After a _TPM_Init, system software may not be able to resume the field upgrade that was in process when the power interruption occurred. In such case, the TPM firmware may be reset to one of two other values:

- the original firmware that was installed at the factory (“initial firmware”); or
- the firmware that was in the TPM when the field upgrade process started (“previous firmware”).

The TPM retains the digest of the first block for these firmware images and checks to see if the first block after _TPM_Init matches either of those digests. If so, the firmware update process restarts and the original firmware may be loaded.

 Copyright © TCG 2006-2020

NOTE 2 The TPM is required to accept the previous firmware as either a vendor-provided update or as recovered from the TPM using TPM2_FirmwareRead().

When the last block of the firmware upgrade is loaded into the TPM (indicated to the TPM by data in the data block in a TPM vendor-specific manner), the TPM will complete the upgrade process. If the TPM is able to resume normal operations without a reboot, it will set the hash algorithm of the next block to TPM_ALG_NULL and return TPM_RC_SUCCESS. If a reboot is required, the TPM shall return TPM_RC_REBOOT in response to the last TPM2_FieldUpgradeData() and all subsequent TPM commands until a _TPM_Init is received.

NOTE 3 Because no additional data is allowed when the response code is not TPM_RC_SUCCESS, the TPM returns TPM_RC_SUCCESS for all calls to TPM2_FieldUpgradeData() except the last. In this manner, the TPM is able to indicate the digest of the next block. If a _TPM_Init occurs while the TPM is in FUM, the next block may be the digest for the first block of the original firmware. If it is not, then the TPM will not accept the original firmware until the next _TPM_Init when the TPM is in FUM.

During the field upgrade process, either the one specified in this clause or a vendor proprietary field upgrade process, the TPM should preserve:

- Primary Seeds;
- Hierarchy *authValue*, *authPolicy*, and *proof* values;
- Lockout *authValue* and authorization failure count values;
- PCR authValue and authPolicy values;
- NV Index allocations and contents;
- Persistent object allocations and contents; and
- Clock.

NOTE 4 A platform manufacturer may provide a means to change preserved data to accommodate a case where a field upgrade fixes a flaw that might have compromised TPM secrets.

27.2 TPM2_FieldUpgradeStart

27.2.1 General Description

This command uses *platformPolicy* and a TPM Vendor Authorization Key to authorize a Field Upgrade Manifest.

If the signature checks succeed, the authorization is valid and the TPM will accept TPM2_FieldUpgradeData().

This signature is checked against the loaded key referenced by *keyHandle*. This key will have a Name that is the same as a value that is part of the TPM firmware data. If the signature is not valid, the TPM shall return TPM_RC_SIGNATURE.

NOTE A loaded key is used rather than a hard-coded key to reduce the amount of memory needed for this key data in case more than one vendor key is needed.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

27.2.2 Command and Response

Table 188 — TPM2_FieldUpgradeStart Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_FieldUpgradeStart
TPMI_RH_PLATFORM	@authorization	TPM_RH_PLATFORM+{PP} Auth Index:1 Auth Role: ADMIN
TPMI_DH_OBJECT	keyHandle	handle of a public area that contains the TPM Vendor Authorization Key that will be used to validate *manifestSignature* Auth Index: None
TPM2B_DIGEST	fuDigest	digest of the first block in the field upgrade sequence
TPMT_SIGNATURE	manifestSignature	signature over *fuDigest* using the key associated with *keyHandle* (not optional)

Table 189 — TPM2_FieldUpgradeStart Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

27.2.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "FieldUpgradeStart_fp.h"
3  #if CC_FieldUpgradeStart  // Conditional expansion of this file
4  TPM_RC
5  TPM2_FieldUpgradeStart(
6      FieldUpgradeStart_In    *in              // IN: input parameter list
7      )
8  {
9      // Not implemented
10     UNUSED_PARAMETER(in);
11     return TPM_RC_SUCCESS;
12 }
13 #endif
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

27.3 TPM2_FieldUpgradeData

27.3.1 General Description

This command will take the actual field upgrade image to be installed on the TPM. The exact format of *fuData* is vendor-specific. This command is only possible following a successful TPM2_FieldUpgradeStart(). If the TPM has not received a properly authorized TPM2_FieldUpgradeStart(), then the TPM shall return TPM_RC_FIELDUPGRADE.

The TPM will validate that the digest of *fuData* matches an expected value. If so, the TPM may buffer or immediately apply the update. If the digest of *fuData* does not match an expected value, the TPM shall return TPM_RC_VALUE.

Copyright © TCG 2006-2020

27.3.2 Command and Response

Table 190 — TPM2_FieldUpgradeData Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_FieldUpgradeData {NV}
TPM2B_MAX_BUFFER	fuData	field upgrade image data

Table 191 — TPM2_FieldUpgradeData Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMT_HA+	nextDigest	tagged digest of the next block TPM_ALG_NULL if field update is complete
TPMT_HA	firstDigest	tagged digest of the first block of the sequence

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

27.3.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "FieldUpgradeData_fp.h"
3  #if CC_FieldUpgradeData  // Conditional expansion of this file
4  TPM_RC
5  TPM2_FieldUpgradeData(
6      FieldUpgradeData_In      *in,            // IN: input parameter list
7      FieldUpgradeData_Out     *out            // OUT: output parameter list
8      )
9  {
10     // Not implemented
11     UNUSED_PARAMETER(in);
12     UNUSED_PARAMETER(out);
13     return TPM_RC_SUCCESS;
14 }
15 #endif
```

27.4 TPM2_FirmwareRead

27.4.1 General Description

This command is used to read a copy of the current firmware installed in the TPM.

The presumption is that the data will be returned in reverse order so that the last block in the sequence would be the first block given to the TPM in case of a failure recovery. If the TPM2_FirmwareRead sequence completes successfully, then the data provided from the TPM will be sufficient to allow the TPM to recover from an abandoned upgrade of this firmware.

To start the sequence of retrieving the data, the caller sets *sequenceNumber* to zero. When the TPM has returned all the firmware data, the TPM will return the Empty Buffer as *fuData*.

The contents of *fuData* are opaque to the caller.

NOTE 1 The caller should retain the ordering of the update blocks so that the blocks sent to the TPM have the same size and inverse order as the blocks returned by a sequence of calls to this command.

NOTE 2 Support for this command is optional even if the TPM implements TPM2_FieldUpgradeStart() and TPM2_FieldUpgradeData().

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

27.4.2 Command and Response

Table 192 — TPM2_FirmwareRead Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_FirmwareRead
UINT32	sequenceNumber	the number of previous calls to this command in this sequence set to 0 on the first call

Table 193 — TPM2_FirmwareRead Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_MAX_BUFFER	fuData	field upgrade image data

 Copyright © TCG 2006-2020

27.4.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "FirmwareRead_fp.h"
3  #if CC_FirmwareRead  // Conditional expansion of this file
4  TPM_RC
5  TPM2_FirmwareRead(
6      FirmwareRead_In      *in,             // IN: input parameter list
7      FirmwareRead_Out     *out             // OUT: output parameter list
8      )
9  {
10     // Not implemented
11     UNUSED_PARAMETER(in);
12     UNUSED_PARAMETER(out);
13     return TPM_RC_SUCCESS;
14 }
15 #endif // CC_FirmwareRead
```

28 Context Management

28.1 Introduction

Three of the commands in this clause (TPM2_ContextSave(), TPM2_ContextLoad(), and TPM2_FlushContext()) implement the resource management described in the "Context Management" clause in TPM 2.0 Part 1.

The fourth command in this clause (TPM2_EvictControl()) is used to control the persistence of loadable objects in TPM memory. Background for this command may be found in the "Owner and Platform Evict Objects" clause in TPM 2.0 Part 1.

28.2 TPM2_ContextSave

28.2.1 General Description

This command saves a session context, object context, or sequence object context outside the TPM.

No authorization sessions of any type are allowed with this command and tag is required to be TPM_ST_NO_SESSIONS.

NOTE This preclusion avoids complex issues of dealing with the same session in *handle* and in the session area. While it might be possible to provide specificity, it would add unnecessary complexity to the TPM and, because this capability would provide no application benefit, use of authorization sessions for audit or encryption is prohibited.

The TPM shall encrypt and integrity protect the TPM2B_CONTEXT_SENSITIVE *context* as described in the "Context Protections" clause in TPM 2.0 Part 1.

See the “Context Data” clause in TPM 2.0 Part 2 for a description of the *context* structure in the response.

 Copyright © TCG 2006-2020

28.2.2 Command and Response

Table 194 — TPM2_ContextSave Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ContextSave
TPMI_DH_CONTEXT	saveHandle	handle of the resource to save Auth Index: None

Table 195 — TPM2_ContextSave Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMS_CONTEXT	context	

 Copyright © TCG 2006-2020

28.2.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "ContextSave_fp.h"
3  #if CC_ContextSave  // Conditional expansion of this file
4  #include "Context_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_CONTEXT_GAP	a *contextID* could not be assigned for a session context save
TPM_RC_TOO_MANY_CONTEXTS	no more contexts can be saved as the counter has maxed out

```
5  TPM_RC
6  TPM2_ContextSave(
7      ContextSave_In      *in,            // IN: input parameter list
8      ContextSave_Out     *out            // OUT: output parameter list
9      )
10 {
11     TPM_RC          result = TPM_RC_SUCCESS;
12     UINT16          fingerprintSize;    // The size of fingerprint in context
13     // blob.
14     UINT64          contextID = 0;      // session context ID
15     TPM2B_SYM_KEY   symKey;
16     TPM2B_IV        iv;
17
18     TPM2B_DIGEST    integrity;
19     UINT16          integritySize;
20     BYTE            *buffer;
21
22     // This command may cause the orderlyState to be cleared due to
23     // the update of state reset data. If the state is orderly and
24     // cannot be changed, exit early.
25     RETURN_IF_ORDERLY;
26
27 // Internal Data Update
28
29 // This implementation does not do things in quite the same way as described in
30 // Part 2 of the specification. In Part 2, it indicates that the
31 // TPMS_CONTEXT_DATA contains two TPM2B values. That is not how this is
32 // implemented. Rather, the size field of the TPM2B_CONTEXT_DATA is used to
33 // determine the amount of data in the encrypted data. That part is not
34 // independently sized. This makes the actual size 2 bytes smaller than
35 // calculated using Part 2. Since this is opaque to the caller, it is not
36 // necessary to fix. The actual size is returned by TPM2_GetCapabilities().
37
38     // Initialize output handle.  At the end of command action, the output
39     // handle of an object will be replaced, while the output handle
40     // for a session will be the same as input
41     out->context.savedHandle = in->saveHandle;
42
43     // Get the size of fingerprint in context blob.  The sequence value in
44     // TPMS_CONTEXT structure is used as the fingerprint
45     fingerprintSize = sizeof(out->context.sequence);
46
47     // Compute the integrity size at the beginning of context blob
48     integritySize = sizeof(integrity.t.size)
49         + CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
50
51 // Perform object or session specific context save
52     switch(HandleGetType(in->saveHandle))
53     {
54         case TPM_HT_TRANSIENT:
```

```
55          {
56              OBJECT              *object = HandleToObject(in->saveHandle);
57              ANY_OBJECT_BUFFER   *outObject;
58              UINT16               objectSize = ObjectIsSequence(object)
59                  ? sizeof(HASH_OBJECT) : sizeof(OBJECT);
60
61              outObject = (ANY_OBJECT_BUFFER *)(out->context.contextBlob.t.buffer
62                                                + integritySize + fingerprintSize);
63
64              // Set size of the context data.  The contents of context blob is vendor
65              // defined.  In this implementation, the size is size of integrity
66              // plus fingerprint plus the whole internal OBJECT structure
67              out->context.contextBlob.t.size = integritySize +
68                  fingerprintSize + objectSize;
69  #if ALG_RSA
70              // For an RSA key, make sure that the key has had the private exponent
71              // computed before saving.
72              if(object->publicArea.type == TPM_ALG_RSA &&
73                 !(object->attributes.publicOnly))
74                  CryptRsaLoadPrivateExponent(&object->publicArea, &object->sensitive);
75  #endif
76              // Make sure things fit
77              pAssert(out->context.contextBlob.t.size
78                      <= sizeof(out->context.contextBlob.t.buffer));
79              // Copy the whole internal OBJECT structure to context blob
80              MemoryCopy(outObject, object, objectSize);
81
82              // Increment object context ID
83              gr.objectContextID++;
84              // If object context ID overflows, TPM should be put in failure mode
85              if(gr.objectContextID == 0)
86                  FAIL(FATAL_ERROR_INTERNAL);
87
88              // Fill in other return values for an object.
89              out->context.sequence = gr.objectContextID;
90              // For regular object, savedHandle is 0x80000000.  For sequence object,
91              // savedHandle is 0x80000001.  For object with stClear, savedHandle
92              // is 0x80000002
93              if(ObjectIsSequence(object))
94              {
95                  out->context.savedHandle = 0x80000001;
96                  SequenceDataExport((HASH_OBJECT *)object,
97                                     (HASH_OBJECT_BUFFER *)outObject);
98              }
99              else
100                 out->context.savedHandle = (object->attributes.stClear == SET)
101                 ? 0x80000002 : 0x80000000;
102 // Get object hierarchy
103             out->context.hierarchy = ObjectGetHierarchy(object);
104
105             break;
106         }
107         case TPM_HT_HMAC_SESSION:
108         case TPM_HT_POLICY_SESSION:
109         {
110             SESSION         *session = SessionGet(in->saveHandle);
111
112             // Set size of the context data.  The contents of context blob is vendor
113             // defined.  In this implementation, the size of context blob is the
114             // size of a internal session structure plus the size of
115             // fingerprint plus the size of integrity
116             out->context.contextBlob.t.size = integritySize +
117                 fingerprintSize + sizeof(*session);
118
119             // Make sure things fit
120             pAssert(out->context.contextBlob.t.size
```

```
121                     < sizeof(out->context.contextBlob.t.buffer));
122
123             // Copy the whole internal SESSION structure to context blob.
124             // Save space for fingerprint at the beginning of the buffer
125             // This is done before anything else so that the actual context
126             // can be reclaimed after this call
127             pAssert(sizeof(*session) <= sizeof(out->context.contextBlob.t.buffer)
128                     - integritySize - fingerprintSize);
129             MemoryCopy(out->context.contextBlob.t.buffer + integritySize
130                        + fingerprintSize, session, sizeof(*session));
131            // Fill in the other return parameters for a session
132            // Get a context ID and set the session tracking values appropriately
133            // TPM_RC_CONTEXT_GAP is a possible error.
134            // SessionContextSave() will flush the in-memory context
135            // so no additional errors may occur after this call.
136             result = SessionContextSave(out->context.savedHandle, &contextID);
137             if(result != TPM_RC_SUCCESS)
138                 return result;
139             // sequence number is the current session contextID
140             out->context.sequence = contextID;
141
142             // use TPM_RH_NULL as hierarchy for session context
143             out->context.hierarchy = TPM_RH_NULL;
144
145             break;
146         }
147         default:
148             // SaveContext may only take an object handle or a session handle.
149             // All the other handle type should be filtered out at unmarshal
150             FAIL(FATAL_ERROR_INTERNAL);
151             break;
152     }
153
154     // Save fingerprint at the beginning of encrypted area of context blob.
155     // Reserve the integrity space
156     pAssert(sizeof(out->context.sequence) <=
157             sizeof(out->context.contextBlob.t.buffer) - integritySize);
158     MemoryCopy(out->context.contextBlob.t.buffer + integritySize,
159                &out->context.sequence, sizeof(out->context.sequence));
160
161     // Compute context encryption key
162     ComputeContextProtectionKey(&out->context, &symKey, &iv);
163
164     // Encrypt context blob
165     CryptSymmetricEncrypt(out->context.contextBlob.t.buffer + integritySize,
166                           CONTEXT_ENCRYPT_ALG, CONTEXT_ENCRYPT_KEY_BITS,
167                           symKey.t.buffer, &iv, ALG_CFB_VALUE,
168                           out->context.contextBlob.t.size - integritySize,
169                           out->context.contextBlob.t.buffer + integritySize);
170
171     // Compute integrity hash for the object
172     // In this implementation, the same routine is used for both sessions
173     // and objects.
174     ComputeContextIntegrity(&out->context, &integrity);
175
176     // add integrity at the beginning of context blob
177     buffer = out->context.contextBlob.t.buffer;
178     TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
179
180     // orderly state should be cleared because of the update of state reset and
181     // state clear data
182     g_clearOrderly = TRUE;
183
184     return result;
185 }
186 #endif // CC_ContextSave
```

28.3 TPM2_ContextLoad

28.3.1 General Description

This command is used to reload a context that has been saved by TPM2_ContextSave().

No authorization sessions of any type are allowed with this command and tag is required to be TPM_ST_NO_SESSIONS (see note in 28.2.1).

The TPM will return TPM_RC_HIERARCHY if the context is associated with a hierarchy that is disabled.

NOTE Contexts for authorization sessions and for sequence objects belong to the NULL hierarchy, which is never disabled.

See the “Context Data” clause in TPM 2.0 Part 2 for a description of the values in the *context* parameter.

If the integrity HMAC of the saved context is not valid, the TPM shall return TPM_RC_INTEGRITY.

The TPM shall perform a check on the decrypted context as described in the "Context Confidentiality Protection" clause of TPM 2.0 Part 1 and enter failure mode if the check fails.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

28.3.2 Command and Response

Table 196 — TPM2_ContextLoad Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ContextLoad
TPMS_CONTEXT	context	the context blob

Table 197 — TPM2_ContextLoad Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_DH_CONTEXT	loadedHandle	the handle assigned to the resource after it has been successfully loaded

 Copyright © TCG 2006-2020

28.3.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "ContextLoad_fp.h"
3  #if CC_ContextLoad  // Conditional expansion of this file
4  #include "Context_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_CONTEXT_GAP	there is only one available slot and this is not the oldest saved session context
TPM_RC_HANDLE	*context.savedHandle*' does not reference a saved session
TPM_RC_HIERARCHY	*context.hierarchy* is disabled
TPM_RC_INTEGRITY	*context* integrity check fail
TPM_RC_OBJECT_MEMORY	no free slot for an object
TPM_RC_SESSION_MEMORY	no free session slots
TPM_RC_SIZE	incorrect context blob size

```
5  TPM_RC
6  TPM2_ContextLoad(
7      ContextLoad_In       *in,            // IN: input parameter list
8      ContextLoad_Out      *out            // OUT: output parameter list
9      )
10 {
11     TPM_RC               result;
12     TPM2B_DIGEST         integrityToCompare;
13     TPM2B_DIGEST         integrity;
14     BYTE                 *buffer;    // defined to save some typing
15     INT32                size;       // defined to save some typing
16     TPM_HT               handleType;
17     TPM2B_SYM_KEY        symKey;
18     TPM2B_IV             iv;
19
20 // Input Validation
21
22 // See discussion about the context format in TPM2_ContextSave Detailed Actions
23
24     // IF this is a session context, make sure that the sequence number is
25     // consistent with the version in the slot
26
27     // Check context blob size
28     handleType = HandleGetType(in->context.savedHandle);
29
30     // Get integrity from context blob
31     buffer = in->context.contextBlob.t.buffer;
32     size = (INT32)in->context.contextBlob.t.size;
33     result = TPM2B_DIGEST_Unmarshal(&integrity, &buffer, &size);
34     if(result != TPM_RC_SUCCESS)
35         return result;
36
37     // the size of the integrity value has to match the size of digest produced
38     // by the integrity hash
39     if(integrity.t.size != CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG))
40         return TPM_RCS_SIZE + RC_ContextLoad_context;
41
42     // Make sure that the context blob has enough space for the fingerprint. This
43     // is elastic pants to go with the belt and suspenders we already have to make
44     // sure that the context is complete and untampered.
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

```
45      if((unsigned)size < sizeof(in->context.sequence))
46          return TPM_RCS_SIZE + RC_ContextLoad_context;
47
48      // After unmarshaling the integrity value, 'buffer' is pointing at the first
49      // byte of the integrity protected and encrypted buffer and 'size' is the number
50      // of integrity protected and encrypted bytes.
51
52      // Compute context integrity
53      ComputeContextIntegrity(&in->context, &integrityToCompare);
54
55      // Compare integrity
56      if(!MemoryEqual2B(&integrity.b, &integrityToCompare.b))
57          return TPM_RCS_INTEGRITY + RC_ContextLoad_context;
58      // Compute context encryption key
59      ComputeContextProtectionKey(&in->context, &symKey, &iv);
60
61      // Decrypt context data in place
62      CryptSymmetricDecrypt(buffer, CONTEXT_ENCRYPT_ALG, CONTEXT_ENCRYPT_KEY_BITS,
63                            symKey.t.buffer, &iv, ALG_CFB_VALUE, size, buffer);
64      // See if the fingerprint value matches. If not, it is symptomatic of either
65      // a broken TPM or that the TPM is under attack so go into failure mode.
66      if(!MemoryEqual(buffer, &in->context.sequence, sizeof(in->context.sequence)))
67          FAIL(FATAL_ERROR_INTERNAL);
68
69      // step over fingerprint
70      buffer += sizeof(in->context.sequence);
71
72      // set the remaining size of the context
73      size -= sizeof(in->context.sequence);
74
75      // Perform object or session specific input check
76      switch(handleType)
77      {
78          case TPM_HT_TRANSIENT:
79          {
80              OBJECT      *outObject;
81
82              if(size > (INT32)sizeof(OBJECT))
83                  FAIL(FATAL_ERROR_INTERNAL);
84
85              // Discard any changes to the handle that the TRM might have made
86              in->context.savedHandle = TRANSIENT_FIRST;
87
88              // If hierarchy is disabled, no object context can be loaded in this
89              // hierarchy
90              if(!HierarchyIsEnabled(in->context.hierarchy))
91                  return TPM_RCS_HIERARCHY + RC_ContextLoad_context;
92
93              // Restore object. If there is no empty space, indicate as much
94              outObject = ObjectContextLoad((ANY_OBJECT_BUFFER *)buffer,
95                                            &out->loadedHandle);
96              if(outObject == NULL)
97                  return TPM_RC_OBJECT_MEMORY;
98
99              break;
100         }
101         case TPM_HT_POLICY_SESSION:
102         case TPM_HT_HMAC_SESSION:
103         {
104             if(size != sizeof(SESSION))
105                 FAIL(FATAL_ERROR_INTERNAL);
106
107             // This command may cause the orderlyState to be cleared due to
108             // the update of state reset data.  If this is the case, check if NV is
109             // available first
110             RETURN_IF_ORDERLY;
```

```
111
112             // Check if input handle points to a valid saved session and that the
113             // sequence number makes sense
114             if(!SequenceNumberForSavedContextIsValid(&in->context))
115                 return TPM_RCS_HANDLE + RC_ContextLoad_context;
116
117             // Restore session.  A TPM_RC_SESSION_MEMORY, TPM_RC_CONTEXT_GAP error
118             // may be returned at this point
119             result = SessionContextLoad((SESSION_BUF *)buffer,
120                                         &in->context.savedHandle);
121             if(result != TPM_RC_SUCCESS)
122                 return result;
123
124             out->loadedHandle = in->context.savedHandle;
125
126             // orderly state should be cleared because of the update of state
127             // reset and state clear data
128             g_clearOrderly = TRUE;
129
130             break;
131         }
132         default:
133             // Context blob may only have an object handle or a session handle.
134             // All the other handle type should be filtered out at unmarshal
135             FAIL(FATAL_ERROR_INTERNAL);
136             break;
137     }
138
139     return TPM_RC_SUCCESS;
140 }
141 #endif // CC_ContextLoad
```

 Copyright © TCG 2006-2020

28.4 TPM2_FlushContext

28.4.1 General Description

This command causes all context associated with a loaded object, sequence object, or session to be removed from TPM memory.

This command may not be used to remove a persistent object from the TPM. Use TPM2_EvictControl to remove a persistent object.

A session does not have to be loaded in TPM memory to have its context flushed. The saved session context associated with the indicated handle is invalidated. When flushing a session, the upper byte of the handle is ignored.

EXAMPLE A command to flush session handle 0x20000000 will flush session handle 0x03000000.

No sessions of any type are allowed with this command and tag is required to be TPM_ST_NO_SESSIONS (see note in 28.2.1).

If the handle is for a Transient Object and the handle is not associated with a loaded object, then the TPM shall return TPM_RC_HANDLE.

If the handle is for an authorization session and the handle does not reference a loaded or active session, then the TPM shall return TPM_RC_HANDLE.

NOTE *flushHandle* is a parameter and not a handle. If it were in the handle area, the TPM would validate that the context for the referenced entity is in the TPM. When a TPM2_FlushContext references a saved session context, it is not necessary for the context to be in the TPM. When the *flushHandle* is in the parameter area, the TPM does not validate that associated context is actually in the TPM.

 Copyright © TCG 2006-2020

28.4.2 Command and Response

Table 198 — TPM2_FlushContext Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_FlushContext
TPMI_DH_CONTEXT	flushHandle	the handle of the item to flush NOTE This is a use of a handle as a parameter.

Table 199 — TPM2_FlushContext Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

28.4.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "FlushContext_fp.h"
3  #if CC_FlushContext  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_HANDLE	*flushHandle* does not reference a loaded object or session

```
4  TPM_RC
5  TPM2_FlushContext(
6      FlushContext_In     *in              // IN: input parameter list
7      )
8  {
9  // Internal Data Update
10
11     // Call object or session specific routine to flush
12     switch(HandleGetType(in->flushHandle))
13     {
14         case TPM_HT_TRANSIENT:
15             if(!IsObjectPresent(in->flushHandle))
16                 return TPM_RCS_HANDLE + RC_FlushContext_flushHandle;
17            // Flush object
18             FlushObject(in->flushHandle);
19             break;
20         case TPM_HT_HMAC_SESSION:
21         case TPM_HT_POLICY_SESSION:
22             if(!SessionIsLoaded(in->flushHandle)
23                && !SessionIsSaved(in->flushHandle)
24                )
25                 return TPM_RCS_HANDLE + RC_FlushContext_flushHandle;
26
27             // If the session to be flushed is the exclusive audit session, then
28             // indicate that there is no exclusive audit session any longer.
29             if(in->flushHandle == g_exclusiveAuditSession)
30                 g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
31
32             // Flush session
33             SessionFlush(in->flushHandle);
34             break;
35         default:
36             // This command only takes object or session handle.  Other handles
37             // should be filtered out at handle unmarshal
38             FAIL(FATAL_ERROR_INTERNAL);
39             break;
40     }
41
42     return TPM_RC_SUCCESS;
43 }
44 #endif // CC_FlushContext
```

28.5 TPM2_EvictControl

28.5.1 General Description

This command allows certain Transient Objects to be made persistent or a persistent object to be evicted.

NOTE 1 A transient object is one that may be removed from TPM memory using either TPM2_FlushContext or TPM2_Startup(). A persistent object is not removed from TPM memory by TPM2_FlushContext() or TPM2_Startup().

If *objectHandle* is a Transient Object, then this call makes a persistent copy of the object and assigns *persistentHandle* to the persistent version of the object. If *objectHandle* is a persistent object, then the call evicts the persistent object. The call does not affect the transient object.

Before execution of TPM2_EvictControl code below, the TPM verifies that *objectHandle* references an object that is resident on the TPM and that *persistentHandle* is a valid handle for a persistent object.

NOTE 2 This requirement simplifies the unmarshaling code so that it only need check that *persistentHandle* is always a persistent object.

If *objectHandle* references a Transient Object:

a) The TPM shall return TPM_RC_ATTRIBUTES if
   1) it is in the hierarchy of TPM_RH_NULL,
   2) only the public portion of the object is loaded, or

      NOTE 3 This is for NV space efficiency. Loading an object whose private part is empty would unnecessarily consume NV resources.

   3) the *stClear* is SET in the object or in an ancestor key.
b) The TPM shall return TPM_RC_HIERARCHY if the object is not in the proper hierarchy as determined by *auth*.
   1) If *auth* is TPM_RH_PLATFORM, the proper hierarchy is the Platform hierarchy.
   2) If *auth* is TPM_RH_OWNER, the proper hierarchy is either the Storage or the Endorsement hierarchy.
c) The TPM shall return TPM_RC_RANGE if *persistentHandle* is not in the proper range as determined by *auth*.
   1) If *auth* is TPM_RH_OWNER, then *persistentHandle* shall be in the inclusive range of 81 00 00 00_{16} to 81 7F FF FF_{16}.
   2) If *auth* is TPM_RH_PLATFORM, then *persistentHandle* shall be in the inclusive range of 81 80 00 00_{16} to 81 FF FF FF_{16}.

   NOTE 4 This separation permits the platform (the platform OEM) a range of indexes that will not interfere with indexes used by the TPM owner (the OS or applications).

d) The TPM shall return TPM_RC_NV_DEFINED if a persistent object exists with the same handle as *persistentHandle*.
e) The TPM shall return TPM_RC_NV_SPACE if insufficient space is available to make the object persistent.
f) The TPM shall return TPM_RC_NV_SPACE if execution of this command will prevent the TPM from being able to hold two transient objects of any kind.

   NOTE 5 This requirement anticipates that a TPM may be implemented such that all TPM memory is non-volatile and not subject to endurance issues. In such case, there is no movement of an object

 Copyright © TCG 2006-2020

between memory of different types and it is necessary that the TPM ensure that it is always possible for the management software to move objects to/from TPM memory in order to ensure that the objects required for command execution can be context restored.

g) If the TPM returns TPM_RC_SUCCESS, the object referenced by *objectHandle* will not be flushed and both *objectHandle* and *persistentHandle* may be used to access the object.

If *objectHandle* references a persistent object:

a) The TPM shall return TPM_RC_RANGE if *objectHandle* is not in the proper range as determined by *auth*. If *auth* is TPM_RC_OWNER, *objectHandle* shall be in the inclusive range of 81 00 00 00_{16} to 81 7F FF FF_{16}. If *auth* is TPM_RC_PLATFORM, *objectHandle* may be any valid persistent object handle.

b) If objectHandle is not the same value as persistentHandle, return TPM_RC_HANDLE.

c) If the TPM returns TPM_RC_SUCCESS, *objectHandle* will be removed from persistent memory and no longer be accessible.

NOTE 5 The persistent object is not converted to a transient object, as this would prevent the immediate revocation of an object by removing it from persistent memory.

 Copyright © TCG 2006-2020

28.5.2 Command and Response

Table 200 — TPM2_EvictControl Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_EvictControl {NV}
TPMI_RH_PROVISION	@auth	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Handle: 1 Auth Role: USER
TPMI_DH_OBJECT	objectHandle	the handle of a loaded object Auth Index: None
TPMI_DH_PERSISTENT	persistentHandle	if *objectHandle* is a transient object handle, then this is the persistent handle for the object if *objectHandle* is a persistent object handle, then it shall be the same value as *persistentHandle*

Table 201 — TPM2_EvictControl Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

28.5.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "EvictControl_fp.h"
3  #if CC_EvictControl  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	an object with *temporary*, *stClear* or *publicOnly* attribute SET cannot be made persistent
TPM_RC_HIERARCHY	*auth* cannot authorize the operation in the hierarchy of *evictObject*
TPM_RC_HANDLE	*evictHandle* of the persistent object to be evicted is not the same as the *persistentHandle* argument
TPM_RC_NV_HANDLE	*persistentHandle* is unavailable
TPM_RC_NV_SPACE	no space in NV to make *evictHandle* persistent
TPM_RC_RANGE	*persistentHandle* is not in the range corresponding to the hierarchy of *evictObject*

```
4  TPM_RC
5  TPM2_EvictControl(
6      EvictControl_In     *in             // IN: input parameter list
7      )
8  {
9      TPM_RC      result;
10     OBJECT      *evictObject;
11
12 // Input Validation
13
14     // Get internal object pointer
15     evictObject = HandleToObject(in->objectHandle);
16
17     // Temporary, stClear or public only objects can not be made persistent
18     if(evictObject->attributes.temporary == SET
19        || evictObject->attributes.stClear == SET
20        || evictObject->attributes.publicOnly == SET)
21         return TPM_RCS_ATTRIBUTES + RC_EvictControl_objectHandle;
22
23     // If objectHandle refers to a persistent object, it should be the same as
24     // input persistentHandle
25     if(evictObject->attributes.evict == SET
26        && evictObject->evictHandle != in->persistentHandle)
27         return TPM_RCS_HANDLE + RC_EvictControl_objectHandle;
28
29     // Additional authorization validation
30     if(in->auth == TPM_RH_PLATFORM)
31     {
32         // To make persistent
33         if(evictObject->attributes.evict == CLEAR)
34         {
35             // PlatformAuth can not set evict object in storage or endorsement
36             // hierarchy
37             if(evictObject->attributes.ppsHierarchy == CLEAR)
38                 return TPM_RCS_HIERARCHY + RC_EvictControl_objectHandle;
39             // Platform cannot use a handle outside of platform persistent range.
40             if(!NvIsPlatformPersistentHandle(in->persistentHandle))
41                 return TPM_RCS_RANGE + RC_EvictControl_persistentHandle;
42         }
43         // PlatformAuth can delete any persistent object
44     }
```

```
45      else if(in->auth == TPM_RH_OWNER)
46      {
47          // OwnerAuth can not set or clear evict object in platform hierarchy
48          if(evictObject->attributes.ppsHierarchy == SET)
49              return TPM_RCS_HIERARCHY + RC_EvictControl_objectHandle;
50
51          // Owner cannot use a handle outside of owner persistent range.
52          if(evictObject->attributes.evict == CLEAR
53             && !NvIsOwnerPersistentHandle(in->persistentHandle))
54              return TPM_RCS_RANGE + RC_EvictControl_persistentHandle;
55      }
56      else
57      {
58          // Other authorization is not allowed in this command and should have been
59          // filtered out in unmarshal process
60          FAIL(FATAL_ERROR_INTERNAL);
61      }
62  // Internal Data Update
63      // Change evict state
64      if(evictObject->attributes.evict == CLEAR)
65      {
66          // Make object persistent
67          if(NvFindHandle(in->persistentHandle) != 0)
68              return TPM_RC_NV_DEFINED;
69          // A TPM_RC_NV_HANDLE or TPM_RC_NV_SPACE error may be returned at this
70          // point
71          result = NvAddEvictObject(in->persistentHandle, evictObject);
72      }
73      else
74      {
75          // Delete the persistent object in NV
76          result = NvDeleteEvict(evictObject->evictHandle);
77      }
78      return result;
79  }
80  #endif // CC_EvictControl
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

29 Clocks and Timers

29.1 TPM2_ReadClock

29.1.1 General Description

This command reads the current TPMS_TIME_INFO structure that contains the current setting of *Time*, *Clock*, *resetCount*, and *restartCount*.

29.1.2 Command and Response

Table 202 — TPM2_ReadClock Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ReadClock

Table 203 — TPM2_ReadClock Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMS_TIME_INFO	currentTime	

 Copyright © TCG 2006-2020

29.1.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "ReadClock_fp.h"
3   #if CC_ReadClock  // Conditional expansion of this file
4   TPM_RC
5   TPM2_ReadClock(
6       ReadClock_Out   *out            // OUT: output parameter list
7       )
8   {
9   // Command Output
10
11      out->currentTime.time = g_time;
12      TimeFillInfo(&out->currentTime.clockInfo);
13
14      return TPM_RC_SUCCESS;
15  }
16  #endif // CC_ReadClock
```

29.2 TPM2_ClockSet

29.2.1 General Description

This command is used to advance the value of the TPM's *Clock*. The command will fail if *newTime* is less than the current value of *Clock* or if the new time is greater than FF FF 00 00 00 00 00 00_{16}. If both of these checks succeed, *Clock* is set to *newTime*. If either of these checks fails, the TPM shall return TPM_RC_VALUE and make no change to *Clock*.

NOTE This maximum setting would prevent *Clock* from rolling over to zero for approximately 8,000 years at the real time *Clock* update rate. If the *Clock* update rate was set so that TPM time was passing 33 percent faster than real time, it would still be more than 6,000 years before *Clock* would roll over to zero. Because *Clock* will not roll over in the lifetime of the TPM, there is no need for external software to deal with the possibility that *Clock* may wrap around.

If the value of *Clock* after the update makes the volatile and non-volatile versions of TPMS_CLOCK_INFO.*clock* differ by more than the reported update interval, then the TPM shall update the non-volatile version of TPMS_CLOCK_INFO.*clock* before returning.

This command requires Platform Authorization or Owner Authorization.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

29.2.2 Command and Response

Table 204 — TPM2_ClockSet Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ClockSet {NV}
TPMI_RH_PROVISION	@auth	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Handle: 1 Auth Role: USER
UINT64	newTime	new *Clock* setting in milliseconds

Table 205 — TPM2_ClockSet Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

29.2.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "ClockSet_fp.h"
3   #if CC_ClockSet  // Conditional expansion of this file
```

Read the current TPMS_TIMER_INFO structure settings

Error Returns	Meaning
TPM_RC_NV_RATE	NV is unavailable because of rate limit
TPM_RC_NV_UNAVAILABLE	NV is inaccessible
TPM_RC_VALUE	invalid new clock

```
4   TPM_RC
5   TPM2_ClockSet(
6       ClockSet_In     *in             // IN: input parameter list
7       )
8   {
9   // Input Validation
10      // new time can not be bigger than 0xFFFF000000000000 or smaller than
11      // current clock
12      if(in->newTime > 0xFFFF000000000000ULL
13         || in->newTime < go.clock)
14          return TPM_RCS_VALUE + RC_ClockSet_newTime;
15
16  // Internal Data Update
17      // Can't modify the clock if NV is not available.
18      RETURN_IF_NV_IS_NOT_AVAILABLE;
19
20      TimeClockUpdate(in->newTime);
21      return TPM_RC_SUCCESS;
22  }
23  #endif // CC_ClockSet
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

29.3 TPM2_ClockRateAdjust

29.3.1 General Description

This command adjusts the rate of advance of *Clock* and *Time* to provide a better approximation to real time.

The *rateAdjust* value is relative to the current rate and not the nominal rate of advance.

EXAMPLE 1 If this command had been called three times with *rateAdjust* = TPM_CLOCK_COARSE_SLOWER and once with *rateAdjust* = TPM_CLOCK_COARSE_FASTER, the net effect will be as if the command had been called twice with *rateAdjust* = TPM_CLOCK_COARSE_SLOWER.

The range of adjustment shall be sufficient to allow *Clock* and *Time* to advance at real time but no more. If the requested adjustment would make the rate advance faster or slower than the nominal accuracy of the input frequency, the TPM shall return TPM_RC_VALUE.

EXAMPLE 2 If the frequency tolerance of the TPM's input clock is +/-10 percent, then the TPM will return TPM_RC_VALUE if the adjustment would make *Clock* run more than 10 percent faster or slower than nominal. That is, if the input oscillator were nominally 100 megahertz (MHz), then 1 millisecond (ms) would normally take 100,000 counts. The update *Clock* should be adjustable so that 1 ms is between 90,000 and 110,000 counts.

The interpretation of “fine” and “coarse” adjustments is implementation-specific.

The nominal rate of advance for *Clock* and *Time* shall be accurate to within 15 percent. That is, with no adjustment applied, *Clock* and *Time* shall be advanced at a rate within 15 percent of actual time.

NOTE If the adjustments are incorrect, it will be possible to make the difference between advance of *Clock/Time* and real time to be as much as 1.15^2 or ~1.33.

Changes to the current *Clock* update rate adjustment need not be persisted across TPM power cycles.

 Copyright © TCG 2006-2020

29.3.2 Command and Response

Table 206 — TPM2_ClockRateAdjust Command

Type	**Name**	**Description**
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ClockRateAdjust
TPMI_RH_PROVISION	@auth	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Handle: 1 Auth Role: USER
TPM_CLOCK_ADJUST	rateAdjust	Adjustment to current *Clock* update rate

Table 207 — TPM2_ClockRateAdjust Response

Type	**Name**	**Description**
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

29.3.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "ClockRateAdjust_fp.h"
3  #if CC_ClockRateAdjust  // Conditional expansion of this file
4  TPM_RC
5  TPM2_ClockRateAdjust(
6      ClockRateAdjust_In  *in              // IN: input parameter list
7      )
8  {
9  // Internal Data Update
10     TimeSetAdjustRate(in->rateAdjust);
11
12     return TPM_RC_SUCCESS;
13 }
14 #endif // CC_ClockRateAdjust
```

30 Capability Commands

30.1 Introduction

The TPM has numerous values that indicate the state, capabilities, and properties of the TPM. These values are needed for proper management of the TPM. The TPM2_GetCapability() command is used to access these values.

TPM2_GetCapability() allows reporting of multiple values in a single call. The values are grouped according to type.

NOTE TPM2_TestParms()is used to determine if a TPM supports a particular combination of algorithm parameters

30.2 TPM2_GetCapability

30.2.1 General Description

This command returns various information regarding the TPM and its current state.

The *capability* parameter determines the category of data returned. The *property* parameter selects the first value of the selected category to be returned. If there is no property that corresponds to the value of *property*, the next higher value is returned, if it exists.

EXAMPLE 1 The list of handles of transient objects currently loaded in the TPM may be read one at a time. On the first read, set the property to TRANSIENT_FIRST and *propertyCount* to one. If a transient object is present, the lowest numbered handle is returned and *moreData* will be YES if transient objects with higher handles are loaded. On the subsequent call, use returned handle value plus 1 in order to access the next higher handle.

The *propertyCount* parameter indicates the number of capabilities in the indicated group that are requested. The TPM will return no more than the number of requested values (*propertyCount*) or until the last property of the requested type has been returned.

NOTE 1 The type of the capability is derived from a combination of *capability* and *property*.

NOTE 2 If the *property* selects an unimplemented property, the next higher implemented property is returned.

When all of the properties of the requested type have been returned, the *moreData* parameter in the response will be set to NO. Otherwise, it will be set to YES.

NOTE 3 The *moreData* parameter will be YES if there are more properties even if the requested number of capabilities has been returned.

The TPM is not required to return more than one value at a time. It is not required to provide the same number of values in response to subsequent requests.

EXAMPLE 2 A TPM may return 4 properties in response to a TPM2_GetCapability(*capability* = TPM_CAP_TPM_PROPERTY, *property* = TPM_PT_MANUFACTURER, *propertyCount* = 8) and for a latter request with the same parameters, the TPM may return as few as one and as many as 8 values.

When the TPM is in Failure mode, a TPM is required to allow use of this command for access of the following capabilities:

 Copyright © TCG 2006-2020

- TPM_PT_MANUFACTURER
- TPM_PT_VENDOR_STRING_1
- TPM_PT_VENDOR_STRING_2 (NOTE 4)
- TPM_PT_VENDOR_STRING_3 (NOTE 4)
- TPM_PT_VENDOR_STRING_4 (NOTE 4)
- TPM_PT_VENDOR_TPM_TYPE
- TPM_PT_FIRMWARE_VERSION_1
- TPM_PT_FIRMWARE_VERSION_2

NOTE 4 If the vendor string does not require one of these values, the property type does not need to exist.

A vendor may optionally allow the TPM to return other values.

If in Failure mode and a capability is requested that is not available in Failure mode, the TPM shall return no value.

EXAMPLE 3 Assume the TPM is in Failure mode and the TPM only supports reporting of the minimum required set of properties (the limited subset of TPML_TAGGED_TPM_PROPERTY values). If a TPM2_GetCapability is received requesting a capability that has a property type value greater than TPM_PT_FIRMWARE_VERSION_2, the TPM may return a zero length list with the moreData parameter set to NO or return the property TPM_PT_FIRMWARE_VERSION_2. If the property type is less than TPM_PT_MANUFACTURER, the TPM will return properties beginning with TPM_PT_MANUFACTURER.

In Failure mode, *tag* is required to be TPM_ST_NO_SESSIONS or the TPM shall return TPM_RC_FAILURE.

The capability categories and the types of the return values are:

capability	***property***	**Return Type**
TPM_CAP_ALGS	TPM_ALG_ID(1)	TPML_ALG_PROPERTY
TPM_CAP_HANDLES	TPM_HANDLE	TPML_HANDLE
TPM_CAP_COMMANDS	TPM_CC	TPML_CCA
TPM_CAP_PP_COMMANDS	TPM_CC	TPML_CC
TPM_CAP_AUDIT_COMMANDS	TPM_CC	TPML_CC
TPM_CAP_PCRS	Reserved	TPML_PCR_SELECTION
TPM_CAP_TPM_PROPERTIES	TPM_PT	TPML_TAGGED_TPM_PROPERTY
TPM_CAP_PCR_PROPERTIES	TPM_PT_PCR	TPML_TAGGED_PCR_PROPERTY
TPM_CAP_ECC_CURVES	TPM_ECC_CURVE(1)	TPML_ECC_CURVE
TPM_CAP_AUTH_POLICIES (3)	TPM_HANDLE(2)	TPML_TAGGED_POLICY
TPM_CAP_ACT(4)	TPM_HANDLE(2)	TPML_ACT_DATA
TPM_CAP_VENDOR_PROPERTY	manufacturer specific	manufacturer-specific values
NOTES: (1) The TPM_ALG_ID or TPM_ECC_CURVE is cast to a UINT32 (2) The TPM will return TPM_RC_VALUE if the handle does not reference the range for permanent handles. (3) TPM_CAP_AUTH_POLICIES was added in revision 01.32. (4) TPM_CAP_ACT was added in revision 01.56.		

 Copyright © TCG 2006-2020

- TPM_CAP_ALGS – Returns a list of TPMS_ALG_PROPERTIES. Each entry is an algorithm ID and a set of properties of the algorithm.
- TPM_CAP_HANDLES – Returns a list of all of the handles within the handle range of the *property* parameter. The range of the returned handles is determined by the handle type (the most-significant octet (MSO) of the *property)*. Any of the defined handle types is allowed

EXAMPLE 4 If the MSO of *property* is TPM_HT_NV_INDEX, then the TPM will return a list of NV Index values.

EXAMPLE 5 If the MSO of property is TPM_HT_PCR, then the TPM will return a list of PCR.

- For this capability, use of TPM_HT_LOADED_SESSION and TPM_HT_SAVED_SESSION is allowed. Requesting handles with a handle type of TPM_HT_LOADED_SESSION will return handles for loaded sessions. The returned handle values will have a handle type of either TPM_HT_HMAC_SESSION or TPM_HT_POLICY_SESSION. If saved sessions are requested, all returned values will have the TPM_HT_HMAC_SESSION handle type because the TPM does not track the session type of saved sessions.

NOTE 5 TPM_HT_LOADED_SESSION and TPM_HT_HMAC_SESSION have the same value, as do TPM_HT_SAVED_SESSION and TPM_HT_POLICY_SESSION. It is not possible to request that the TPM return a list of loaded HMAC sessions without including the policy sessions.

- TPM_CAP_COMMANDS – Returns a list of the command attributes for all of the commands implemented in the TPM, starting with the TPM_CC indicated by the *property* parameter. If vendor specific commands are implemented, the vendor-specific command attribute with the lowest *commandIndex*, is returned after the non-vendor-specific (base) command.

NOTE 6 The type of the property parameter is a TPM_CC while the type of the returned list is TPML_CCA.

- TPM_CAP_PP_COMMANDS – Returns a list of all of the commands currently requiring Physical Presence for confirmation of platform authorization. The list will start with the TPM_CC indicated by *property*.
- TPM_CAP_AUDIT_COMMANDS – Returns a list of all of the commands currently set for command audit.
- TPM_CAP_PCRS – Returns the current allocation of PCR in a TPML_PCR_SELECTION. The *property* parameter shall be zero. The TPM will always respond to this command with the full PCR allocation and *moreData* will be NO.

  The TPML_PCR_SELECTION must include a TPMS_PCR_SELECTION for each PCR bank in which there is at least one allocated PCR. The TPML_PCR_SELECTION may return a TPMS_PCR_SELECTION for each implemented PCR bank. The TPML_PCR_SELECTION may return a TPMS_PCR_SELECTION for each implemented hash algorithm.
- TPM_CAP_TPM_PROPERTIES – Returns a list of tagged properties. The tag is a TPM_PT and the property is a 32-bit value. The properties are returned in groups. Each property group is on a 256-value boundary (that is, the boundary occurs when the TPM_PT is evenly divisible by 256). The TPM will only return values in the same group as the *property* parameter in the command.
- TPM_CAP_PCR_PROPERTIES – Returns a list of tagged PCR properties. The tag is a TPM_PT_PCR and the property is a TPMS_PCR_SELECT.

  The input command property is a TPM_PT_PCR (see TPM 2.0 Part 2 for PCR properties to be requested) that specifies the first property to be returned. If propertyCount is greater than 1, the list of properties begins with that property and proceeds in TPM_PT_PCR sequence.

  Each item in the list is a TPMS_PCR_SELECT structure that contains a bitmap of all PCR.

NOTE 7 A PCR index in all banks (all hash algorithms) has the same properties, so the hash algorithm is not specified here.

 Copyright © TCG 2006-2020

- TPM_CAP_TPM_ECC_CURVES – Returns a list of ECC curve identifiers currently available for use in the TPM.
- TPM_CAP_AUTH_POLICIES - Returns a list of tagged policies reporting the authorization policies for the permanent handles.
- TPM_CAP_ACT – Returns a list of TPMS_ACT_DATA, each of which contains the handle for the ACT, the remaining time before it expires, and the ACT attributes.

The *moreData* parameter will have a value of YES if there are more values of the requested type that were not returned.

If no next capability exists, the TPM will return a zero-length list and *moreData* will have a value of NO.

 Copyright © TCG 2006-2020

30.2.2 Command and Response

Table 208 — TPM2_GetCapability Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_GetCapability
TPM_CAP	capability	group selection; determines the format of the response
UINT32	property	further definition of information
UINT32	propertyCount	number of properties of the indicated type to return

Table 209 — TPM2_GetCapability Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_YES_NO	moreData	flag to indicate if there are more values of this type
TPMS_CAPABILITY_DATA	capabilityData	the capability data

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

30.2.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "GetCapability_fp.h"
3  #if CC_GetCapability  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_HANDLE	value of *property* is in an unsupported handle range for the TPM_CAP_HANDLES *capability* value
TPM_RC_VALUE	invalid *capability*; or *property* is not 0 for the TPM_CAP_PCRS *capability* value

```
4  TPM_RC
5  TPM2_GetCapability(
6      GetCapability_In     *in,            // IN: input parameter list
7      GetCapability_Out    *out            // OUT: output parameter list
8      )
9  {
10     TPMU_CAPABILITIES    *data = &out->capabilityData.data;
11 // Command Output
12
13     // Set output capability type the same as input type
14     out->capabilityData.capability = in->capability;
15
16     switch(in->capability)
17     {
18         case TPM_CAP_ALGS:
19             out->moreData = AlgorithmCapGetImplemented((TPM_ALG_ID)in->property,
20                                                        in->propertyCount,
21                                                        &data->algorithms);
22             break;
23         case TPM_CAP_HANDLES:
24             switch(HandleGetType((TPM_HANDLE)in->property))
25             {
26                 case TPM_HT_TRANSIENT:
27                     // Get list of handles of loaded transient objects
28                     out->moreData = ObjectCapGetLoaded((TPM_HANDLE)in->property,
29                                                        in->propertyCount,
30                                                        &data->handles);
31                     break;
32                 case TPM_HT_PERSISTENT:
33                     // Get list of handles of persistent objects
34                     out->moreData = NvCapGetPersistent((TPM_HANDLE)in->property,
35                                                        in->propertyCount,
36                                                        &data->handles);
37                     break;
38                 case TPM_HT_NV_INDEX:
39                     // Get list of defined NV index
40                     out->moreData = NvCapGetIndex((TPM_HANDLE)in->property,
41                                                   in->propertyCount,
42                                                   &data->handles);
43                     break;
44                 case TPM_HT_LOADED_SESSION:
45                     // Get list of handles of loaded sessions
46                     out->moreData = SessionCapGetLoaded((TPM_HANDLE)in->property,
47                                                         in->propertyCount,
48                                                         &data->handles);
49                     break;
50 #ifdef TPM_HT_SAVED_SESSION
51                 case TPM_HT_SAVED_SESSION:
52 #else
```

```
53                  case TPM_HT_ACTIVE_SESSION:
54  #endif
55              // Get list of handles of
56                      out->moreData = SessionCapGetSaved((TPM_HANDLE)in->property,
57                                                         in->propertyCount,
58                                                         &data->handles);
59                      break;
60                  case TPM_HT_PCR:
61                      // Get list of handles of PCR
62                      out->moreData = PCRCapGetHandles((TPM_HANDLE)in->property,
63                                                       in->propertyCount,
64                                                       &data->handles);
65                      break;
66                  case TPM_HT_PERMANENT:
67                      // Get list of permanent handles
68                      out->moreData = PermanentCapGetHandles((TPM_HANDLE)in->property,
69                                                             in->propertyCount,
70                                                             &data->handles);
71                      break;
72                  default:
73                      // Unsupported input handle type
74                      return TPM_RCS_HANDLE + RC_GetCapability_property;
75                      break;
76              }
77              break;
78          case TPM_CAP_COMMANDS:
79              out->moreData = CommandCapGetCCList((TPM_CC)in->property,
80                                                  in->propertyCount,
81                                                  &data->command);
82              break;
83          case TPM_CAP_PP_COMMANDS:
84              out->moreData = PhysicalPresenceCapGetCCList((TPM_CC)in->property,
85                                                           in->propertyCount,
86                                                           &data->ppCommands);
87              break;
88          case TPM_CAP_AUDIT_COMMANDS:
89              out->moreData = CommandAuditCapGetCCList((TPM_CC)in->property,
90                                                       in->propertyCount,
91                                                       &data->auditCommands);
92              break;
93          case TPM_CAP_PCRS:
94              // Input property must be 0
95              if(in->property != 0)
96                  return TPM_RCS_VALUE + RC_GetCapability_property;
97              out->moreData = PCRCapGetAllocation(in->propertyCount,
98                                                  &data->assignedPCR);
99              break;
100         case TPM_CAP_PCR_PROPERTIES:
101             out->moreData = PCRCapGetProperties((TPM_PT_PCR)in->property,
102                                                 in->propertyCount,
103                                                 &data->pcrProperties);
104             break;
105         case TPM_CAP_TPM_PROPERTIES:
106             out->moreData = TPMCapGetProperties((TPM_PT)in->property,
107                                                 in->propertyCount,
108                                                 &data->tpmProperties);
109             break;
110 #if ALG_ECC
111         case TPM_CAP_ECC_CURVES:
112             out->moreData = CryptCapGetECCCurve((TPM_ECC_CURVE)in->property,
113                                                 in->propertyCount,
114                                                 &data->eccCurves);
115             break;
116 #endif // ALG_ECC
117         case TPM_CAP_AUTH_POLICIES:
118             if(HandleGetType((TPM_HANDLE)in->property) != TPM_HT_PERMANENT)
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

```
119                 return TPM_RCS_VALUE + RC_GetCapability_property;
120             out->moreData = PermanentHandleGetPolicy((TPM_HANDLE)in->property,
121                                                       in->propertyCount,
122                                                       &data->authPolicies);
123             break;
124         case TPM_CAP_ACT:
125             if(((TPM_RH)in->property < TPM_RH_ACT_0)
126                 || ((TPM_RH)in->property > TPM_RH_ACT_F))
127                  return TPM_RCS_VALUE + RC_GetCapability_property;
128             out->moreData = ActGetCapabilityData((TPM_HANDLE)in->property,
129                                                   in->propertyCount,
130                                                   &data->actData);
131             break;
132         case TPM_CAP_VENDOR_PROPERTY:
133             // vendor property is not implemented
134         default:
135             // Unsupported TPM_CAP value
136             return TPM_RCS_VALUE + RC_GetCapability_capability;
137             break;
138     }
139
140     return TPM_RC_SUCCESS;
141 }
142 #endif // CC_GetCapability
```

30.3 TPM2_TestParms

30.3.1 General Description

This command is used to check to see if specific combinations of algorithm parameters are supported.

The TPM will unmarshal the provided TPMT_PUBLIC_PARMS. If the parameters unmarshal correctly, then the TPM will return TPM_RC_SUCCESS, indicating that the parameters are valid for the TPM. The TPM will return the appropriate unmarshaling error if a parameter is not valid.

 Copyright © TCG 2006-2020

30.3.2 Command and Response

Table 210 — TPM2_TestParms Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_TestParms
TPMT_PUBLIC_PARMS	parameters	algorithm parameters to be validated

Table 211 — TPM2_TestParms Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	TPM_RC

 Copyright © TCG 2006-2020

30.3.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "TestParms_fp.h"
3  #if CC_TestParms  // Conditional expansion of this file
4  TPM_RC
5  TPM2_TestParms(
6      TestParms_In    *in             // IN: input parameter list
7      )
8  {
9      // Input parameter is not reference in command action
10     NOT_REFERENCED(in);
11
12     // The parameters are tested at unmarshal process.  We do nothing in command
13     // action
14     return TPM_RC_SUCCESS;
15 }
16 #endif // CC_TestParms
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31 Non-volatile Storage

31.1 Introduction

The NV commands are used to create, update, read, and delete allocations of space in NV memory. Before an Index may be used, it must be defined (TPM2_NV_DefineSpace()).

An Index may be modified if the proper write authorization is provided or read if the proper read authorization is provided. Different controls are available for reading and writing.

An Index may have an Index-specific *authValue* and *authPolicy*. The *authValue* may be used to authorize reading if TPMA_NV_AUTHREAD is SET and writing if TPMA_NV_AUTHWRITE is SET. The *authPolicy* may be used to authorize reading if TPMA_NV_POLICYREAD is SET and writing if TPMA_NV_POLICYWRITE is SET.

For commands that have both *authHandle* and *nvIndex* parameters, *authHandle* can be an NV Index, Platform Authorization, or Owner Authorization. If *authHandle* is an NV Index, it must be the same as *nvIndex* (TPM_RC_NV_AUTHORIZATION).

TPMA_NV_PPREAD and TPMA_NV_PPWRITE indicate if reading or writing of the NV Index may be authorized by *platformAuth* or *platformPolicy*.

TPMA_NV_OWNERREAD and TPMA_NV_OWNERWRITE indicate if reading or writing of the NV Index may be authorized by *ownerAuth* or *ownerPolicy*.

If an operation on an NV index requires authorization, and the authHandle parameter is the handle of an NV Index, then the nvIndex parameter must have the same value or the TPM will return TPM_RC_NV_AUTHORIZATION.

NOTE 1 This check ensures that the authorization that was provided is associated with the NV Index being authorized.

For creating an Index, Owner Authorization may not be used if *shEnable* is CLEAR and Platform Authorization may not be used if *phEnableNV* is CLEAR.

If an Index was defined using Platform Authorization, then that Index is not accessible when *phEnableNV* is CLEAR. If an Index was defined using Owner Authorization, then that Index is not accessible when *shEnable* is CLEAR.

For read access control, any combination of TPMA_NV_PPREAD, TPMA_NV_OWNERREAD, TPMA_NV_AUTHREAD, or TPMA_NV_POLICYREAD is allowed as long as at least one is SET.

For write access control, any combination of TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, TPMA_NV_AUTHWRITE, or TPMA_NV_POLICYWRITE is allowed as long as at least one is SET.

If an Index has been defined and not written, then any operation on the NV Index that requires read authorization will fail (TPM_RC_NV_INITIALIZED). This check may be made before or after other authorization checks but shall be performed before checking the NV Index *authValue*. An authorization failure due to the NV Index not having been written shall not be logged by the dictionary attack logic.

If TPMA_NV_CLEAR_STCLEAR is SET, then the TPMA_NV_WRITTEN will be CLEAR on each TPM2_Startup(TPM_SU_CLEAR). TPMA_NV_CLEAR_STCLEAR shall not be SET if the *nvIndexType* is TPM_NT_COUNTER.

The code in the “Detailed Actions” clause of each command is written to interface with an implementation-dependent library that allows access to NV memory. The actions assume no specific layout of the structure of the NV data.

Only one NV Index may be directly referenced in a command.

NOTE 2 This means that, if *authHandle* references an NV Index, then *nvIndex* will have the same value. However, this does not limit the number of changes that may occur as side effects. For example, any number of NV Indexes might be relocated as a result of deleting or adding a NV Index.

31.2 NV Counters

When an Index has the TPM_NT_COUNTER attribute, it behaves as a monotonic counter and may only be updated using TPM2_NV_Increment().

When an NV counter is created, the TPM shall initialize the 8-octet counter value with a number that is greater than any count value for any NV counter on the TPM since the time of TPM manufacture.

An NV counter may be defined with the TPMA_NV_ORDERLY attribute to indicate that the NV Index is expected to be modified at a high frequency and that the data is only required to persist when the TPM goes through an orderly shutdown process. The TPM may update the counter value in RAM and occasionally update the non-volatile version of the counter. An orderly shutdown is one occasion to update the non-volatile count. If the difference between the volatile and non-volatile version of the counter becomes as large as MAX_ORDERLY_COUNT, this shall be another occasion for updating the non-volatile count.

Before an NV counter can be used, the TPM shall validate that the count is not less than a previously reported value. If the TPMA_NV_ORDERLY attribute is not SET, or if the TPM experienced an orderly shutdown, then the count is assumed to be correct. If the TPMA_NV_ORDERLY attribute is SET, and the TPM shutdown was not orderly, then the TPM shall OR MAX_ORDERLY_COUNT to the contents of the non-volatile counter and set that as the current count.

NOTE 1 Because the TPM would have updated the NV Index if the difference between the count values was equal to MAX_ORDERLY_COUNT + 1, the highest value that could have been in the NV Index is MAX_ORDERLY_COUNT so it is safe to restore that value.

NOTE 2 The TPM may implement the RAM portion of the counter such that the effective value of the NV counter is the sum of both the volatile and non-volatile parts. If so, then the TPM may initialize the RAM version of the counter to MAX_ORDERLY_COUNT and no update of NV is necessary.

NOTE 3 When a new NV counter is created, the TPM may search all the counters to determine which has the highest value. In this search, the TPM would use the sum of the non-volatile and RAM portions of the counter. The RAM portion of the counter shall be properly initialized to reflect shutdown process (orderly or not) of the TPM.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.3 TPM2_NV_DefineSpace

31.3.1 General Description

This command defines the attributes of an NV Index and causes the TPM to reserve space to hold the data associated with the NV Index. If a definition already exists at the NV Index, the TPM will return TPM_RC_NV_DEFINED.

The TPM will return TPM_RC_ATTRIBUTES if *nvIndexType* has a reserved value in *publicInfo.*

NOTE 1 It is not required that any of these three attributes be set.

The TPM shall return TPM_RC_ATTRIBUTES if TPMA_NV_WRITTEN, TPMA_NV_READLOCKED, or TPMA_NV_WRITELOCKED is SET.

If *nvIndexType* is TPM_NT_COUNTER, TPM_NT_BITS, TPM_NT_PIN_FAIL, or TPM_NT_PIN_PASS, then *publicInfo→dataSize* shall be set to eight (8) or the TPM shall return TPM_RC_SIZE.

If *nvIndexType* is TPM_NT_EXTEND, then *publicInfo→dataSize* shall match the digest size of the *publicInfo.nameAlg* or the TPM shall return TPM_RC_SIZE.

NOTE 2 TPM_RC_ATTRIBUTES could be returned by a TPM that is based on the reference code of older versions of the specification but the correct response for this error is TPM_RC_SIZE.

If the NV Index is an ordinary Index and *publicInfo→dataSize* is larger than supported by the TPM implementation then the TPM shall return TPM_RC_SIZE.

NOTE 3 The limit for the data size may vary according to the type of the index. For example, if the index has TPMA_NV_ORDERLY SET, then the maximum size of an ordinary NV Index may be less than the size of an ordinary NV Index that has TPMA_NV_ORDERLY CLEAR.

At least one of TPMA_NV_PPREAD, TPMA_NV_OWNERREAD, TPMA_NV_AUTHREAD, or TPMA_NV_POLICYREAD shall be SET or the TPM shall return TPM_RC_ATTRIBUTES.

At least one of TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, TPMA_NV_AUTHWRITE, or TPMA_NV_POLICYWRITE shall be SET or the TPM shall return TPM_RC_ATTRIBUTES.

If TPMA_NV_CLEAR_STCLEAR is SET, then *nvIndexType* shall not be TPM_NT_COUNTER or the TPM shall return TPM_RC_ATTRIBUTES.

If *platformAuth/platformPolicy* is used for authorization, then TPMA_NV_PLATFORMCREATE shall be SET in *publicInfo*. If *ownerAuth/ownerPolicy* is used for authorization, TPMA_NV_PLATFORMCREATE shall be CLEAR in *publicInfo.* If TPMA_NV_PLATFORMCREATE is not set correctly for the authorization, the TPM shall return TPM_RC_ATTRIBUTES.

If TPMA_NV_POLICY_DELETE is SET, then the authorization shall be with Platform Authorization or the TPM shall return TPM_RC_ATTRIBUTES.

If *nvIndexType* is TPM_NT_PIN_FAIL, then TPMA_NV_NO_DA shall be SET. Otherwise, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE 4 The intent of a PIN Fail index is that its DA protection is on a per-index basis, not based on the global DA protection. This avoids conflict over which type of dictionary attack protection is in use.

If *nvIndexType* is TPM_NT_PIN_FAIL or TPM_NT_PIN_PASS, then at least one of TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, or TPMA_NV_POLICYWRITE shall be SET or the TPM shall return TPM_RC_ATTRIBUTES. TPMA_NV_AUTHWRITE shall be CLEAR. Otherwise, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE 5 If TPMA_NV_AUTHWRITE was SET for a PIN Pass index, a user knowing the authorization value could decrease pinCount or increase pinLimit, defeating the purpose of a PIN Pass index. The requirement is also enforced for a PIN Fail index for consistency.

 Copyright © TCG 2006-2020

If the implementation does not support TPM2_NV_Increment(), the TPM shall return TPM_RC_ATTRIBUTES if *nvIndexType* is TPM_NT_COUNTER.

If the implementation does not support TPM2_NV_SetBits(), the TPM shall return TPM_RC_ATTRIBUTES if *nvIndexType* is TPM_NT_BITS.

If the implementation does not support TPM2_NV_Extend(), the TPM shall return TPM_RC_ATTRIBUTES if *nvIndexType* is TPM_NT_EXTEND.

If the implementation does not support TPM2_NV_UndefineSpaceSpecial(), the TPM shall return TPM_RC_ATTRIBUTES if TPMA_NV_POLICY_DELETE is SET.

After the successful completion of this command, the NV Index exists but TPMA_NV_WRITTEN will be CLEAR. Any access of the NV data will return TPM_RC_NV_UNINITIALIZED.

In some implementations, an NV Index with the TPM_NT_COUNTER attribute may require special TPM resources that provide higher endurance than regular NV. For those implementations, if this command fails because of lack of resources, the TPM will return TPM_RC_NV_SPACE.

The value of *auth* is saved in the created structure. The size of *auth* is limited to be no larger than the size of the digest produced by the NV Index's *nameAlg* (TPM_RC_SIZE).

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.3.2 Command and Response

Table 212 — TPM2_NV_DefineSpace Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_DefineSpace {NV}
TPMI_RH_PROVISION	@authHandle	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPM2B_AUTH	auth	the authorization value
TPM2B_NV_PUBLIC	publicInfo	the public parameters of the NV area

Table 213 — TPM2_NV_DefineSpace Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

31.3.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "NV_DefineSpace_fp.h"
3  #if CC_NV_DefineSpace  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_HIERARCHY	for authorizations using TPM_RH_PLATFORM *phEnable_NV* is clear preventing access to NV data in the platform hierarchy.
TPM_RC_ATTRIBUTES	attributes of the index are not consistent
TPM_RC_NV_DEFINED	index already exists
TPM_RC_NV_SPACE	insufficient space for the index
TPM_RC_SIZE	'auth->size' or '*publicInfo->authPolicy*.size' is larger than the digest size of '*publicInfo->nameAlg*'; or '*publicInfo->dataSize*' is not consistent with '*publicInfo*->attributes' (this includes the case when the index is larger than a MAX_NV_BUFFER_SIZE but the TPMA_NV_WRITEALL attribute is SET)

```
4  TPM_RC
5  TPM2_NV_DefineSpace(
6      NV_DefineSpace_In   *in             // IN: input parameter list
7      )
8  {
9      TPMA_NV         attributes = in->publicInfo.nvPublic.attributes;
10     UINT16          nameSize;
11
12     nameSize = CryptHashGetDigestSize(in->publicInfo.nvPublic.nameAlg);
13
14 // Input Validation
15
16     // Checks not specific to type
17
18     // If the UndefineSpaceSpecial command is not implemented, then can't have
19     // an index that can only be deleted with policy
20 #if CC_NV_UndefineSpaceSpecial == NO
21     if(IS_ATTRIBUTE(attributes, TPMA_NV, POLICY_DELETE))
22         return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
23 #endif
24
25     // check that the authPolicy consistent with hash algorithm
26
27     if(in->publicInfo.nvPublic.authPolicy.t.size != 0
28        && in->publicInfo.nvPublic.authPolicy.t.size != nameSize)
29         return TPM_RCS_SIZE + RC_NV_DefineSpace_publicInfo;
30
31    // make sure that the authValue is not too large
32     if(MemoryRemoveTrailingZeros(&in->auth)
33        > CryptHashGetDigestSize(in->publicInfo.nvPublic.nameAlg))
34         return TPM_RCS_SIZE + RC_NV_DefineSpace_auth;
35
36     // If an index is being created by the owner and shEnable is
37     // clear, then we would not reach this point because ownerAuth
38     // can't be given when shEnable is CLEAR. However, if phEnable
39     // is SET but phEnableNV is CLEAR, we have to check here
40     if(in->authHandle == TPM_RH_PLATFORM && gc.phEnableNV == CLEAR)
41         return TPM_RCS_HIERARCHY + RC_NV_DefineSpace_authHandle;
42
43     // Attribute checks
44     // Eliminate the unsupported types
```

 Copyright © TCG 2006-2020

```
45      switch(GET_TPM_NT(attributes))
46      {
47  #if CC_NV_Increment == YES
48          case TPM_NT_COUNTER:
49  #endif
50  #if CC_NV_SetBits == YES
51          case TPM_NT_BITS:
52  #endif
53  #if CC_NV_Extend == YES
54          case TPM_NT_EXTEND:
55  #endif
56  #if CC_PolicySecret == YES && defined TPM_NT_PIN_PASS
57          case TPM_NT_PIN_PASS:
58          case TPM_NT_PIN_FAIL:
59  #endif
60          case TPM_NT_ORDINARY:
61              break;
62          default:
63              return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
64              break;
65      }
66      // Check that the sizes are OK based on the type
67      switch(GET_TPM_NT(attributes))
68      {
69          case TPM_NT_ORDINARY:
70              // Can't exceed the allowed size for the implementation
71              if(in->publicInfo.nvPublic.dataSize > MAX_NV_INDEX_SIZE)
72                  return TPM_RCS_SIZE + RC_NV_DefineSpace_publicInfo;
73              break;
74          case TPM_NT_EXTEND:
75              if(in->publicInfo.nvPublic.dataSize != nameSize)
76                  return TPM_RCS_SIZE + RC_NV_DefineSpace_publicInfo;
77              break;
78          default:
79              // Everything else needs a size of 8
80              if(in->publicInfo.nvPublic.dataSize != 8)
81                  return TPM_RCS_SIZE + RC_NV_DefineSpace_publicInfo;
82              break;
83      }
84      // Handle other specifics
85      switch(GET_TPM_NT(attributes))
86      {
87          case TPM_NT_COUNTER:
88              // Counter can't have TPMA_NV_CLEAR_STCLEAR SET (don't clear counters)
89              if(IS_ATTRIBUTE(attributes, TPMA_NV, CLEAR_STCLEAR))
90                  return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
91              break;
92  #ifdef TPM_NT_PIN_FAIL
93          case TPM_NT_PIN_FAIL:
94              // NV_NO_DA must be SET and AUTHWRITE must be CLEAR
95              // NOTE: As with a PIN_PASS index, the authValue of the index is not
96              // available until the index is written. If AUTHWRITE is the only way to
97              // write then index, it could never be written. Rather than go through
98              // all of the other possible ways to write the Index, it is simply
99              // prohibited to write the index with the authValue. Other checks
100             // below will insure that there seems to be a way to write the index
101             // (i.e., with platform authorization , owner authorization,
102             // or with policyAuth.)
103             // It is not allowed to create a PIN Index that can't be modified.
104             if(!IS_ATTRIBUTE(attributes, TPMA_NV, NO_DA))
105                 return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
106 #endif
107 #ifdef TPM_NT_PIN_PASS
108         case TPM_NT_PIN_PASS:
109             // AUTHWRITE must be CLEAR (see note above to TPM_NT_PIN_FAIL)
110             if(IS_ATTRIBUTE(attributes, TPMA_NV, AUTHWRITE)
```

```
111                 || IS_ATTRIBUTE(attributes, TPMA_NV, GLOBALLOCK)
112                 || IS_ATTRIBUTE(attributes, TPMA_NV, WRITEDEFINE))
113                  return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
114 #endif  // this comes before break because PIN_FAIL falls through
115             break;
116         default:
117             break;
118     }
119
120     // Locks may not be SET and written cannot be SET
121     if(IS_ATTRIBUTE(attributes, TPMA_NV, WRITTEN)
122        || IS_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED)
123        || IS_ATTRIBUTE(attributes, TPMA_NV, READLOCKED))
124         return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
125
126     // There must be a way to read the index.
127     if(!IS_ATTRIBUTE(attributes, TPMA_NV, OWNERREAD)
128        && !IS_ATTRIBUTE(attributes, TPMA_NV, PPREAD)
129        && !IS_ATTRIBUTE(attributes, TPMA_NV, AUTHREAD)
130        && !IS_ATTRIBUTE(attributes, TPMA_NV, POLICYREAD))
131         return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
132
133     // There must be a way to write the index
134     if(!IS_ATTRIBUTE(attributes, TPMA_NV, OWNERWRITE)
135        && !IS_ATTRIBUTE(attributes, TPMA_NV, PPWRITE)
136        && !IS_ATTRIBUTE(attributes, TPMA_NV, AUTHWRITE)
137        && !IS_ATTRIBUTE(attributes, TPMA_NV, POLICYWRITE))
138         return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
139
140     // An index with TPMA_NV_CLEAR_STCLEAR can't have TPMA_NV_WRITEDEFINE SET
141     if(IS_ATTRIBUTE(attributes, TPMA_NV, CLEAR_STCLEAR)
142        &&  IS_ATTRIBUTE(attributes, TPMA_NV, WRITEDEFINE))
143         return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
144
145     // Make sure that the creator of the index can delete the index
146     if((IS_ATTRIBUTE(attributes, TPMA_NV, PLATFORMCREATE)
147         && in->authHandle == TPM_RH_OWNER)
148        || (!IS_ATTRIBUTE(attributes, TPMA_NV, PLATFORMCREATE)
149            && in->authHandle == TPM_RH_PLATFORM))
150         return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_authHandle;
151
152     // If TPMA_NV_POLICY_DELETE is SET, then the index must be defined by
153     // the platform
154     if(IS_ATTRIBUTE(attributes, TPMA_NV, POLICY_DELETE)
155        &&  TPM_RH_PLATFORM != in->authHandle)
156         return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
157
158     // Make sure that the TPMA_NV_WRITEALL is not set if the index size is larger
159     // than the allowed NV buffer size.
160     if(in->publicInfo.nvPublic.dataSize > MAX_NV_BUFFER_SIZE
161        &&  IS_ATTRIBUTE(attributes, TPMA_NV, WRITEALL))
162         return TPM_RCS_SIZE + RC_NV_DefineSpace_publicInfo;
163
164     // And finally, see if the index is already defined.
165     if(NvIndexIsDefined(in->publicInfo.nvPublic.nvIndex))
166         return TPM_RC_NV_DEFINED;
167
168 // Internal Data Update
169     // define the space.  A TPM_RC_NV_SPACE error may be returned at this point
170     return NvDefineIndex(&in->publicInfo.nvPublic, &in->auth);
171 }
172 #endif // CC_NV_DefineSpace
```

 Copyright © TCG 2006-2020

31.4 TPM2_NV_UndefineSpace

31.4.1 General Description

This command removes an Index from the TPM.

If *nvIndex* is not defined, the TPM shall return TPM_RC_HANDLE.

If *nvIndex* references an Index that has its TPMA_NV_PLATFORMCREATE attribute SET, the TPM shall return TPM_RC_NV_AUTHORIZATION unless Platform Authorization is provided.

If *nvIndex* references an Index that has its TPMA_NV_POLICY_DELETE attribute SET, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE An Index with TPMA_NV_PLATFORMCREATE CLEAR may be deleted with Platform Authorization as long as shEnable is SET. If shEnable is CLEAR, indexes created using Owner Authorization are not accessible even for deletion by the platform.

 Copyright © TCG 2006-2020

31.4.2 Command and Response

Table 214 — TPM2_NV_UndefineSpace Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_UndefineSpace {NV}
TPMI_RH_PROVISION	@authHandle	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index to remove from NV space Auth Index: None

Table 215 — TPM2_NV_UndefineSpace Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.4.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "NV_UndefineSpace_fp.h"
3  #if CC_NV_UndefineSpace  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	TPMA_NV_POLICY_DELETE is SET in the Index referenced by *nvIndex* so this command may not be used to delete this Index (see TPM2_NV_UndefineSpaceSpecial())
TPM_RC_NV_AUTHORIZATION	attempt to use *ownerAuth* to delete an index created by the platform

```
4  TPM_RC
5  TPM2_NV_UndefineSpace(
6      NV_UndefineSpace_In     *in              // IN: input parameter list
7      )
8  {
9      NV_REF           locator;
10     NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
11
12 // Input Validation
13     // This command can't be used to delete an index with TPMA_NV_POLICY_DELETE SET
14     if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, POLICY_DELETE))
15         return TPM_RCS_ATTRIBUTES + RC_NV_UndefineSpace_nvIndex;
16
17     // The owner may only delete an index that was defined with ownerAuth. The
18     // platform may delete an index that was created with either authorization.
19     if(in->authHandle == TPM_RH_OWNER
20        && IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, PLATFORMCREATE))
21         return TPM_RC_NV_AUTHORIZATION;
22
23 // Internal Data Update
24
25     // Call implementation dependent internal routine to delete NV index
26     return NvDeleteIndex(nvIndex, locator);
27 }
28 #endif // CC_NV_UndefineSpace
```

 Copyright © TCG 2006-2020

31.5 TPM2_NV_UndefineSpaceSpecial

31.5.1 General Description

This command allows removal of a platform-created NV Index that has TPMA_NV_POLICY_DELETE SET.

This command requires that the policy of the NV Index be satisfied before the NV Index may be deleted. Because administrative role is required, the policy must contain a command that sets the policy command code to TPM_CC_NV_UndefineSpaceSpecial. This indicates that the policy that is being used is a policy that is for this command, and not a policy that would approve another use. That is, authority to use an entity does not grant authority to undefine the entity.

Since the index is deleted, the Empty Buffer is used as the authValue when generating the response HMAC.

If *nvIndex* is not defined, the TPM shall return TPM_RC_HANDLE.

If *nvIndex* references an Index that has its TPMA_NV_PLATFORMCREATE or TPMA_NV_POLICY_DELETE attribute CLEAR, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE An Index with TPMA_NV_PLATFORMCREATE CLEAR may be deleted with TPM2_UndefineSpace()as long as shEnable is SET. If shEnable is CLEAR, indexes created using Owner Authorization are not accessible even for deletion by the platform.

 Copyright © TCG 2006-2020

31.5.2 Command and Response

Table 216 — TPM2_NV_UndefineSpaceSpecial Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_UndefineSpaceSpecial {NV}
TPMI_RH_NV_INDEX	@nvIndex	Index to be deleted Auth Index: 1 Auth Role: ADMIN
TPMI_RH_PLATFORM	@platform	TPM_RH_PLATFORM + {PP} Auth Index: 2 Auth Role: USER

Table 217 — TPM2_NV_UndefineSpaceSpecial Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

31.5.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "NV_UndefineSpaceSpecial_fp.h"
3  #include "SessionProcess_fp.h"
4  #if CC_NV_UndefineSpaceSpecial  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	TPMA_NV_POLICY_DELETE is not SET in the Index referenced by *nvIndex*

```
5  TPM_RC
6  TPM2_NV_UndefineSpaceSpecial(
7      NV_UndefineSpaceSpecial_In  *in              // IN: input parameter list
8      )
9  {
10     TPM_RC           result;
11     NV_REF           locator;
12     NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
13 // Input Validation
14     // This operation only applies when the TPMA_NV_POLICY_DELETE attribute is SET
15     if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, POLICY_DELETE))
16         return TPM_RCS_ATTRIBUTES + RC_NV_UndefineSpaceSpecial_nvIndex;
17 // Internal Data Update
18     // Call implementation dependent internal routine to delete NV index
19     result = NvDeleteIndex(nvIndex, locator);
20
21     // If we just removed the index providing the authorization, make sure that the
22     // authorization session computation is modified so that it doesn't try to
23     // access the authValue of the just deleted index
24     if(result == TPM_RC_SUCCESS)
25         SessionRemoveAssociationToHandle(in->nvIndex);
26     return result;
27 }
28 #endif // CC_NV_UndefineSpaceSpecial
```

 Copyright © TCG 2006-2020

31.6 TPM2_NV_ReadPublic

31.6.1 General Description

This command is used to read the public area and Name of an NV Index. The public area of an Index is not privacy-sensitive and no authorization is required to read this data.

 Copyright © TCG 2006-2020

31.6.2 Command and Response

Table 218 — TPM2_NV_ReadPublic Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_ReadPublic
TPMI_RH_NV_INDEX	nvIndex	the NV Index Auth Index: None

Table 219 — TPM2_NV_ReadPublic Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_NV_PUBLIC	nvPublic	the public area of the NV Index
TPM2B_NAME	nvName	the Name of the *nvIndex*

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.6.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "NV_ReadPublic_fp.h"
3  #if CC_NV_ReadPublic  // Conditional expansion of this file
4  TPM_RC
5  TPM2_NV_ReadPublic(
6      NV_ReadPublic_In    *in,             // IN: input parameter list
7      NV_ReadPublic_Out   *out             // OUT: output parameter list
8      )
9  {
10     NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, NULL);
11
12 // Command Output
13
14     // Copy index public data to output
15     out->nvPublic.nvPublic = nvIndex->publicArea;
16
17     // Compute NV name
18     NvGetIndexName(nvIndex, &out->nvName);
19
20     return TPM_RC_SUCCESS;
21 }
22 #endif // CC_NV_ReadPublic
```

 Copyright © TCG 2006-2020

31.7 TPM2_NV_Write

31.7.1 General Description

This command writes a value to an area in NV memory that was previously defined by TPM2_NV_DefineSpace().

Proper authorizations are required for this command as determined by TPMA_NV_PPWRITE; TPMA_NV_OWNERWRITE; TPMA_NV_AUTHWRITE; and, if TPMA_NV_POLICY_WRITE is SET, the *authPolicy* of the NV Index.

If the TPMA_NV_WRITELOCKED attribute of the NV Index is SET, then the TPM shall return TPM_RC_NV_LOCKED.

NOTE 1 If authorization sessions are present, they are checked before checks to see if writes to the NV Index are locked.

If *nvIndexType* is TPM_NT_COUNTER, TPM_NT_BITS or TPM_NT_EXTEND, then the TPM shall return TPM_RC_ATTRIBUTES.

If *offset* and the *size* field of *data* add to a value that is greater than the *dataSize* field of the NV Index referenced by *nvIndex*, the TPM shall return an error (TPM_RC_NV_RANGE). The implementation may return an error (TPM_RC_VALUE) if it performs an additional check and determines that *offset* is greater than the *dataSize* field of the NV Index.

If the TPMA_NV_WRITEALL attribute of the NV Index is SET, then the TPM shall return TPM_RC_NV_RANGE if the size of the *data* parameter of the command is not the same as the *data* field of the NV Index.

If all checks succeed, the TPM will merge the *data.size* octets of *data.buffer* value into the *nvIndex→data* starting at *nvIndex→data[offset].* If the NV memory is implemented with a technology that has endurance limitations, the TPM shall check that the merged data is different from the current contents of the NV Index and only perform a write to NV memory if they differ.

After successful completion of this command, TPMA_NV_WRITTEN for the NV Index will be SET.

NOTE 2 Once SET, TPMA_NV_WRITTEN remains SET until the NV Index is undefined or the NV Index is cleared.

 Copyright © TCG 2006-2020

31.7.2 Command and Response

Table 220 — TPM2_NV_Write Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_Write {NV}
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index of the area to write Auth Index: None
TPM2B_MAX_NV_BUFFER	data	the data to write
UINT16	offset	the octet offset into the NV Area

Table 221 — TPM2_NV_Write Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

31.7.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "NV_Write_fp.h"
3  #if CC_NV_Write  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	Index referenced by *nvIndex* has either TPMA_NV_BITS, TPMA_NV_COUNTER, or TPMA_NV_EVENT attribute SET
TPM_RC_NV_AUTHORIZATION	the authorization was valid but the authorizing entity (*authHandle*) is not allowed to write to the Index referenced by *nvIndex*
TPM_RC_NV_LOCKED	Index referenced by *nvIndex* is write locked
TPM_RC_NV_RANGE	if TPMA_NV_WRITEALL is SET then the write is not the size of the Index referenced by *nvIndex*; otherwise, the write extends beyond the limits of the Index

```
4  TPM_RC
5  TPM2_NV_Write(
6      NV_Write_In     *in             // IN: input parameter list
7      )
8  {
9      NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, NULL);
10     TPMA_NV          attributes = nvIndex->publicArea.attributes;
11     TPM_RC           result;
12
13 // Input Validation
14
15     // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
16     // or TPM_RC_NV_LOCKED
17     result = NvWriteAccessChecks(in->authHandle,
18                                  in->nvIndex,
19                                  attributes);
20     if(result != TPM_RC_SUCCESS)
21         return result;
22
23     // Bits index, extend index or counter index may not be updated by
24     // TPM2_NV_Write
25     if(IsNvCounterIndex(attributes)
26        || IsNvBitsIndex(attributes)
27        || IsNvExtendIndex(attributes))
28         return TPM_RC_ATTRIBUTES;
29
30     // Make sure that the offset is not too large
31     if(in->offset > nvIndex->publicArea.dataSize)
32         return TPM_RCS_VALUE + RC_NV_Write_offset;
33
34     // Make sure that the selection is within the range of the Index
35     if(in->data.t.size > (nvIndex->publicArea.dataSize - in->offset))
36         return TPM_RC_NV_RANGE;
37
38     // If this index requires a full sized write, make sure that input range is
39     // full sized.
40     // Note: if the requested size is the same as the Index data size, then offset
41     // will have to be zero. Otherwise, the range check above would have failed.
42     if(IS_ATTRIBUTE(attributes, TPMA_NV, WRITEALL)
43        && in->data.t.size < nvIndex->publicArea.dataSize)
44         return TPM_RC_NV_RANGE;
45
46 // Internal Data Update
47
```

 Copyright © TCG 2006-2020

```
48      // Perform the write.  This called routine will SET the TPMA_NV_WRITTEN
49      // attribute if it has not already been SET. If NV isn't available, an error
50      // will be returned.
51      return NvWriteIndexData(nvIndex, in->offset, in->data.t.size,
52                              in->data.t.buffer);
53  }
54  #endif // CC_NV_Write
```

31.8 TPM2_NV_Increment

31.8.1 General Description

This command is used to increment the value in an NV Index that has the TPM_NT_COUNTER attribute. The data value of the NV Index is incremented by one.

NOTE 1 The NV Index counter is an unsigned value.

If *nvIndexType* is not TPM_NT_COUNTER in the indicated NV Index, the TPM shall return TPM_RC_ATTRIBUTES.

If TPMA_NV_WRITELOCKED is SET, the TPM shall return TPM_RC_NV_LOCKED.

If TPMA_NV_WRITTEN is CLEAR, it will be SET.

If TPMA_NV_ORDERLY is SET, and the difference between the volatile and non-volatile versions of this field is greater than MAX_ORDERLY_COUNT, then the non-volatile version of the counter is updated.

NOTE 2 If a TPM implements TPMA_NV_ORDERLY and an Index is defined with TPMA_NV_ORDERLY and TPM_NT_COUNTER both SET, then in the Event of a non-orderly shutdown, the non-volatile value for the counter Index will be advanced by MAX_ORDERLY_COUNT at the next TPM2_Startup().

NOTE 3 An allowed implementation would keep a counter value in NV and a resettable counter in RAM. The reported value of the NV Index would be the sum of the two values. When the RAM count increments past the maximum allowed value (MAX_ORDERLY_COUNT), the non-volatile version of the count is updated with the sum of the values and the RAM count is reset to zero.

 Copyright © TCG 2006-2020

31.8.2 Command and Response

Table 222 — TPM2_NV_Increment Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_Increment {NV}
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index to increment Auth Index: None

Table 223 — TPM2_NV_Increment Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

31.8.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "NV_Increment_fp.h"
3  #if CC_NV_Increment  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	NV index is not a counter
TPM_RC_NV_AUTHORIZATION	authorization failure
TPM_RC_NV_LOCKED	Index is write locked

```
4  TPM_RC
5  TPM2_NV_Increment(
6      NV_Increment_In     *in              // IN: input parameter list
7      )
8  {
9      TPM_RC          result;
10     NV_REF          locator;
11     NV_INDEX       *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
12     UINT64          countValue;
13
14 // Input Validation
15
16     // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
17     // or TPM_RC_NV_LOCKED
18     result = NvWriteAccessChecks(in->authHandle,
19                                  in->nvIndex,
20                                  nvIndex->publicArea.attributes);
21     if(result != TPM_RC_SUCCESS)
22         return result;
23
24     // Make sure that this is a counter
25     if(!IsNvCounterIndex(nvIndex->publicArea.attributes))
26         return TPM_RCS_ATTRIBUTES + RC_NV_Increment_nvIndex;
27
28 // Internal Data Update
29
30     // If counter index is not been written, initialize it
31     if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
32         countValue = NvReadMaxCount();
33     else
34         // Read NV data in native format for TPM CPU.
35         countValue = NvGetUINT64Data(nvIndex, locator);
36
37     // Do the increment
38     countValue++;
39
40     // Write NV data back. A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may
41     // be returned at this point. If necessary, this function will set the
42     // TPMA_NV_WRITTEN attribute
43     result = NvWriteUINT64Data(nvIndex, countValue);
44     if(result == TPM_RC_SUCCESS)
45     {
46         // If a counter just rolled over, then force the NV update.
47         // Note, if this is an orderly counter, then the write-back needs to be
48         // forced, for other counters, the write-back will happen anyway
49         if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY)
50            && (countValue & MAX_ORDERLY_COUNT) == 0 )
51         {
52             // Need to force an NV update of orderly data
```

 Copyright © TCG 2006-2020

```
53              SET_NV_UPDATE(UT_ORDERLY);
54          }
55      }
56      return result;
57  }
58  #endif // CC_NV_Increment
```

31.9 TPM2_NV_Extend

31.9.1 General Description

This command extends a value to an area in NV memory that was previously defined by TPM2_NV_DefineSpace.

If *nvIndexType* is not TPM_NT_EXTEND, then the TPM shall return TPM_RC_ATTRIBUTES.

Proper write authorizations are required for this command as determined by TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, TPMA_NV_AUTHWRITE, and the *authPolicy* of the NV Index.

After successful completion of this command, TPMA_NV_WRITTEN for the NV Index will be SET.

NOTE 1 Once SET, TPMA_NV_WRITTEN remains SET until the NV Index is undefined, unless the TPMA_NV_CLEAR_STCLEAR attribute is SET and a TPM Reset or TPM Restart occurs.

If the TPMA_NV_WRITELOCKED attribute of the NV Index is SET, then the TPM shall return TPM_RC_NV_LOCKED.

NOTE 2 If authorization sessions are present, they are checked before checks to see if writes to the NV Index are locked.

The *data.buffer* parameter may be larger than the defined size of the NV Index.

The Index will be updated by:

$$nvIndex{\rightarrow}data_{new} \coloneqq \mathbf{H}_{nameAkg}(nvIndex{\rightarrow}data_{old} \mid\mid data.buffer) \qquad (41)$$

where

$nvIndex{\rightarrow}data_{new}$	the value of the data field in the NV Index after the command returns
$\mathbf{H}_{nameAkg}()$	the hash algorithm indicated in *nvIndex→nameAlg*
$nvIndex{\rightarrow}data_{old}$	the value of the data field in the NV Index before the command is called
$data.buffer$	the data buffer of the command parameter

NOTE 3 If TPMA_NV_WRITTEN is CLEAR, then $nvIndex{\rightarrow}data_{old}$ is a Zero Digest.

31.9.2 Command and Response

Table 224 — TPM2_NV_Extend Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_Extend {NV}
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index to extend Auth Index: None
TPM2B_MAX_NV_BUFFER	data	the data to extend

Table 225 — TPM2_NV_Extend Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

31.9.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "NV_Extend_fp.h"
3  #if CC_NV_Extend  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	the TPMA_NV_EXTEND attribute is not SET in the Index referenced by *nvIndex*
TPM_RC_NV_AUTHORIZATION	the authorization was valid but the authorizing entity (*authHandle*) is not allowed to write to the Index referenced by *nvIndex*
TPM_RC_NV_LOCKED	the Index referenced by *nvIndex* is locked for writing

```
4  TPM_RC
5  TPM2_NV_Extend(
6      NV_Extend_In    *in             // IN: input parameter list
7      )
8  {
9      TPM_RC                  result;
10     NV_REF                  locator;
11     NV_INDEX               *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
12
13     TPM2B_DIGEST           oldDigest;
14     TPM2B_DIGEST           newDigest;
15     HASH_STATE             hashState;
16
17 // Input Validation
18
19     // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
20     // or TPM_RC_NV_LOCKED
21     result = NvWriteAccessChecks(in->authHandle,
22                                  in->nvIndex,
23                                  nvIndex->publicArea.attributes);
24     if(result != TPM_RC_SUCCESS)
25         return result;
26
27     // Make sure that this is an extend index
28     if(!IsNvExtendIndex(nvIndex->publicArea.attributes))
29         return TPM_RCS_ATTRIBUTES + RC_NV_Extend_nvIndex;
30
31 // Internal Data Update
32
33     // Perform the write.
34     oldDigest.t.size = CryptHashGetDigestSize(nvIndex->publicArea.nameAlg);
35     pAssert(oldDigest.t.size <= sizeof(oldDigest.t.buffer));
36     if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
37     {
38         NvGetIndexData(nvIndex, locator, 0, oldDigest.t.size, oldDigest.t.buffer);
39     }
40     else
41     {
42         MemorySet(oldDigest.t.buffer, 0, oldDigest.t.size);
43     }
44     // Start hash
45     newDigest.t.size = CryptHashStart(&hashState, nvIndex->publicArea.nameAlg);
46
47     // Adding old digest
48     CryptDigestUpdate2B(&hashState, &oldDigest.b);
49
50     // Adding new data
```

 Copyright © TCG 2006-2020

```
51      CryptDigestUpdate2B(&hashState, &in->data.b);
52
53      // Complete hash
54      CryptHashEnd2B(&hashState, &newDigest.b);
55
56      // Write extended hash back.
57      // Note, this routine will SET the TPMA_NV_WRITTEN attribute if necessary
58      return NvWriteIndexData(nvIndex, 0, newDigest.t.size, newDigest.t.buffer);
59  }
60  #endif // CC_NV_Extend
```

31.10 TPM2_NV_SetBits

31.10.1 General Description

This command is used to SET bits in an NV Index that was created as a bit field. Any number of bits from 0 to 64 may be SET. The contents of *bits* are ORed with the current contents of the NV Index.

If TPMA_NV_WRITTEN is not SET, then, for the purposes of this command, the NV Index is considered to contain all zero bits and *data* is ORed with that value.

If TPM_NT_BITS is not SET, then the TPM shall return TPM_RC_ATTRIBUTES.

After successful completion of this command, TPMA_NV_WRITTEN for the NV Index will be SET.

NOTE TPMA_NV_WRITTEN will be SET even if no bits were SET.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.10.2 Command and Response

Table 226 — TPM2_NV_SetBits Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_SetBits {NV}
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	NV Index of the area in which the bit is to be set Auth Index: None
UINT64	bits	the data to OR with the current contents

Table 227 — TPM2_NV_SetBits Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

31.10.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "NV_SetBits_fp.h"
3  #if CC_NV_SetBits  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	the TPMA_NV_BITS attribute is not SET in the Index referenced by *nvIndex*
TPM_RC_NV_AUTHORIZATION	the authorization was valid but the authorizing entity (*authHandle*) is not allowed to write to the Index referenced by *nvIndex*
TPM_RC_NV_LOCKED	the Index referenced by *nvIndex* is locked for writing

```
4  TPM_RC
5  TPM2_NV_SetBits(
6      NV_SetBits_In   *in             // IN: input parameter list
7      )
8  {
9      TPM_RC           result;
10     NV_REF           locator;
11     NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
12     UINT64           oldValue;
13     UINT64           newValue;
14
15 // Input Validation
16
17     // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
18     // or TPM_RC_NV_LOCKED
19     result = NvWriteAccessChecks(in->authHandle,
20                                  in->nvIndex,
21                                  nvIndex->publicArea.attributes);
22     if(result != TPM_RC_SUCCESS)
23         return result;
24
25     // Make sure that this is a bit field
26     if(!IsNvBitsIndex(nvIndex->publicArea.attributes))
27         return TPM_RCS_ATTRIBUTES + RC_NV_SetBits_nvIndex;
28
29     // If index is not been written, initialize it
30     if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
31         oldValue = 0;
32     else
33         // Read index data
34         oldValue = NvGetUINT64Data(nvIndex, locator);
35
36     // Figure out what the new value is going to be
37     newValue = oldValue | in->bits;
38
39 // Internal Data Update
40     return  NvWriteUINT64Data(nvIndex, newValue);
41 }
42 #endif // CC_NV_SetBits
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.11 TPM2_NV_WriteLock

31.11.1 General Description

If the TPMA_NV_WRITEDEFINE or TPMA_NV_WRITE_STCLEAR attributes of an NV location are SET, then this command may be used to inhibit further writes of the NV Index.

Proper write authorization is required for this command as determined by TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, TPMA_NV_AUTHWRITE, and the *authPolicy* of the NV Index.

It is not an error if TPMA_NV_WRITELOCKED for the NV Index is already SET.

If neither TPMA_NV_WRITEDEFINE nor TPMA_NV_WRITE_STCLEAR of the NV Index is SET, then the TPM shall return TPM_RC_ATTRIBUTES.

If the command is properly authorized and TPMA_NV_WRITE_STCLEAR or TPMA_NV_WRITEDEFINE is SET, then the TPM shall SET TPMA_NV_WRITELOCKED for the NV Index. TPMA_NV_WRITELOCKED will be clear on the next TPM2_Startup(TPM_SU_CLEAR) if either TPMA_NV_WRITEDEFINE is CLEAR or TPMA_NV_WRITTEN is CLEAR.

Copyright © TCG 2006-2020

31.11.2 Command and Response

Table 228 — TPM2_NV_WriteLock Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_WriteLock {NV}
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index of the area to lock Auth Index: None

Table 229 — TPM2_NV_WriteLock Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

31.11.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "NV_WriteLock_fp.h"
3  #if CC_NV_WriteLock  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	neither TPMA_NV_WRITEDEFINE nor TPMA_NV_WRITE_STCLEAR is SET in Index referenced by *nvIndex*
TPM_RC_NV_AUTHORIZATION	the authorization was valid but the authorizing entity (*authHandle*) is not allowed to write to the Index referenced by *nvIndex*

```
4  TPM_RC
5  TPM2_NV_WriteLock(
6      NV_WriteLock_In     *in             // IN: input parameter list
7      )
8  {
9      TPM_RC           result;
10     NV_REF           locator;
11     NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
12     TPMA_NV          nvAttributes = nvIndex->publicArea.attributes;
13
14 // Input Validation:
15
16     // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
17     // or TPM_RC_NV_LOCKED
18     result = NvWriteAccessChecks(in->authHandle, in->nvIndex, nvAttributes);
19     if(result != TPM_RC_SUCCESS)
20     {
21         if(result == TPM_RC_NV_AUTHORIZATION)
22             return result;
23         // If write access failed because the index is already locked, then it is
24         // no error.
25         return TPM_RC_SUCCESS;
26     }
27     // if neither TPMA_NV_WRITEDEFINE nor TPMA_NV_WRITE_STCLEAR is set, the index
28     // can not be write-locked
29     if(!IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITEDEFINE)
30        && !IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITE_STCLEAR))
31         return TPM_RCS_ATTRIBUTES + RC_NV_WriteLock_nvIndex;
32 // Internal Data Update
33     // Set the WRITELOCK attribute.
34     // Note: if TPMA_NV_WRITELOCKED were already SET, then the write access check
35     // above would have failed and this code isn't executed.
36     SET_ATTRIBUTE(nvAttributes, TPMA_NV, WRITELOCKED);
37
38     // Write index info back
39     return NvWriteIndexAttributes(nvIndex->publicArea.nvIndex, locator,
40                                   nvAttributes);
41 }
42 #endif // CC_NV_WriteLock
```

 Copyright © TCG 2006-2020

31.12 TPM2_NV_GlobalWriteLock

31.12.1 General Description

The command will SET TPMA_NV_WRITELOCKED for all indexes that have their TPMA_NV_GLOBALLOCK attribute SET.

If an Index has both TPMA_NV_GLOBALLOCK and TPMA_NV_WRITEDEFINE SET, then this command will permanently lock the NV Index for writing unless TPMA_NV_WRITTEN is CLEAR.

NOTE If an Index is defined with TPMA_NV_GLOBALLOCK SET, then the global lock does not apply until the next time this command is executed.

This command requires either platformAuth/platformPolicy or ownerAuth/ownerPolicy.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.12.2 Command and Response

Table 230 — TPM2_NV_GlobalWriteLock Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_GlobalWriteLock {NV}
TPMI_RH_PROVISION	@authHandle	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER

Table 231 — TPM2_NV_GlobalWriteLock Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

31.12.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "NV_GlobalWriteLock_fp.h"
3  #if CC_NV_GlobalWriteLock  // Conditional expansion of this file
4  TPM_RC
5  TPM2_NV_GlobalWriteLock(
6      NV_GlobalWriteLock_In   *in             // IN: input parameter list
7      )
8  {
9      // Input parameter (the authorization handle) is not reference in command action.
10     NOT_REFERENCED(in);
11
12 // Internal Data Update
13
14     // Implementation dependent method of setting the global lock
15     return NvSetGlobalLock();
16 }
17 #endif // CC_NV_GlobalWriteLock
```

 Copyright © TCG 2006-2020

31.13 TPM2_NV_Read

31.13.1 General Description

This command reads a value from an area in NV memory previously defined by TPM2_NV_DefineSpace().

Proper authorizations are required for this command as determined by TPMA_NV_PPREAD, TPMA_NV_OWNERREAD, TPMA_NV_AUTHREAD, and the *authPolicy* of the NV Index.

If TPMA_NV_READLOCKED of the NV Index is SET, then the TPM shall return TPM_RC_NV_LOCKED.

If *offset* and the *size* field of *data* add to a value that is greater than the *dataSize* field of the NV Index referenced by *nvIndex*, the TPM shall return an error (TPM_RC_NV_RANGE). The implementation may return an error (TPM_RC_VALUE) if it performs an additional check and determines that *offset* is greater than the *dataSize* field of the NV Index.

For an NV Index with the TPM_NT_COUNTER or TPM_NT_BITS attribute SET, the TPM may ignore the *offset* parameter and use an offset of 0. Therefore, it is recommended that the caller set the *offset* parameter to 0 for interoperability.

NOTE 1 If authorization sessions are present, they are checked before the read-lock status of the NV Index is checked.

If the NV Index has been defined but the TPMA_NV_WRITTEN attribute is CLEAR, then this command shall return TPM_RC_NV_UNINITIALIZED even if *size* is zero.

The *data* parameter in the response may be encrypted using parameter encryption.

 Copyright © TCG 2006-2020

31.13.2 Command and Response

Table 232 — TPM2_NV_Read Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_Read
TPMI_RH_NV_AUTH	@authHandle	the handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index to be read Auth Index: None
UINT16	size	number of octets to read
UINT16	offset	octet offset into the NV area This value shall be less than or equal to the size of the *nvIndex* data.

Table 233 — TPM2_NV_Read Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_MAX_NV_BUFFER	data	the data read

 Copyright © TCG 2006-2020

31.13.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "NV_Read_fp.h"
3  #if CC_NV_Read  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_NV_AUTHORIZATION	the authorization was valid but the authorizing entity (*authHandle*) is not allowed to read from the Index referenced by *nvIndex*
TPM_RC_NV_LOCKED	the Index referenced by *nvIndex* is read locked
TPM_RC_NV_RANGE	read range defined by *size* and *offset* is outside the range of the Index referenced by *nvIndex*
TPM_RC_NV_UNINITIALIZED	the Index referenced by *nvIndex* has not been initialized (written)
TPM_RC_VALUE	the read size is larger than the MAX_NV_BUFFER_SIZE

```
4  TPM_RC
5  TPM2_NV_Read(
6      NV_Read_In      *in,             // IN: input parameter list
7      NV_Read_Out     *out             // OUT: output parameter list
8      )
9  {
10     NV_REF           locator;
11     NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
12     TPM_RC           result;
13
14 // Input Validation
15     // Common read access checks. NvReadAccessChecks() may return
16     // TPM_RC_NV_AUTHORIZATION, TPM_RC_NV_LOCKED, or TPM_RC_NV_UNINITIALIZED
17     result = NvReadAccessChecks(in->authHandle, in->nvIndex,
18                                 nvIndex->publicArea.attributes);
19     if(result != TPM_RC_SUCCESS)
20         return result;
21
22     // Make sure the data will fit the return buffer
23     if(in->size > MAX_NV_BUFFER_SIZE)
24         return TPM_RCS_VALUE + RC_NV_Read_size;
25
26     // Verify that the offset is not too large
27     if(in->offset > nvIndex->publicArea.dataSize)
28         return TPM_RCS_VALUE + RC_NV_Read_offset;
29
30     // Make sure that the selection is within the range of the Index
31     if(in->size > (nvIndex->publicArea.dataSize - in->offset))
32         return TPM_RC_NV_RANGE;
33
34 // Command Output
35     // Set the return size
36     out->data.t.size = in->size;
37
38     // Perform the read
39     NvGetIndexData(nvIndex, locator, in->offset, in->size, out->data.t.buffer);
40
41     return TPM_RC_SUCCESS;
42 }
43 #endif // CC_NV_Read
```

 Copyright © TCG 2006-2020

31.14 TPM2_NV_ReadLock

31.14.1 General Description

If TPMA_NV_READ_STCLEAR is SET in an Index, then this command may be used to prevent further reads of the NV Index until the next TPM2_Startup (TPM_SU_CLEAR).

Proper authorizations are required for this command as determined by TPMA_NV_PPREAD, TPMA_NV_OWNERREAD, TPMA_NV_AUTHREAD, and the *authPolicy* of the NV Index.

NOTE Only an entity that may read an Index is allowed to lock the NV Index for read.

If the command is properly authorized and TPMA_NV_READ_STCLEAR of the NV Index is SET, then the TPM shall SET TPMA_NV_READLOCKED for the NV Index. If TPMA_NV_READ_STCLEAR of the NV Index is CLEAR, then the TPM shall return TPM_RC_ATTRIBUTES. TPMA_NV_READLOCKED will be CLEAR by the next TPM2_Startup(TPM_SU_CLEAR).

It is not an error to use this command for an Index that is already locked for reading.

An Index that had not been written may be locked for reading.

31.14.2 Command and Response

Table 234 — TPM2_NV_ReadLock Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_ReadLock {NV}
TPMI_RH_NV_AUTH	@authHandle	the handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index to be locked Auth Index: None

Table 235 — TPM2_NV_ReadLock Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

31.14.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "NV_ReadLock_fp.h"
3  #if CC_NV_ReadLock  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	TPMA_NV_READ_STCLEAR is not SET so Index referenced by *nvIndex* may not be write locked
TPM_RC_NV_AUTHORIZATION	the authorization was valid but the authorizing entity (*authHandle*) is not allowed to read from the Index referenced by *nvIndex*

```
4  TPM_RC
5  TPM2_NV_ReadLock(
6      NV_ReadLock_In   *in              // IN: input parameter list
7      )
8  {
9      TPM_RC           result;
10     NV_REF           locator;
11     // The referenced index has been checked multiple times before this is called
12     // so it must be present and will be loaded into cache
13     NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
14     TPMA_NV          nvAttributes = nvIndex->publicArea.attributes;
15
16 // Input Validation
17     // Common read access checks. NvReadAccessChecks() may return
18     // TPM_RC_NV_AUTHORIZATION, TPM_RC_NV_LOCKED, or TPM_RC_NV_UNINITIALIZED
19     result = NvReadAccessChecks(in->authHandle,
20                                 in->nvIndex,
21                                 nvAttributes);
22     if(result == TPM_RC_NV_AUTHORIZATION)
23         return TPM_RC_NV_AUTHORIZATION;
24     // Index is already locked for write
25     else if(result == TPM_RC_NV_LOCKED)
26             return TPM_RC_SUCCESS;
27
28     // If NvReadAccessChecks return TPM_RC_NV_UNINITALIZED, then continue.
29     // It is not an error to read lock an uninitialized Index.
30
31     // if TPMA_NV_READ_STCLEAR is not set, the index can not be read-locked
32     if(!IS_ATTRIBUTE(nvAttributes, TPMA_NV, READ_STCLEAR))
33         return TPM_RCS_ATTRIBUTES + RC_NV_ReadLock_nvIndex;
34
35 // Internal Data Update
36
37     // Set the READLOCK attribute
38     SET_ATTRIBUTE(nvAttributes, TPMA_NV, READLOCKED);
39
40     // Write NV info back
41     return NvWriteIndexAttributes(nvIndex->publicArea.nvIndex,
42                                   locator,
43                                   nvAttributes);
44 }
45 #endif // CC_NV_ReadLock
```

 Copyright © TCG 2006-2020

31.15 TPM2_NV_ChangeAuth

31.15.1 General Description

This command allows the authorization secret for an NV Index to be changed.

If successful, the authorization secret (*authValue*) of the NV Index associated with *nvIndex* is changed.

This command requires that a policy session be used for authorization of *nvIndex* so that the ADMIN role may be asserted and that *commandCode* in the policy session context shall be TPM_CC_NV_ChangeAuth. That is, the policy must contain a specific authorization for changing the authorization value of the referenced entity.

NOTE The reason for this restriction is to ensure that the administrative actions on *nvIndex* require explicit approval while other commands may use policy that is not command-dependent.

The size of the *newAuth* value may be no larger than the size of the digest produced by the *nameAlg* of the NV Index.

Since the NV Index authorization is changed before the response HMAC is calculated, the newAuth value is used when generating the response HMAC key if required. See TPM 2.0 Part 4 ComputeResponseHMAC().

 Copyright © TCG 2006-2020

31.15.2 Command and Response

Table 236 — TPM2_NV_ChangeAuth Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_ChangeAuth {NV}
TPMI_RH_NV_INDEX	@nvIndex	handle of the entity Auth Index: 1 Auth Role: ADMIN
TPM2B_AUTH	newAuth	new authorization value

Table 237 — TPM2_NV_ChangeAuth Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

31.15.3 Detailed Actions

```
1 #include "Tpm.h"
2 #include "NV_ChangeAuth_fp.h"
3 #if CC_NV_ChangeAuth  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_SIZE	*newAuth* size is larger than the digest size of the Name algorithm for the Index referenced by '*nvIndex*

```
4  TPM_RC
5  TPM2_NV_ChangeAuth(
6      NV_ChangeAuth_In    *in              // IN: input parameter list
7      )
8  {
9      NV_REF           locator;
10     NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
11
12 // Input Validation
13
14     // Remove trailing zeros and make sure that the result is not larger than the
15     // digest of the nameAlg.
16     if(MemoryRemoveTrailingZeros(&in->newAuth)
17        > CryptHashGetDigestSize(nvIndex->publicArea.nameAlg))
18         return TPM_RCS_SIZE + RC_NV_ChangeAuth_newAuth;
19
20 // Internal Data Update
21     // Change authValue
22     return NvWriteIndexAuth(locator, &in->newAuth);
23 }
24 #endif // CC_NV_ChangeAuth
```


 Copyright © TCG 2006-2020

31.16 TPM2_NV_Certify

31.16.1 General Description

The purpose of this command is to certify the contents of an NV Index or portion of an NV Index.

If the *sign* attribute is not SET in the key referenced by *signHandle* then the TPM shall return TPM_RC_KEY.

If the NV Index has been defined but the TPMA_NV_WRITTEN attribute is CLEAR, then this command shall return TPM_RC_NV_UNINITIALIZED even if *size* is zero.

If proper authorization for reading the NV Index is provided, the portion of the NV Index selected by *size* and *offset* are included in an attestation block and signed using the key indicated by *signHandle*. The attestation includes *size* and *offset* so that the range of the data can be determined. It also includes the NV index Name.

For an NV Index with the TPM_NT_COUNTER or TPM_NT_BITS attribute SET, the TPM may ignore the *offset* parameter and use an offset of 0. Therefore, it is recommended that the caller set the *offset* parameter to 0 for interoperability.

If *offset* and *size* add to a value that is greater than the *dataSize* field of the NV Index referenced by *nvIndex*, the TPM shall return an error (TPM_RC_NV_RANGE). The implementation may return an error (TPM_RC_VALUE) if it performs an additional check and determines that *offset* is greater than the *dataSize* field of the NV Index, or if *size* is greater than MAX_NV_BUFFER_SIZE.

NOTE 1 See 18.1 for description of how the signing scheme is selected.

NOTE 2 If *signHandle* is TPM_RH_NULL, the TPMS_ATTEST structure is returned and *signature* is a NULL Signature.

If *size* and *offset* are both zero (0), then *certifyInfo* in the response will contain a TPMS_NV_DIGEST_CERTIFY_INFO, otherwise, it will contain a TPMS_NV_CERTIFY_INFO. The digest in the TPMS_NV_DIGEST_CERTIFY_INFO is created using the digest of the selected signing scheme.

NOTE 3 TPMS_NV_DIGEST_CERTIFY_INFO was added in revision 01.53. It permits TPM2_NV_Certify() to certify NV Index contents that are larger than MAX_NV_BUFFER_SIZE.

 Copyright © TCG 2006-2020

31.16.2 Command and Response

Table 238 — TPM2_NV_Certify Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_Certify
TPMI_DH_OBJECT+	@signHandle	handle of the key used to sign the attestation structure Auth Index: 1 Auth Role: USER
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value for the NV Index Auth Index: 2 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	Index for the area to be certified Auth Index: None
TPM2B_DATA	qualifyingData	user-provided qualifying data
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL
UINT16	size	number of octets to certify
UINT16	offset	octet offset into the NV area This value shall be less than or equal to the size of the *nvIndex* data.

Table 239 — TPM2_NV_Certify Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPM2B_ATTEST	certifyInfo	the structure that was signed
TPMT_SIGNATURE	signature	the asymmetric signature over *certifyInfo* using the key referenced by *signHandle*

 Copyright © TCG 2006-2020

31.16.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "Attest_spt_fp.h"
3  #include "NV_Certify_fp.h"
4  #if CC_NV_Certify  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_NV_AUTHORIZATION	the authorization was valid but the authorizing entity (*authHandle*) is not allowed to read from the Index referenced by *nvIndex*
TPM_RC_KEY	*signHandle* does not reference a signing key
TPM_RC_NV_LOCKED	Index referenced by *nvIndex* is locked for reading
TPM_RC_NV_RANGE	*offset* plus *size* extends outside of the data range of the Index referenced by *nvIndex*
TPM_RC_NV_UNINITIALIZED	Index referenced by *nvIndex* has not been written
TPM_RC_SCHEME	*inScheme* is not an allowed value for the key definition

```
5  TPM_RC
6  TPM2_NV_Certify(
7      NV_Certify_In   *in,            // IN: input parameter list
8      NV_Certify_Out  *out            // OUT: output parameter list
9      )
10 {
11     TPM_RC                  result;
12     NV_REF                  locator;
13     NV_INDEX               *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
14     TPMS_ATTEST             certifyInfo;
15     OBJECT                 *signObject = HandleToObject(in->signHandle);
16 // Input Validation
17     if(!IsSigningObject(signObject))
18         return TPM_RCS_KEY + RC_NV_Certify_signHandle;
19     if(!CryptSelectSignScheme(signObject, &in->inScheme))
20         return TPM_RCS_SCHEME + RC_NV_Certify_inScheme;
21
22     // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
23     // or TPM_RC_NV_LOCKED
24     result = NvReadAccessChecks(in->authHandle, in->nvIndex,
25                                 nvIndex->publicArea.attributes);
26     if(result != TPM_RC_SUCCESS)
27         return result;
28
29     // make sure that the selection is within the range of the Index (cast to avoid
30     // any wrap issues with addition)
31     if((UINT32)in->size + (UINT32)in->offset > (UINT32)nvIndex->publicArea.dataSize)
32         return TPM_RC_NV_RANGE;
33     // Make sure the data will fit the return buffer.
34     // NOTE: This check may be modified if the output buffer will not hold the
35     // maximum sized NV buffer as part of the certified data. The difference in
36     // size could be substantial if the signature scheme was produced a large
37     // signature (e.g., RSA 4096).
38     if(in->size > MAX_NV_BUFFER_SIZE)
39         return TPM_RCS_VALUE + RC_NV_Certify_size;
40
41 // Command Output
42
43     // Fill in attest information common fields
44     FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData,
45                      &certifyInfo);
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

```
46
47        // Get the name of the index
48        NvGetIndexName(nvIndex, &certifyInfo.attested.nv.indexName);
49
50        // See if this is old format or new format
51        if ((in->size != 0) || (in->offset != 0))
52        {
53            // NV certify specific fields
54            // Attestation type
55            certifyInfo.type = TPM_ST_ATTEST_NV;
56
57            // Set the return size
58            certifyInfo.attested.nv.nvContents.t.size = in->size;
59
60            // Set the offset
61            certifyInfo.attested.nv.offset = in->offset;
62
63            // Perform the read
64            NvGetIndexData(nvIndex, locator, in->offset, in->size,
65                certifyInfo.attested.nv.nvContents.t.buffer);
66        }
67        else
68        {
69            HASH_STATE                  hashState;
70            // This is to sign a digest of the data
71            certifyInfo.type = TPM_ST_ATTEST_NV_DIGEST;
72            // Initialize the hash before calling the function to add the Index data to
73            // the hash.
74            certifyInfo.attested.nvDigest.nvDigest.t.size =
75                CryptHashStart(&hashState, in->inScheme.details.any.hashAlg);
76            NvHashIndexData(&hashState, nvIndex, locator, 0,
77                nvIndex->publicArea.dataSize);
78            CryptHashEnd2B(&hashState, &certifyInfo.attested.nvDigest.nvDigest.b);
79        }
80        // Sign attestation structure.  A NULL signature will be returned if
81        // signObject is NULL.
82        return SignAttestInfo(signObject, &in->inScheme, &certifyInfo,
83                              &in->qualifyingData, &out->certifyInfo, &out->signature);
84    }
85    #endif // CC_NV_Certify
```

 Copyright © TCG 2006-2020

32 Attached Components

32.1 Introduction

This section contains commands that allow interaction with an Attached Component (AC).

NOTE The Attached Component feature was added in revision 01.40.

 Copyright © TCG 2006-2020

32.2 TPM2_AC_GetCapability

32.2.1 General Description

The purpose of this command is to obtain information about an Attached Component referenced by an AC handle.

The returned list contains 0 or more values starting at the first tagged value that is equal to or greater than *capability*.

The list returned in *capabilitiesData* contains tagged values that indicate the type of the value.

The TPM will return the lesser of a) the available values, b) the number requested in *count,* or c) the number that will fit within the available response buffer. If additional values with higher *capability* numbers are available, *moreData* will be YES.

NOTE TPM2_AC_GetCapability() was added in revision 01.40.

 Copyright © TCG 2006-2020

32.2.2 Command and Response

Table 240 — TPM2_AC_GetCapability Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_AC_GetCapability
TPMI_RH_AC	ac	handle indicating the Attached Component Auth Index: None
TPM_AT	capability	starting info type
UINT32	count	maximum number of values to return

Table 241 — TPM2_AC_GetCapability Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPMI_YES_NO	moreData	flag to indicate whether there are more values
TPML_AC_CAPABILITIES	capabilitiesData	list of capabilities

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

32.2.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "AC_GetCapability_fp.h"
3   #include "AC_spt_fp.h"
4   #if CC_AC_GetCapability  // Conditional expansion of this file
5   TPM_RC
6   TPM2_AC_GetCapability(
7       AC_GetCapability_In     *in,            // IN: input parameter list
8       AC_GetCapability_Out    *out            // OUT: output parameter list
9       )
10  {
11  // Command Output
12      out->moreData = AcCapabilitiesGet(in->ac, in->count, &out->capabilitiesData);
13
14      return TPM_RC_SUCCESS;
15  }
16  #endif // CC_AC_GetCapability
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

32.3 TPM2_AC_Send

32.3.1 General Description

The purpose of this command is to send (copy) a loaded object from the TPM to an Attached Component.

The Object referenced by *sendObject* is required to have *fixedTpm, fixedParent,* and *encryptedDuplication* attributes CLEAR (TPM_RC_ATTRIBUTES). Authorization for *sendObject* is required to be a policy session. The *policySession→commandCode* of the policy session context is required to be TPM_CC_AC_Send (TPM_RC_POLICY_FAIL) to demonstrate that the policy is specific for this command.

Authorization to send to the *ac* is provided by the session associated with *authHandle*.

If an NV Alias is not defined for *ac,* then *authHandle* is required to be either TPM_RH_OWNER or TPM_RH_PLATFORM (TPM_RC_HANDLE).

If an NV Alias is defined for *ac*, then the authorization for *authHandle* is required to be compatible with the write authorization attributes (TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE TPMA_NV_AUTHWRITE, and TPMA_NV_POLICYWRITE) in the NV Alias (TPM_RC_NV_AUTHORIZATION).

NOTE 1 If authorization for *authHandle* is the handle of an NV Index, then it is required to be the NV Alias value for *ac* (TPM_RC_NV_AUTHORIZATION).

If authorization succeeds, the TPM will attempt to send *acDataIn* and relevant portions of *sendObject* to the AC referenced by *ac*.

The TPM will return TPM_RC_SUCCESS if it succeeds in performing all the required authorizations and validations. If problems occur in the process of sending the object from the TPM to the AC, the response code will be TPM_RC_SUCCESS with the AC-dependent error reported in acDataOut.

NOTE 2 TPM2_AC_Send() was added in revision 01.40.

 Copyright © TCG 2006-2020

32.3.2 Command and Response

Table 242 — TPM2_AC_Send Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	Tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_AC_Send
TPMI_DH_OBJECT	@sendObject	handle of the object being sent to ac Auth Index: 1 Auth Role: DUP
TPMI_RH_NV_AUTH	@authHandle	the handle indicating the source of the authorization value Auth Index: 2 Auth Role: USER
TPMI_RH_AC	ac	handle indicating the Attached Component to which the object will be sent Auth Index: None
TPM2B_MAX_BUFFER	acDataIn	Optional non sensitive information related to the object

Table 243 — TPM2_AC_Send Response

Type	Name	Description
TPM_ST	Tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMS_AC_OUTPUT	acDataOut	May include AC specific data or information about an error.

 Copyright © TCG 2006-2020

32.3.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "AC_Send_fp.h"
3  #include "AC_spt_fp.h"
4  #if CC_AC_Send  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	key to duplicate has *fixedParent* SET
TPM_RC_HASH	for an RSA key, the *nameAlg* digest size for the *newParent* is not compatible with the key size
TPM_RC_HIERARCHY	*encryptedDuplication* is SET and *newParentHandle* specifies Null Hierarchy
TPM_RC_KEY	*newParentHandle* references invalid ECC key (public point not on the curve)
TPM_RC_SIZE	input encryption key size does not match the size specified in symmetric algorithm
TPM_RC_SYMMETRIC	*encryptedDuplication* is SET but no symmetric algorithm is provided
TPM_RC_TYPE	*newParentHandle* is neither a storage key nor TPM_RH_NULL; or the object has a NULL *nameAlg*
TPM_RC_VALUE	for an RSA *newParent*, the sizes of the digest and the encryption key are too large to be OAEP encoded

```
5  TPM_RC
6  TPM2_AC_Send(
7      AC_Send_In    *in,              // IN: input parameter list
8      AC_Send_Out   *out              // OUT: output parameter list
9  )
10 {
11     NV_REF           locator;
12     TPM_HANDLE       nvAlias = ((in->ac - AC_FIRST) + NV_AC_FIRST);
13     NV_INDEX        *nvIndex = NvGetIndexInfo(nvAlias, &locator);
14     OBJECT          *object = HandleToObject(in->sendObject);
15     TPM_RC           result;
16 // Input validation
17     // If there is an NV alias, then the index must allow the authorization provided
18     if(nvIndex != NULL)
19     {
20         // Common access checks, NvWriteAccessCheck() may return
21         // TPM_RC_NV_AUTHORIZATION or TPM_RC_NV_LOCKED
22         result = NvWriteAccessChecks(in->authHandle, nvAlias,
23                                      nvIndex->publicArea.attributes);
24         if(result != TPM_RC_SUCCESS)
25             return result;
26     }
27     // If 'ac' did not have an alias then the authorization had to be with either
28     // platform or owner authorization. The type of TPMI_RH_NV_AUTH only allows
29     // owner or platform or an NV index. If it was a valid index, it would have had
30     // an alias and be processed above, so only success here is if this is a
31     // permanent handle.
32     else if(HandleGetType(in->authHandle) != TPM_HT_PERMANENT)
33         return TPM_RCS_HANDLE + RC_AC_Send_authHandle;
34     // Make sure that the object to be duplicated has the right attributes
35     if(IS_ATTRIBUTE(object->publicArea.objectAttributes,
36                     TPMA_OBJECT, encryptedDuplication)
37        || IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT,
38                        fixedParent)
```

 Copyright © TCG 2006-2020

```
39          || IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, fixedTPM))
40           return TPM_RCS_ATTRIBUTES + RC_AC_Send_sendObject;
41  // Command output
42      // Do the implementation dependent send
43      return AcSendObject(in->ac, object, &out->acDataOut);
44  }
45  #endif // TPM_CC_AC_Send
```

 Copyright © TCG 2006-2020

32.4 TPM2_Policy_AC_SendSelect

32.4.1 General Description

This command allows qualification of the sending (copying) of an Object to an Attached Component (AC). Qualification includes selection of the receiving AC and the method of authentication for the AC, and, in certain circumstances, the Object to be sent may be specified.

If this command is not used in conjunction with TPM2_PolicyAuthorize(), then only the *authHandleName* and *acName* are selected and *includeObject* should be CLEAR.

NOTE 1 In the absence of TPM2_PolicyAuthorize(), a policy session cannot create a *policyDigest* that simultaneously equals the *authPolicy* in an Object and names that Object. This is because the *authPolicy* recorded in an Object is unable to include the Name of the Object as the Name of an Object depends on the Object's *authPolicy*.

NOTE 2 An object's *authPolicy* can incorporate the use of TPM2_PolicyAuthorize(). If the authorizing entity for the TPM2_PolicyAuthorize() command specifies only the *ac* and the *authHandle*, then the resultant *policyDigest* may be applied to the sending of any number of Objects. If the authorizing entity for the TPM2_PolicyAuthorize() specifies also the Name of the Object to be sent, then the resultant *policyDigest* applies only to that specific Object.

If either policySession→cpHash or policySession→nameHash has been previously set, the TPM shall return TPM_RC_CPHASH. Otherwise, policySession→nameHash will be set to:

$$nameHash \coloneqq \mathbf{H}_{policyAlg}(objectName \mathbin{||} authHandleName \mathbin{||} acName) \quad (42)$$

NOTE 3 A policy cannot specify both *cpHash* and *nameHash* because *policySession→nameHash* and *policySession→cpHash* may share the same memory space.

If the command succeeds, *policySession→policyDigest* will be updated according to the setting of the input parameter *includeObject*. If *includeObject* is SET, *policySession→policyDigest* is updated by:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_Policy_AC_SendSelect} \mathbin{||} objectName \mathbin{||} authHandleName \mathbin{||} acName \mathbin{||} includeObject) \quad (43)$$

but if includeObject is CLEAR, *policySession→policyDigest* is updated by:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_Policy_AC_SendSelect} \mathbin{||} authHandleName \mathbin{||} acName \mathbin{||} includeObject) \quad (44)$$

NOTE 4 *policySession→nameHash* receives the digest of all Names so that the check performed in TPM2_AC_Send() may be the same regardless of which Names are included in *policySession→policyDigest*. This means that, when TPM2_Policy_AC_SendSelect() is executed, it is only valid for a specific triple of *objectName*, *authHandleName*, and *acName*.

If the command succeeds, *policySession→commandCode* is set to TPM_CC_AC_Send.

NOTE 5 The normal use of TPM2_Policy_AC_SendSelect() is before a TPM2_PolicyAuthorize(). An authorized entity would approve a *policyDigest* that allows sending to a specific Attached Component. The authorizing entity may want to limit the authorization so that the approval allows only a specific Object to be sent to the Attached Component. In that case, the authorizing entity would approve the *policyDigest* of *equation* (44).

NOTE 6 TPM2_Policy_AC_SendSelect() was added in revision 01.40.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

32.4.2 Command and Response

Table 244 — TPM2_Policy_AC_SendSelect Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	Tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Policy_AC_SendSelect
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_NAME	objectName	the Name of the Object to be sent
TPM2B_NAME	authHandleName	the Name associated with *authHandle* used in the TPM2_AC_Send() command
TPM2B_NAME	acName	the Name of the Attached Component to which the Object will be sent
TPMI_YES_NO	includeObject	if SET, *objectName* will be included in the value in *policySession→policyDigest*

Table 245 — TPM2_Policy_AC_SendSelect Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

32.4.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "Policy_AC_SendSelect_fp.h"
3  #if CC_Policy_AC_SendSelect      // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_COMMAND_CODE	*commandCode* of '*policySession*; is not empty
TPM_RC_CPHASH	*cpHash* of *policySession* is not empty

```
4  TPM_RC
5  TPM2_Policy_AC_SendSelect(
6      Policy_AC_SendSelect_In  *in               // IN: input parameter list
7      )
8  {
9      SESSION          *session;
10     HASH_STATE       hashState;
11     TPM_CC           commandCode = TPM_CC_Policy_AC_SendSelect;
12
13 // Input Validation
14
15     // Get pointer to the session structure
16     session = SessionGet(in->policySession);
17
18     // cpHash in session context must be empty
19     if(session->u1.cpHash.t.size != 0)
20         return TPM_RC_CPHASH;
21     // commandCode in session context must be empty
22     if(session->commandCode != 0)
23         return TPM_RC_COMMAND_CODE;
24 // Internal Data Update
25     // Update name hash
26     session->u1.cpHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
27
28     //  add objectName
29     CryptDigestUpdate2B(&hashState, &in->objectName.b);
30
31     // add authHandleName
32     CryptDigestUpdate2B(&hashState, &in->authHandleName.b);
33
34     //  add ac name
35     CryptDigestUpdate2B(&hashState, &in->acName.b);
36
37     //  complete hash
38     CryptHashEnd2B(&hashState, &session->u1.cpHash.b);
39
40     // update policy hash
41     // Old policyDigest size should be the same as the new policyDigest size since
42     // they are using the same hash algorithm
43     session->u2.policyDigest.t.size
44         = CryptHashStart(&hashState, session->authHashAlg);
45 //  add old policy
46     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
47
48     //  add command code
49     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
50
51     //  add objectName
52     if(in->includeObject == YES)
53         CryptDigestUpdate2B(&hashState, &in->objectName.b);
54
```


Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

```
55      //  add authHandleName
56      CryptDigestUpdate2B(&hashState, &in->authHandleName.b);
57
58      // add acName
59      CryptDigestUpdate2B(&hashState, &in->acName.b);
60
61      //  add includeObject
62      CryptDigestUpdateInt(&hashState, sizeof(TPMI_YES_NO), in->includeObject);
63
64      //  complete digest
65      CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
66
67      // set commandCode in session context
68      session->commandCode = TPM_CC_AC_Send;
69
70      return TPM_RC_SUCCESS;
71  }
72  #endif // CC_Policy_AC_SendSelect
```

33 Authenticated Countdown Timer

33.1 Introduction

This section contains commands that allow interaction with an Authenticated Countdown Timer (ACT).

NOTE The Authenticated Countdown Timer was added in revision 01.56.

33.2 TPM2_ACT_SetTimeout

33.2.1 General Description

This command is used to set the time remaining before an Authenticated Countdown Timer (ACT) expires.

This command sets TPMS_ACT_DATA.*timeout* (ACT Timeout) to *startTimeout*. The *startTimeout* value is an integer number of seconds and may be zero. The *startTimeout* parameter may be greater, equal, or less than the current value of ACT Timeout.

When ACT Timeout is non-zero, it will count down, once per second until it reaches zero, at which time the *signaled* attribute of the TPMA_ACT associated with *actHandle* is SET.

When ACT Timeout is zero and the *signaled* attribute is SET, writing a *startTimeout* of FF FF FF FF_{16} will clear *signaled* and stop the counting.

There are four states for ACT Timeout and *startTimeout*. The *signaled* attribute will be set as follows:

1) If ACT Timeout is zero and *startTimeout* is non-zero, then *signaled* will be CLEAR.

2) If ACT Timeout is non-zero and *startTimeout* is non-zero, then *signaled* will be CLEAR.

3) If ACT Timeout is zero and *startTimeout* is zero, then *signaled* will be unchanged.

4) If ACT Timeout is non-zero and *startTimeout* is zero, then *signaled* will be SET.

NOTE 1 The ACT signals on a transition from non-zero to zero. The transition can occur either due to TPM2_ACT_SetTimeout() or a decrement. The effect of *signaled* is platform dependent.

NOTE 2 It may take up to one second until ACT Timeout will be set and *signaled* will be CLEAR or SET by TPM2_ACT_SetTimeout() or TPM2_Startup(STATE). This allows the counting and signaling to take place synchronously with the hardware clock tick.

NOTE 3 TPM2_ACT_SetTimeout() was added in revision 01.56.

 Copyright © TCG 2006-2020

33.2.2 Command and Response

Table 246 — TPM2_ACT_SetTimeout Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ACT_SetTimeout
TPMI_RH_ACT	@actHandle	Handle of the selected ACT Auth Index: 1 Auth Role: USER
UINT32	startTimeout	the start timeout value for the ACT in seconds

Table 247 — TPM2_ACT_SetTimeout Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

33.2.3 Detailed Actions

```
1  #include "Tpm.h"
2  #include "ACT_SetTimeout_fp.h"
3  #if CC_ACT_SetTimeout  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_RETRY	returned when an update for the selected ACT is already pending
TPM_RC_VALUE	attempt to disable signaling from an ACT that has not expired

```
4  TPM_RC
5  TPM2_ACT_SetTimeout(
6      ACT_SetTimeout_In      *in             // IN: input parameter list
7      )
8  {
9      // If 'startTimeout' is UINT32_MAX, then this is an attempt to disable the ACT
10     // and turn off the signaling for the ACT. This is only valid if the ACT
11     // is signaling.
12     if((in->startTimeout == UINT32_MAX) && !ActGetSignaled(in->actHandle))
13         return TPM_RC_VALUE + RC_ACT_SetTimeout_startTimeout;
14     return ActCounterUpdate(in->actHandle, in->startTimeout);
15 }
16 #endif // CC_ACT_SetTimeout
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

34 Vendor Specific

34.1 Introduction

This section contains commands that are vendor specific but made public in order to prevent proliferation.

This specification does define TPM2_Vendor_TCG_Test() in order to have at least one command that can be used to ensure the proper operation of the command dispatch code when processing a vendor-specific command.

34.2 TPM2_Vendor_TCG_Test

34.2.1 General Description

This is a placeholder to allow testing of the dispatch code.

 Copyright © TCG 2006-2020

34.2.2 Command and Response

Table 248 — TPM2_Vendor_TCG_Test Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Vendor_TCG_Test
TPM2B_DATA	inputData	dummy data

Table 249 — TPM2_Vendor_TCG_Test Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	TPM_RC_SUCCESS
TPM2B_DATA	outputData	dummy data

 Copyright © TCG 2006-2020

34.2.3 Detailed Actions

```
1  #include "Tpm.h"
2  #if CC_Vendor_TCG_Test      // Conditional expansion of this file
3  #include "Vendor_TCG_Test_fp.h"
4  TPM_RC
5  TPM2_Vendor_TCG_Test(
6      Vendor_TCG_Test_In        *in,             // IN: input parameter list
7      Vendor_TCG_Test_Out       *out             // OUT: output parameter list
8      )
9  {
10     out->outputData = in->inputData;
11     return TPM_RC_SUCCESS;
12 }
13 #endif // CC_Vendor_TCG_Test
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

Trusted Platform Module Library

Part 3: Commands

Family "2.0"

Level 00 Revision 01.59

November 8, 2019

Published

Contact: admin@trustedcomputinggroup.org

TCG Published

Copyright © TCG 2006-2020

Licenses and Notices

Copyright Licenses:

- Trusted Computing Group (TCG) grants to the user of the source code in this specification (the "Source Code") a worldwide, irrevocable, nonexclusive, royalty free, copyright license to reproduce, create derivative works, distribute, display and perform the Source Code and derivative works thereof, and to grant others the rights granted herein.
- The TCG grants to the user of the other parts of the specification (other than the Source Code) the rights to reproduce, distribute, display, and perform the specification solely for the purpose of developing products based on such documents.

Source Code Distribution Conditions:

- Redistributions of Source Code must retain the above copyright licenses, this list of conditions and the following disclaimers.
- Redistributions in binary form must reproduce the above copyright licenses, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.

Disclaimers:

- THE COPYRIGHT LICENSES SET FORTH ABOVE DO NOT REPRESENT ANY FORM OF LICENSE OR WAIVER, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, WITH RESPECT TO PATENT RIGHTS HELD BY TCG MEMBERS (OR OTHER THIRD PARTIES) THAT MAY BE NECESSARY TO IMPLEMENT THIS SPECIFICATION OR OTHERWISE. Contact TCG Administration (admin@trustedcomputinggroup.org) for information on specification licensing rights available through TCG membership agreements.
- THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO EXPRESS OR IMPLIED WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, ACCURACY, COMPLETENESS, OR NONINFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.
- Without limitation, TCG and its members and licensors disclaim all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification and to the implementation of this specification, and TCG disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this specification or any information herein.

Any marks and brands contained herein are the property of their respective owners.

CONTENTS

1 Scope ... 1
2 Terms and Definitions ... 1
3 Symbols and abbreviated terms ... 1
4 Notation ... 2
4.1 Introduction ... 2
4.2 Table Decorations ... 2
4.3 Handle and Parameter Demarcation ... 3
4.4 AuthorizationSize and ParameterSize ... 3
4.5 Return Code Alias ... 4
5 Command Processing ... 4
5.1 Introduction ... 4
5.2 Command Header Validation ... 4
5.3 Mode Checks ... 5
5.4 Handle Area Validation ... 5
5.5 Session Area Validation ... 6
5.6 Authorization Checks ... 7
5.7 Parameter Decryption ... 9
5.8 Parameter Unmarshaling ... 9
5.9 Command Post Processing ... 11
6 Response Values ... 12
6.1 Tag ... 12
6.2 Response Codes ... 12
7 Implementation Dependent ... 15
8 Detailed Actions Assumptions ... 16
8.1 Introduction ... 16
8.2 Pre-processing ... 16
8.3 Post Processing ... 16
9 Start-up ... 17
9.1 Introduction ... 17
9.2 _TPM_Init ... 17
9.3 TPM2_Startup ... 19
9.4 TPM2_Shutdown ... 24
10 Testing ... 27
10.1 Introduction ... 27
10.2 TPM2_SelfTest ... 28
10.3 TPM2_IncrementalSelfTest ... 31
10.4 TPM2_GetTestResult ... 34
11 Session Commands ... 37
11.1 TPM2_StartAuthSession ... 37
11.2 TPM2_PolicyRestart ... 41
12 Object Commands ... 44
12.1 TPM2_Create ... 44

 Copyright © TCG 2006-2020

12.2 TPM2_Load ... 49
12.3 TPM2_LoadExternal ... 52
12.4 TPM2_ReadPublic ... 56
12.5 TPM2_ActivateCredential ... 59
12.6 TPM2_MakeCredential ... 62
12.7 TPM2_Unseal ... 65
12.8 TPM2_ObjectChangeAuth ... 68
12.9 TPM2_CreateLoaded ... 71

13 Duplication Commands ... 74

13.1 TPM2_Duplicate ... 74
13.2 TPM2_Rewrap ... 77
13.3 TPM2_Import ... 80

14 Asymmetric Primitives ... 84

14.1 Introduction ... 84
14.2 TPM2_RSA_Encrypt ... 84
14.3 TPM2_RSA_Decrypt ... 88
14.4 TPM2_ECDH_KeyGen ... 91
14.5 TPM2_ECDH_ZGen ... 94
14.6 TPM2_ECC_Parameters ... 97
14.7 TPM2_ZGen_2Phase ... 100

15 Symmetric Primitives ... 103

15.1 Introduction ... 103
15.2 TPM2_EncryptDecrypt ... 105
15.3 TPM2_EncryptDecrypt2 ... 108
15.4 TPM2_Hash ... 111
15.5 TPM2_HMAC ... 114
15.6 TPM2_MAC ... 117

16 Random Number Generator ... 120

16.1 TPM2_GetRandom ... 120
16.2 TPM2_StirRandom ... 123

17 Hash/HMAC/Event Sequences ... 126

17.1 Introduction ... 126
17.2 TPM2_HMAC_Start ... 126
17.3 TPM2_MAC_Start ... 129
17.4 TPM2_HashSequenceStart ... 132
17.5 TPM2_SequenceUpdate ... 135
17.6 TPM2_SequenceComplete ... 138
17.7 TPM2_EventSequenceComplete ... 141

18 Attestation Commands ... 144

18.1 Introduction ... 144
18.2 TPM2_Certify ... 146
18.3 TPM2_CertifyCreation ... 149
18.4 TPM2_Quote ... 152
18.5 TPM2_GetSessionAuditDigest ... 155
18.6 TPM2_GetCommandAuditDigest ... 158

 Copyright © TCG 2006-2020

18.7 TPM2_GetTime ... 161
18.8 TPM2_CertifyX509 ... 163
19 Ephemeral EC Keys ... 167
19.1 Introduction ... 167
19.2 TPM2_Commit ... 168
19.3 TPM2_EC_Ephemeral ... 171
20 Signing and Signature Verification ... 174
20.1 TPM2_VerifySignature ... 174
20.2 TPM2_Sign ... 177
21 Command Audit ... 180
21.1 Introduction ... 180
21.2 TPM2_SetCommandCodeAuditStatus ... 181
22 Integrity Collection (PCR) ... 184
22.1 Introduction ... 184
22.2 TPM2_PCR_Extend ... 185
22.3 TPM2_PCR_Event ... 188
22.4 TPM2_PCR_Read ... 191
22.5 TPM2_PCR_Allocate ... 194
22.6 TPM2_PCR_SetAuthPolicy ... 197
22.7 TPM2_PCR_SetAuthValue ... 200
22.8 TPM2_PCR_Reset ... 203
22.9 _TPM_Hash_Start ... 206
22.10 _TPM_Hash_Data ... 208
22.11 _TPM_Hash_End ... 210
23 Enhanced Authorization (EA) Commands ... 212
23.1 Introduction ... 212
23.2 Signed Authorization Actions ... 213
23.3 TPM2_PolicySigned ... 217
23.4 TPM2_PolicySecret ... 221
23.5 TPM2_PolicyTicket ... 224
23.6 TPM2_PolicyOR ... 227
23.7 TPM2_PolicyPCR ... 230
23.8 TPM2_PolicyLocality ... 234
23.9 TPM2_PolicyNV ... 237
23.10 TPM2_PolicyCounterTimer ... 240
23.11 TPM2_PolicyCommandCode ... 243
23.12 TPM2_PolicyPhysicalPresence ... 246
23.13 TPM2_PolicyCpHash ... 249
23.14 TPM2_PolicyNameHash ... 252
23.15 TPM2_PolicyDuplicationSelect ... 255
23.16 TPM2_PolicyAuthorize ... 258
23.17 TPM2_PolicyAuthValue ... 261
23.18 TPM2_PolicyPassword ... 264
23.19 TPM2_PolicyGetDigest ... 267
23.20 TPM2_PolicyNvWritten ... 270
23.21 TPM2_PolicyTemplate ... 273

 Copyright © TCG 2006-2020

23.22 TPM2_PolicyAuthorizeNV 276
24 Hierarchy Commands 279
24.1 TPM2_CreatePrimary 279
24.2 TPM2_HierarchyControl 282
24.3 TPM2_SetPrimaryPolicy 285
24.4 TPM2_ChangePPS 288
24.5 TPM2_ChangeEPS 291
24.6 TPM2_Clear 294
24.7 TPM2_ClearControl 297
24.8 TPM2_HierarchyChangeAuth 300
25 Dictionary Attack Functions 303
25.1 Introduction 303
25.2 TPM2_DictionaryAttackLockReset 303
25.3 TPM2_DictionaryAttackParameters 306
26 Miscellaneous Management Functions 309
26.1 Introduction 309
26.2 TPM2_PP_Commands 309
26.3 TPM2_SetAlgorithmSet 312
27 Field Upgrade 315
27.1 Introduction 315
27.2 TPM2_FieldUpgradeStart 317
27.3 TPM2_FieldUpgradeData 320
27.4 TPM2_FirmwareRead 323
28 Context Management 326
28.1 Introduction 326
28.2 TPM2_ContextSave 326
28.3 TPM2_ContextLoad 329
28.4 TPM2_FlushContext 332
28.5 TPM2_EvictControl 335
29 Clocks and Timers 339
29.1 TPM2_ReadClock 339
29.2 TPM2_ClockSet 342
29.3 TPM2_ClockRateAdjust 345
30 Capability Commands 348
30.1 Introduction 348
30.2 TPM2_GetCapability 348
30.3 TPM2_TestParms 354
31 Non-volatile Storage 357
31.1 Introduction 357
31.2 NV Counters 359
31.3 TPM2_NV_DefineSpace 360
31.4 TPM2_NV_UndefineSpace 364
31.5 TPM2_NV_UndefineSpaceSpecial 367
31.6 TPM2_NV_ReadPublic 370

 Copyright © TCG 2006-2020

31.7 TPM2_NV_Write 373
31.8 TPM2_NV_Increment 376
31.9 TPM2_NV_Extend 379
31.10 TPM2_NV_SetBits 382
31.11 TPM2_NV_WriteLock 385
31.12 TPM2_NV_GlobalWriteLock 388
31.13 TPM2_NV_Read 391
31.14 TPM2_NV_ReadLock 394
31.15 TPM2_NV_ChangeAuth 397
31.16 TPM2_NV_Certify 400

32 Attached Components 403

32.1 Introduction 403
32.2 TPM2_AC_GetCapability 404
32.3 TPM2_AC_Send 407
32.4 TPM2_Policy_AC_SendSelect 410

33 Authenticated Countdown Timer 413

33.1 Introduction 413
33.2 TPM2_ACT_SetTimeout 413

34 Vendor Specific 416

34.1 Introduction 416
34.2 TPM2_Vendor_TCG_Test 416

 Copyright © TCG 2006-2020

Tables

Table 1 — Command Modifiers and Decoration 2
Table 2 — Separators 3
Table 3 — Unmarshaling Errors 10
Table 4 — Command-Independent Response Codes 13
Table 5 — TPM2_Startup Command 22
Table 6 — TPM2_Startup Response 22
Table 7 — TPM2_Shutdown Command 25
Table 8 — TPM2_Shutdown Response 25
Table 9 — TPM2_SelfTest Command 29
Table 10 — TPM2_SelfTest Response 29
Table 11 — TPM2_IncrementalSelfTest Command 32
Table 12 — TPM2_IncrementalSelfTest Response 32
Table 13 — TPM2_GetTestResult Command 35
Table 14 — TPM2_GetTestResult Response 35
Table 15 — TPM2_StartAuthSession Command 39
Table 16 — TPM2_StartAuthSession Response 39
Table 17 — TPM2_PolicyRestart Command 42
Table 18 — TPM2_PolicyRestart Response 42
Table 19 — TPM2_Create Command 47
Table 20 — TPM2_Create Response 47
Table 21 — TPM2_Load Command 50
Table 22 — TPM2_Load Response 50
Table 23 — TPM2_LoadExternal Command 54
Table 24 — TPM2_LoadExternal Response 54
Table 25 — TPM2_ReadPublic Command 57
Table 26 — TPM2_ReadPublic Response 57
Table 27 — TPM2_ActivateCredential Command 60
Table 28 — TPM2_ActivateCredential Response 60
Table 29 — TPM2_MakeCredential Command 63
Table 30 — TPM2_MakeCredential Response 63
Table 31 — TPM2_Unseal Command 66
Table 32 — TPM2_Unseal Response 66
Table 33 — TPM2_ObjectChangeAuth Command 69
Table 34 — TPM2_ObjectChangeAuth Response 69
Table 35 — TPM2_CreateLoaded Command 72
Table 36 — TPM2_CreateLoaded Response 72
Table 37 — TPM2_Duplicate Command 75

 Copyright © TCG 2006-2020

Table 38 — TPM2_Duplicate Response ... 75
Table 39 — TPM2_Rewrap Command ... 78
Table 40 — TPM2_Rewrap Response ... 78
Table 41 — TPM2_Import Command ... 82
Table 42 — TPM2_Import Response ... 82
Table 43 — Padding Scheme Selection ... 84
Table 44 — Message Size Limits Based on Padding ... 85
Table 45 — TPM2_RSA_Encrypt Command ... 86
Table 46 — TPM2_RSA_Encrypt Response ... 86
Table 47 — TPM2_RSA_Decrypt Command ... 89
Table 48 — TPM2_RSA_Decrypt Response ... 89
Table 49 — TPM2_ECDH_KeyGen Command ... 92
Table 50 — TPM2_ECDH_KeyGen Response ... 92
Table 51 — TPM2_ECDH_ZGen Command ... 95
Table 52 — TPM2_ECDH_ZGen Response ... 95
Table 53 — TPM2_ECC_Parameters Command ... 98
Table 54 — TPM2_ECC_Parameters Response ... 98
Table 55 — TPM2_ZGen_2Phase Command ... 101
Table 56 — TPM2_ZGen_2Phase Response ... 101
Table 57 — Symmetric Chaining Process ... 104
Table 58 — TPM2_EncryptDecrypt Command ... 106
Table 59 — TPM2_EncryptDecrypt Response ... 106
Table 60 — TPM2_EncryptDecrypt2 Command ... 109
Table 61 — TPM2_EncryptDecrypt2 Response ... 109
Table 62 — TPM2_Hash Command ... 112
Table 63 — TPM2_Hash Response ... 112
Table 64 — TPM2_HMAC Command ... 115
Table 65 — TPM2_HMAC Response ... 115
Table 66 — TPM2_MAC Command ... 118
Table 67 — TPM2_MAC Response ... 118
Table 68 — TPM2_GetRandom Command ... 121
Table 69 — TPM2_GetRandom Response ... 121
Table 70 — TPM2_StirRandom Command ... 124
Table 71 — TPM2_StirRandom Response ... 124
Table 72 — Hash Selection Matrix ... 126
Table 73 — TPM2_HMAC_Start Command ... 127
Table 74 — TPM2_HMAC_Start Response ... 127
Table 75 — Algorithm Selection Matrix ... 129
Table 76 — TPM2_MAC_Start Command ... 130

 Copyright © TCG 2006-2020

Table 77 — TPM2_MAC_Start Response 130
Table 78 — TPM2_HashSequenceStart Command 133
Table 79 — TPM2_HashSequenceStart Response 133
Table 80 — TPM2_SequenceUpdate Command 136
Table 81 — TPM2_SequenceUpdate Response 136
Table 82 — TPM2_SequenceComplete Command 139
Table 83 — TPM2_SequenceComplete Response 139
Table 84 — TPM2_EventSequenceComplete Command 142
Table 85 — TPM2_EventSequenceComplete Response 142
Table 86 — TPM2_Certify Command 147
Table 87 — TPM2_Certify Response 147
Table 88 — TPM2_CertifyCreation Command 150
Table 89 — TPM2_CertifyCreation Response 150
Table 90 — TPM2_Quote Command 153
Table 91 — TPM2_Quote Response 153
Table 92 — TPM2_GetSessionAuditDigest Command 156
Table 93 — TPM2_GetSessionAuditDigest Response 156
Table 94 — TPM2_GetCommandAuditDigest Command 159
Table 95 — TPM2_GetCommandAuditDigest Response 159
Table 96 — TPM2_GetTime Command 162
Table 97 — TPM2_GetTime Response 162
Table 98 — TPM2_CertifyX509 Command 165
Table 99 — TPM2_CertifyX509 Response 165
Table 100 — TPM2_Commit Command 169
Table 101 — TPM2_Commit Response 169
Table 102 — TPM2_EC_Ephemeral Command 172
Table 103 — TPM2_EC_Ephemeral Response 172
Table 104 — TPM2_VerifySignature Command 175
Table 105 — TPM2_VerifySignature Response 175
Table 106 — TPM2_Sign Command 178
Table 107 — TPM2_Sign Response 178
Table 108 — TPM2_SetCommandCodeAuditStatus Command 182
Table 109 — TPM2_SetCommandCodeAuditStatus Response 182
Table 110 — TPM2_PCR_Extend Command 186
Table 111 — TPM2_PCR_Extend Response 186
Table 112 — TPM2_PCR_Event Command 189
Table 113 — TPM2_PCR_Event Response 189
Table 114 — TPM2_PCR_Read Command 192
Table 115 — TPM2_PCR_Read Response 192

 Copyright © TCG 2006-2020

Table 116 — TPM2_PCR_Allocate Command ... 195
Table 117 — TPM2_PCR_Allocate Response ... 195
Table 118 — TPM2_PCR_SetAuthPolicy Command ... 198
Table 119 — TPM2_PCR_SetAuthPolicy Response ... 198
Table 120 — TPM2_PCR_SetAuthValue Command ... 201
Table 121 — TPM2_PCR_SetAuthValue Response ... 201
Table 122 — TPM2_PCR_Reset Command ... 204
Table 123 — TPM2_PCR_Reset Response ... 204
Table 124 — TPM2_PolicySigned Command ... 219
Table 125 — TPM2_PolicySigned Response ... 219
Table 126 — TPM2_PolicySecret Command ... 222
Table 127 — TPM2_PolicySecret Response ... 222
Table 128 — TPM2_PolicyTicket Command ... 225
Table 129 — TPM2_PolicyTicket Response ... 225
Table 130 — TPM2_PolicyOR Command ... 228
Table 131 — TPM2_PolicyOR Response ... 228
Table 132 — TPM2_PolicyPCR Command ... 232
Table 133 — TPM2_PolicyPCR Response ... 232
Table 134 — TPM2_PolicyLocality Command ... 235
Table 135 — TPM2_PolicyLocality Response ... 235
Table 136 — TPM2_PolicyNV Command ... 238
Table 137 — TPM2_PolicyNV Response ... 238
Table 138 — TPM2_PolicyCounterTimer Command ... 241
Table 139 — TPM2_PolicyCounterTimer Response ... 241
Table 140 — TPM2_PolicyCommandCode Command ... 244
Table 141 — TPM2_PolicyCommandCode Response ... 244
Table 142 — TPM2_PolicyPhysicalPresence Command ... 247
Table 143 — TPM2_PolicyPhysicalPresence Response ... 247
Table 144 — TPM2_PolicyCpHash Command ... 250
Table 145 — TPM2_PolicyCpHash Response ... 250
Table 146 — TPM2_PolicyNameHash Command ... 253
Table 147 — TPM2_PolicyNameHash Response ... 253
Table 148 — TPM2_PolicyDuplicationSelect Command ... 256
Table 149 — TPM2_PolicyDuplicationSelect Response ... 256
Table 150 — TPM2_PolicyAuthorize Command ... 259
Table 151 — TPM2_PolicyAuthorize Response ... 259
Table 152 — TPM2_PolicyAuthValue Command ... 262
Table 153 — TPM2_PolicyAuthValue Response ... 262
Table 154 — TPM2_PolicyPassword Command ... 265

Table 155 — TPM2_PolicyPassword Response ... 265
Table 156 — TPM2_PolicyGetDigest Command ... 268
Table 157 — TPM2_PolicyGetDigest Response ... 268
Table 158 — TPM2_PolicyNvWritten Command ... 271
Table 159 — TPM2_PolicyNvWritten Response ... 271
Table 160 — TPM2_PolicyTemplate Command ... 274
Table 161 — TPM2_PolicyTemplate Response ... 274
Table 162 — TPM2_PolicyAuthorizeNV Command ... 277
Table 163 — TPM2_PolicyAuthorizeNV Response ... 277
Table 164 — TPM2_CreatePrimary Command ... 280
Table 165 — TPM2_CreatePrimary Response ... 280
Table 166 — TPM2_HierarchyControl Command ... 283
Table 167 — TPM2_HierarchyControl Response ... 283
Table 168 — TPM2_SetPrimaryPolicy Command ... 286
Table 169 — TPM2_SetPrimaryPolicy Response ... 286
Table 170 — TPM2_ChangePPS Command ... 289
Table 171 — TPM2_ChangePPS Response ... 289
Table 172 — TPM2_ChangeEPS Command ... 292
Table 173 — TPM2_ChangeEPS Response ... 292
Table 174 — TPM2_Clear Command ... 295
Table 175 — TPM2_Clear Response ... 295
Table 176 — TPM2_ClearControl Command ... 298
Table 177 — TPM2_ClearControl Response ... 298
Table 178 — TPM2_HierarchyChangeAuth Command ... 301
Table 179 — TPM2_HierarchyChangeAuth Response ... 301
Table 180 — TPM2_DictionaryAttackLockReset Command ... 304
Table 181 — TPM2_DictionaryAttackLockReset Response ... 304
Table 182 — TPM2_DictionaryAttackParameters Command ... 307
Table 183 — TPM2_DictionaryAttackParameters Response ... 307
Table 184 — TPM2_PP_Commands Command ... 310
Table 185 — TPM2_PP_Commands Response ... 310
Table 186 — TPM2_SetAlgorithmSet Command ... 313
Table 187 — TPM2_SetAlgorithmSet Response ... 313
Table 188 — TPM2_FieldUpgradeStart Command ... 318
Table 189 — TPM2_FieldUpgradeStart Response ... 318
Table 190 — TPM2_FieldUpgradeData Command ... 321
Table 191 — TPM2_FieldUpgradeData Response ... 321
Table 192 — TPM2_FirmwareRead Command ... 324
Table 193 — TPM2_FirmwareRead Response ... 324

 Copyright © TCG 2006-2020

Table 194 — TPM2_ContextSave Command 327
Table 195 — TPM2_ContextSave Response 327
Table 196 — TPM2_ContextLoad Command 330
Table 197 — TPM2_ContextLoad Response 330
Table 198 — TPM2_FlushContext Command 333
Table 199 — TPM2_FlushContext Response 333
Table 200 — TPM2_EvictControl Command 337
Table 201 — TPM2_EvictControl Response 337
Table 202 — TPM2_ReadClock Command 340
Table 203 — TPM2_ReadClock Response 340
Table 204 — TPM2_ClockSet Command 343
Table 205 — TPM2_ClockSet Response 343
Table 206 — TPM2_ClockRateAdjust Command 346
Table 207 — TPM2_ClockRateAdjust Response 346
Table 208 — TPM2_GetCapability Command 352
Table 209 — TPM2_GetCapability Response 352
Table 210 — TPM2_TestParms Command 355
Table 211 — TPM2_TestParms Response 355
Table 212 — TPM2_NV_DefineSpace Command 362
Table 213 — TPM2_NV_DefineSpace Response 362
Table 214 — TPM2_NV_UndefineSpace Command 365
Table 215 — TPM2_NV_UndefineSpace Response 365
Table 216 — TPM2_NV_UndefineSpaceSpecial Command 368
Table 217 — TPM2_NV_UndefineSpaceSpecial Response 368
Table 218 — TPM2_NV_ReadPublic Command 371
Table 219 — TPM2_NV_ReadPublic Response 371
Table 220 — TPM2_NV_Write Command 374
Table 221 — TPM2_NV_Write Response 374
Table 222 — TPM2_NV_Increment Command 377
Table 223 — TPM2_NV_Increment Response 377
Table 224 — TPM2_NV_Extend Command 380
Table 225 — TPM2_NV_Extend Response 380
Table 226 — TPM2_NV_SetBits Command 383
Table 227 — TPM2_NV_SetBits Response 383
Table 228 — TPM2_NV_WriteLock Command 386
Table 229 — TPM2_NV_WriteLock Response 386
Table 230 — TPM2_NV_GlobalWriteLock Command 389
Table 231 — TPM2_NV_GlobalWriteLock Response 389
Table 232 — TPM2_NV_Read Command 392

 Copyright © TCG 2006-2020

Table 233 — TPM2_NV_Read Response ... 392
Table 234 — TPM2_NV_ReadLock Command ... 395
Table 235 — TPM2_NV_ReadLock Response ... 395
Table 236 — TPM2_NV_ChangeAuth Command ... 398
Table 237 — TPM2_NV_ChangeAuth Response ... 398
Table 238 — TPM2_NV_Certify Command ... 401
Table 239 — TPM2_NV_Certify Response ... 401
Table 240 — TPM2_AC_GetCapability Command ... 405
Table 241 — TPM2_AC_GetCapability Response ... 405
Table 242 — TPM2_AC_Send Command ... 408
Table 243 — TPM2_AC_Send Response ... 408
Table 244 — TPM2_Policy_AC_SendSelect Command ... 411
Table 245 — TPM2_Policy_AC_SendSelect Response ... 411
Table 246 — TPM2_ACT_SetTimeout Command ... 414
Table 247 — TPM2_ACT_SetTimeout Response ... 414
Table 248 — TPM2_Vendor_TCG_Test Command ... 417
Table 249 — TPM2_Vendor_TCG_Test Response ... 417

 Copyright © TCG 2006-2020

Trusted Platform Module Library
Part 3: Commands

1 Scope

This TPM 2.0 Part 3 of the *Trusted Platform Module Library* specification contains the definitions of the TPM commands. These commands make use of the constants, flags, structures, and union definitions defined in TPM 2.0 Part 2.

The detailed description of the operation of the commands is written in the C language with extensive comments. The behavior of the C code in this TPM 2.0 Part 3 is normative but does not fully describe the behavior of a TPM. The combination of this TPM 2.0 Part 3 and TPM 2.0 Part 4 is sufficient to fully describe the required behavior of a TPM.

The code in parts 3 and 4 is written to define the behavior of a compliant TPM. In some cases (e.g., firmware update), it is not possible to provide a compliant implementation. In those cases, any implementation provided by the vendor that meets the general description of the function provided in TPM 2.0 Part 3 would be compliant.

The code in parts 3 and 4 is not written to meet any particular level of conformance nor does this specification require that a TPM meet any particular level of conformance.

2 Terms and Definitions

For the purposes of this document, the terms and definitions given in TPM 2.0 Part 1 apply.

3 Symbols and abbreviated terms

For the purposes of this document, the symbols and abbreviated terms given in TPM 2.0 Part 1 apply.

Copyright © TCG 2006-2020

4 Notation

4.1 Introduction

For the purposes of this document, the notation given in TPM 2.0 Part 1 applies.

Command and response tables use various decorations to indicate the fields of the command and the allowed types. These decorations are described in this clause.

4.2 Table Decorations

The symbols and terms in the Notation column of Table 1 are used in the tables for the command schematics. These values indicate various qualifiers for the parameters or descriptions with which they are associated.

Table 1 — Command Modifiers and Decoration

Notation	Meaning
+	A Type decoration – When appended to a value in the Type column of a command, this symbol indicates that the parameter is allowed to use the "null" value of the data type (see in TPM 2.0 Part 2, *Conditional Types*). The null value is usually TPM_RH_NULL for a handle or TPM_ALG_NULL for an algorithm selector. NOTE This decoration is not appended to response parameters.
@	A Name decoration – When this symbol precedes a handle parameter in the "Name" column, it indicates that an authorization session is required for use of the entity associated with the handle. If a handle does not have this symbol, then an authorization session is not allowed.
+PP	A Description modifier – This modifier may follow TPM_RH_PLATFORM in the "Description" column to indicate that Physical Presence is required when *platformAuth*/*platformPolicy* is provided.
+{PP}	A Description modifier – This modifier may follow TPM_RH_PLATFORM to indicate that Physical Presence may be required when *platformAuth*/*platformPolicy* is provided. The commands with this notation may be in the *setList* or *clearList* of TPM2_PP_Commands().
{NV}	A Description modifier – This modifier may follow the *commandCode* in the "Description" column to indicate that the command may result in an update of NV memory and be subject to rate throttling by the TPM. If the command code does not have this notation, then a write to NV memory does not occur as part of the command actions. NOTE Any command that uses authorization may cause a write to NV if there is an authorization failure. A TPM may use the occasion of command execution to update the NV copy of clock.
{F}	A Description modifier – This modifier indicates that the "flushed" attribute will be SET in the TPMA_CC for the command. The modifier may follow the *commandCode* in the "Description" column to indicate that any transient handle context used by the command will be flushed from the TPM when the command completes. This may be combined with the {NV} modifier but not with the {E} modifier. EXAMPLE 1 {NV F} EXAMPLE 2 TPM2_SequenceComplete() will flush the context associated with the *sequenceHandle*.
{E}	A Description modifier – This modifier indicates that the "extensive" attribute will be SET in the TPMA_CC for the command. This modifier may follow the *commandCode* in the "Description" column to indicate that the command may flush many objects and re-enumeration of the loaded context likely will be required. This may be combined with the {NV} modifier but not with the {F} modifier. EXAMPLE 1 {NV E} EXAMPLE 2 TPM2_Clear() will flush all contexts associated with the Storage hierarchy and the Endorsement hierarchy.

 Copyright © TCG 2006-2020

Notation	Meaning
Auth Index:	A Description modifier – When a handle has a “@” decoration, the “Description” column will contain an “Auth Index:” entry for the handle. This entry indicates the number of the authorization session. The authorization sessions associated with handles will occur in the session area in the order of the handles with the “@” modifier. Sessions used only for encryption/decryption or only for audit will follow the handles used for authorization.
Auth Role:	A Description modifier – This will be in the “Description” column of a handle with the “@” decoration. It may have a value of USER, ADMIN or DUP. If the handle has the Auth Role of USER and the handle is an Object, the type of authorization is determined by the setting of *userWithAuth* in the Object's attributes. If the handle is TPM_RH_OWNER, TPM_RH_ENDORSEMENT, or TPM_RH_PLATFORM, operation is as if *userWithAuth* is SET. If the handle references an NV Index, then the allowed authorizations are determined by the settings of the attributes of the NV Index as described in TPM 2.0 Part 2, "TPMA_NV (NV Index Attributes)." If the Auth Role is ADMIN and the handle is an Object, the type of authorization is determined by the setting of *adminWithPolicy* in the Object's attributes. If the handle is TPM_RH_OWNER, TPM_RH_ENDORSEMENT, or TPM_RH_PLATFORM, operation is as if *adminWithPolicy* is SET. If the handle is an NV index, operation is as if *adminWithPolicy* is SET (see 5.6 e)2)). If the DUP role is selected, authorization may only be with a policy session (DUP role only applies to Objects). When either ADMIN or DUP role is selected, a policy command that selects the command being authorized is required to be part of the policy. EXAMPLE TPM2_Certify requires the ADMIN role for the first handle (*objectHandle*). The policy authorization for objectHandle is required to contain TPM2_PolicyCommandCode(*commandCode* == TPM_CC_Certify). This sets the state of the policy so that it can be used for ADMIN role authorization in TPM2_Certify().

4.3 Handle and Parameter Demarcation

The demarcations between the header, handle, and parameter parts are indicated by:

Table 2 — Separators

Separator	Meaning
(dashed line)	the values immediately following are in the handle area
(double line)	the values immediately following are in the parameter area

4.4 AuthorizationSize and ParameterSize

Authorization sessions are not shown in the command or response schematics. When the tag of a command or response is TPM_ST_SESSIONS, then a 32-bit value will be present in the command/response buffer to indicate the size of the authorization field or the parameter field. This value shall immediately follow the handle area (which may contain no handles). For a command, this value (*authorizationSize*) indicates the size of the Authorization Area and shall have a value of 9 or more. For a response, this value (*parameterSize*) indicates the size of the parameter area and may have a value of zero.

If the *authorizationSize* field is present in the command, *parameterSize* will be present in the response, but only if the *responseCode* is TPM_RC_SUCCESS.

When authorization is required to use the TPM entity associated with a handle, then at least one session will be present. To indicate this, the command *tag* Description field contains TPM_ST_SESSIONS. Addional sessions for audit, encrypt, and decrypt may be present.

 Copyright © TCG 2006-2020

When the command *tag* Description field contains TPM_ST_NO_SESSIONS, then no sessions are allowed and the *authorizationSize* field is not present.

When a command allows use of sessions when not required, the command *tag* Description field will indicate the types of sessions that may be used with the command.

4.5 Return Code Alias

For the RC_FMT1 return codes that may add a parameter, handle, or session number, the prefix TPM_RCS_ is an alias for TPM_RC_.

TPM_RC_n is added, where n is the parameter, handle, or session number. In addition, TPM_RC_H is added for handle, TPM_RC_P for parameter, and TPM_RC_S for session errors.

NOTE TPM_RCS_ is a programming convention. Programmers should only add numbers to TPM_RCS_ return codes, never TPM_RC_ return codes. Only return codes that can have a number added have the TPM_RCS_ alias defined. Attempting to use a TPM_RCS_ return code that does not have the TPM_RCS_ alias will cause a compiler error.

EXAMPLE 1 Since TPM_RC_VALUE can have a number added, TPM_RCS_VALUE is defined. A program can use the construct "TPM_RCS_VALUE + number". Since TPM_RC_SIGNATURE cannot have a number added, TPM_RCS_SIGNATURE is not defined. A program using the construct "TPM_RCS_SIGNATURE + number" will not compile, alerting the programmer that the construct is incorrect.

By convention, the number to be added is of the form RC_CommandName_ParameterName where CommmandName is the name of the command with the TPM2_ prefix removed. The parameter name alone is insufficient because the same parameter name could be in a different position in different commands.

EXAMPLE 2 TPM2_HMAC_Start with parameters that result in TPM_ALG_NULL as the hash algorithm will returns TPM_RC_VALUE plus the parameter number. Since *hashAlg* is the second parameter, This code results:

```
#define RC_HMAC_Start_hashAlg          (TPM_RC_P + TPM_RC_2)

return TPM_RCS_VALUE + RC_HMAC_Start_hashAlg;
```

5 Command Processing

5.1 Introduction

This clause defines the command validations that are required of any implementation and the response code returned if the indicated check fails. Unless stated otherwise, the order of the checks is not normative and different TPM may give different responses when a command has multiple errors.

In the description below, some statements that describe a check may be followed by a response code in parentheses. This is the normative response code should the indicated check fail. A normative response code may also be included in the statement.

5.2 Command Header Validation

Before a TPM may begin the actions associated with a command, a set of command format and consistency checks shall be performed. These checks are listed below and should be performed in the indicated order.

a) The TPM shall successfully unmarshal a TPMI_ST_COMMAND_TAG and verify that it is either TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS (TPM_RC_BAD_TAG).

 Copyright © TCG 2006-2020

b) The TPM shall successfully unmarshal a UINT32 as the *commandSize*. If the TPM has an interface buffer that is loaded by some hardware process, the number of octets in the input buffer for the command reported by the hardware process shall exactly match the value in *commandSize* (TPM_RC_COMMAND_SIZE).

NOTE A TPM may have direct access to system memory and unmarshal directly from that memory.

c) The TPM shall successfully unmarshal a TPM_CC and verify that the command is implemented (TPM_RC_COMMAND_CODE).

5.3 Mode Checks

The following mode checks shall be performed in the order listed:

a) If the TPM is in Failure mode, then the *commandCode* is TPM_CC_GetTestResult or TPM_CC_GetCapability (TPM_RC_FAILURE) and the command *tag* is TPM_ST_NO_SESSIONS (TPM_RC_FAILURE).

NOTE 1 In Failure mode, the TPM has no cryptographic capability and processing of sessions is not supported.

b) The TPM is in Field Upgrade mode (FUM), the *commandCode* is TPM_CC_FieldUpgradeData (TPM_RC_UPGRADE).

c) If the TPM has not been initialized (TPM2_Startup()), then the *commandCode* is TPM_CC_Startup (TPM_RC_INITIALIZE).

NOTE 2 The TPM may enter Failure mode during _TPM_Init processing, before TPM2_Startup(). Since the platform firmware cannot know that the TPM is in Failure mode without accessing it, and since the first command is required to be TPM2_Startup(), the expected sequence will be that platform firmware (the CRTM) will issue TPM2_Startup() and receive TPM_RC_FAILURE indicating that the TPM is in Failure mode.

There may be failures where a TPM cannot record that it received TPM2_Startup(). In those cases, a TPM in failure mode may process TPM2_GetTestResult(), TPM2_GetCapability(), or the field upgrade commands. As a side effect, that TPM may process TPM2_GetTestResult(), TPM2_GetCapability() or the field upgrade commands before TPM2_Startup().

This is a corner case exception to the rule that TPM2_Startup() must be the first command.

The mode checks may be performed before or after the command header validation.

5.4 Handle Area Validation

After successfully unmarshaling and validating the command header, the TPM shall perform the following checks on the handles and sessions. These checks may be performed in any order.

NOTE 1 A TPM is required to perform the handle area validation before the authorization checks because an authorization cannot be performed unless the authorization values and attributes for the referenced entity are known by the TPM. For them to be known, the referenced entity must be in the TPM and accessible.

a) The TPM shall successfully unmarshal the number of handles required by the command and validate that the value of the handle is consistent with the command syntax. If not, the TPM shall return TPM_RC_VALUE.

NOTE 2 The TPM may unmarshal a handle and validate that it references an entity on the TPM before unmarshaling a subsequent handle.

NOTE 3 If the submitted command contains fewer handles than required by the syntax of the command, the TPM may continue to read into the next area and attempt to interpret the data as a handle.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

b) For all handles in the handle area of the command, the TPM will validate that the referenced entity is present in the TPM.

1) If the handle references a transient object, the handle shall reference a loaded object (TPM_RC_REFERENCE_H0 + N where N is the number of the handle in the command).

NOTE 4 If the hierarchy for a transient object is disabled, then the transient objects will be flushed so this check will fail.

2) If the handle references a persistent object, then

i) the hierarchy associated with the object (platform or storage, based on the handle value) is enabled (TPM_RC_HANDLE);

ii) the handle shall reference a persistent object that is currently in TPM non-volatile memory (TPM_RC_HANDLE);

iii) if the handle references a persistent object that is associated with the endorsement hierarchy, that the endorsement hierarchy is not disabled (TPM_RC_HANDLE); and

NOTE 5 The reference implementation keeps an internal attribute, passed down from a primary key to its descendents, indicating the object's hierarchy.

iv) if the TPM implementation moves a persistent object to RAM for command processing then sufficient RAM space is available (TPM_RC_OBJECT_MEMORY).

3) If the handle references an NV Index, then

i) an Index exists that corresponds to the handle (TPM_RC_HANDLE); and

ii) the hierarchy associated with the existing NV Index is not disabled (TPM_RC_HANDLE).

iii) If the command requires write access to the index data then TPMA_NV_WRITELOCKED is not SET (TPM_RC_NV_LOCKED)

iv) If the command requires read access to the index data then TPMA_NV_READLOCKED is not SET (TPM_RC_NV_LOCKED)

4) If the handle references a session, then the session context shall be present in TPM memory (TPM_RC_REFERENCE_H0 + N).

5) If the handle references a primary seed for a hierarchy (TPM_RH_ENDORSEMENT, TPM_RH_OWNER, or TPM_RH_PLATFORM) then the enable for the hierarchy is SET (TPM_RC_HIERARCHY).

6) If the handle references a PCR, then the value is within the range of PCR supported by the TPM (TPM_RC_VALUE)

NOTE 6 In the reference implementation, this TPM_RC_VALUE is returned by the unmarshaling code for a TPMI_DH_PCR.

5.5 Session Area Validation

a) If the tag is TPM_ST_SESSIONS and the command requires TPM_ST_NO_SESSIONS, the TPM will return TPM_RC_AUTH_CONTEXT.

b) If the tag is TPM_ST_NO_SESSIONS and the command requires TPM_ST_SESSIONS, the TPM will return TPM_RC_AUTH_MISSING.

c) If the tag is TPM_ST_SESSIONS, the TPM will attempt to unmarshal an *authorizationSize* and return TPM_RC_AUTHSIZE if the value is not within an acceptable range.

1) The minimum value is (sizeof(TPM_HANDLE) + sizeof(UINT16) + sizeof(TPMA_SESSION) + sizeof(UINT16)).

 Copyright © TCG 2006-2020

2) The maximum value of authorizationSize is equal to commandSize – (sizeof(TPM_ST) + sizeof(UINT32) + sizeof(TPM_CC) + (N * sizeof(TPM_HANDLE)) + sizeof(UINT32)) where N is the number of handles associated with the *commandCode* and may be zero.

NOTE 1 (sizeof(TPM_ST) + sizeof(UINT32) + sizeof(TPM_CC)) is the size of a command header. The last UINT32 contains the authorizationSize octets, which are not counted as being in the authorization session area.

d) The TPM will unmarshal the authorization sessions and perform the following validations:

1) If the session handle is not a handle for an HMAC session, a handle for a policy session, or, TPM_RS_PW then the TPM shall return TPM_RC_HANDLE.

2) If the session is not loaded, the TPM will return the warning TPM_RC_REFERENCE_S0 + N where N is the number of the session. The first session is session zero, N = 0.

NOTE 2 If the HMAC and policy session contexts use the same memory, the type of the context must match the type of the handle.

3) If the maximum allowed number of sessions have been unmarshaled and fewer octets than indicated in *authorizationSize* were unmarshaled (that is, *authorizationSize* is too large), the TPM shall return TPM_RC_AUTHSIZE.

4) The consistency of the authorization session attributes is checked.

i) Only one session is allowed for:

(a) session auditing (TPM_RC_ATTRIBUTES) – this session may be used for encrypt or decrypt but may not be a session that is also used for authorization;

(b) decrypting a command parameter (TPM_RC_ATTRIBUTES) – this may be any of the authorization sessions, or the audit session, or a session may be added for the single purpose of decrypting a command parameter, as long as the total number of sessions does not exceed three; and

(c) encrypting a response parameter (TPM_RC_ATTRIBUTES) – this may be any of the authorization sessions, or the audit session if present, ora session may be added for the single purpose of encrypting a response parameter, as long as the total number of sessions does not exceed three.

NOTE 3 A session used for decrypting a command parameter may also be used for encrypting a response parameter.

ii) If a session is not being used for authorization, at least one of decrypt, encrypt, or audit must be SET. (TPM_RC_ATTRIBUTES).

5) An authorization session is present for each of the handles with the “@” decoration (TPM_RC_AUTH_MISSING).

5.6 Authorization Checks

After unmarshaling and validating the handles and the consistency of the authorization sessions, the authorizations shall be checked. Authorization checks only apply to handles if the handle in the command schematic has the “@” decoration. Authorization checks must be performed in this order.

a) The public and sensitive portions of the object shall be present on the TPM (TPM_RC_AUTH_UNAVAILABLE).

b) If the associated handle is TPM_RH_PLATFORM, and the command requires confirmation with physical presence, then physical presence is asserted (TPM_RC_PP).

c) If the object or NV Index is subject to DA protection, and the authorization is with an HMAC or password, then the TPM is not in lockout (TPM_RC_LOCKOUT).

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

NOTE 1 An object is subject to DA protection if its *noDA* attribute is CLEAR. An NV Index is subject to DA protection if its TPMA_NV_NO_DA attribute is CLEAR.

NOTE 2 An HMAC or password is required in a policy session when the policy contains TPM2_PolicyAuthValue() or TPM2_PolicyPassword().

d) If the command requires a handle to have DUP role authorization, then the associated authorization session is a policy session (TPM_RC_AUTH_TYPE).

e) If the command requires a handle to have ADMIN role authorization:

1) If the entity being authorized is an object and its *adminWithPolicy* attribute is SET, or a hierarchy, then the authorization session is a policy session (TPM_RC_AUTH_TYPE).

NOTE 3 If adminWithPolicy is CLEAR, then any type of authorization session is allowed.

2) If the entity being authorized is an NV Index, then the associated authorization session is a policy session.

NOTE 4 The only commands that are currently defined that require use of ADMIN role authorization are commands that operate on objects and NV Indices.

f) If the command requires a handle to have USER role authorization:

1) If the entity being authorized is an object and its *userWithAuth* attribute is CLEAR, then the associated authorization session is a policy session (TPM_RC_POLICY_FAIL).

NOTE 5 There is no check for a hierarchy, because a hierarchy operates as if userWithAuth is SET.

2) If the entity being authorized is an NV Index;

i) if the authorization session is a policy session;

(a) the TPMA_NV_POLICYWRITE attribute of the NV Index is SET if the command modifies the NV Index data (TPM_RC_AUTH_UNAVAILABLE);

(b) the TPMA_NV_POLICYREAD attribute of the NV Index is SET if the command reads the NV Index data (TPM_RC_AUTH_UNAVAILABLE);

ii) if the authorization is an HMAC session or a password;

(a) the TPMA_NV_AUTHWRITE attribute of the NV Index is SET if the command modifies the NV Index data (TPM_RC_AUTH_UNAVAILABLE);

(b) the TPMA_NV_AUTHREAD attribute of the NV Index is SET if the command reads the NV Index data (TPM_RC_AUTH_UNAVAILABLE).

g) If the authorization is provided by a policy session, then:

1) if *policySession→timeOut* has been set, the session shall not have expired (TPM_RC_EXPIRED);

2) if *policySession→cpHash* has been set, it shall match the *cpHash* of the command (TPM_RC_POLICY_FAIL);

3) if *policySession→commandCode* has been set, then *commandCode* of the command shall match (TPM_RC_POLICY_CC);

4) *policySession→policyDigest* shall match the *authPolicy* associated with the handle (TPM_RC_POLICY_FAIL);

5) if *policySession→pcrUpdateCounter* has been set, then it shall match the value of *pcrUpdateCounter* (TPM_RC_PCR_CHANGED);

6) if *policySession→commandLocality* has been set, it shall match the locality of the command (TPM_RC_LOCALITY),

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

7) if *policySession→cpHash* contains a template, and the command is TPM2_Create(), TPM2_CreatePrimary(), or TPM2_CreateLoaded(), then the *inPublic* parmeter matches the contents of *policySession→cpHash;* and

8) if the policy requires that an authValue be provided in order to satisfy the policy, then *session.hmac* is not an Empty Buffer.

h) If the authorization uses an HMAC, then the HMAC is properly constructed using the *authValue* associated with the handle and/or the session secret (TPM_RC_AUTH_FAIL or TPM_RC_BAD_AUTH).

NOTE 6 A policy session may require proof of knowledge of the authValue of the object being authorized.

i) If the authorization uses a password, then the password matches the *authValue* associated with the handle (TPM_RC_AUTH_FAIL or TPM_RC_BAD_AUTH).

If the TPM returns an error other than TPM_RC_AUTH_FAIL then the TPM shall not alter any TPM state. If the TPM return TPM_RC_AUTH_FAIL, then the TPM shall not alter any TPM state other than *lockoutCount*.

NOTE 7 The TPM may decrease failedTries regardless of any other processing performed by the TPM. That is, the TPM may exit Lockout mode, regardless of the return code.

5.7 Parameter Decryption

If an authorization session has the TPMA_SESSION.*decrypt* attribute SET, and the command does not allow a command parameter to be encrypted, then the TPM will return TPM_RC_ATTRIBUTES. Otherwise, the TPM will decrypt the parameter using the values associated with the session before parsing parameters.

NOTE The size of the parameter to be encrypted can be zero.

5.8 Parameter Unmarshaling

5.8.1 Introduction

The detailed actions for each command assume that the input parameters of the command have been unmarshaled into a command-specific structure with the structure defined by the command schematic. Additionally, a response-specific output structure is assumed which will receive the values produced by the detailed actions.

NOTE An implementation is not required to process parameters in this manner or to separate the parameter parsing from the command actions. This method was chosen for the specification so that the normative behavior described by the detailed actions would be clear and unencumbered.

Unmarshaling is the process of processing the parameters in the input buffer and preparing the parameters for use by the command-specific action code. No data movement need take place but it is required that the TPM validate that the parameters meet the requirements of the expected data type as defined in TPM 2.0 Part 2.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

5.8.2 Unmarshaling Errors

When an error is encountered while unmarshaling a command parameter, an error response code is returned and no command processing occurs. A table defining a data type may have response codes embedded in the table to indicate the error returned when the input value does not match the parameters of the table.

NOTE In the reference implementation, a parameter number is added to the response code so that the offending parameter can be isolated. This is optional.

In many cases, the table contains no specific response code value and the return code will be determined as defined in Table 3.

Table 3 — Unmarshaling Errors

Response Code	Meaning
TPM_RC_ASYMMETRIC	a parameter that should be an asymmetric algorithm selection does not have a value that is supported by the TPM
TPM_RC_BAD_TAG	a parameter that should be a command tag selection has a value that is not supported by the TPM
TPM_RC_COMMAND_CODE	a parameter that should be a command code does not have a value that is supported by the TPM
TPM_RC_HASH	a parameter that should be a hash algorithm selection does not have a value that is supported by the TPM
TPM_RC_INSUFFICIENT	the input buffer did not contain enough octets to allow unmarshaling of the expected data type;
TPM_RC_KDF	a parameter that should be a key derivation scheme (KDF) selection does not have a value that is supported by the TPM
TPM_RC_KEY_SIZE	a parameter that is a key size has a value that is not supported by the TPM
TPM_RC_MODE	a parameter that should be a symmetric encryption mode selection does not have a value that is supported by the TPM
TPM_RC_RESERVED	a non-zero value was found in a reserved field of an attribute structure (TPMA_)
TPM_RC_SCHEME	a parameter that should be signing or encryption scheme selection does not have a value that is supported by the TPM
TPM_RC_SIZE	the value of a size parameter is larger or smaller than allowed
TPM_RC_SYMMETRIC	a parameter that should be a symmetric algorithm selection does not have a value that is supported by the TPM
TPM_RC_TAG	a parameter that should be a structure tag has a value that is not supported by the TPM
TPM_RC_TYPE	The type parameter of a TPMT_PUBLIC or TPMT_SENSITIVE has a value that is not supported by the TPM
TPM_RC_VALUE	a parameter does not have one of its allowed values

In some commands, a parameter may not be used because of various options of that command. However, the unmarshaling code is required to validate that all parameters have values that are allowed by the TPM 2.0 Part 2 definition of the parameter type even if that parameter is not used in the command actions.

 Copyright © TCG 2006-2020

5.9 Command Post Processing

When the code that implements the detailed actions of the command completes, it returns a response code. If that code is not TPM_RC_SUCCESS, the post processing code will not update any session or audit data and will return a 10-octet response packet.

If the command completes successfully, the tag of the command determines if any authorization sessions will be in the response. If so, the TPM will encrypt the first parameter of the response if indicated by the authorization attributes. The TPM will then generate a new nonce value for each session and, if appropriate, generate an HMAC.

If authorization HMAC computations are performed on the response, the HMAC keys used in the response will be the same as the HMAC keys used in processing the HMAC in the command.

NOTE 1 This primarily affects authorizations associated with a first write to an NV Index using a bound session. The computation of the HMAC in the response is performed as if the Name of the Index did not change as a consequence of the command actions. The session binding to the NV Index will not persist to any subsequent command.

NOTE 2 The authorization attributes were validated during the session area validation to ensure that only one session was used for parameter encryption of the response and that the command allowed encryption in the response.

NOTE 3 No session nonce value is used for a password authorization but the session data is present.

Additionally, if the command is being audited by Command Audit, the audit digest is updated with the *cpHash* of the command and *rpHash* of the response.

 Copyright © TCG 2006-2020

6 Response Values

6.1 Tag

When a command completes successfully, the *tag* parameter in the response shall have the same value as the *tag* parameter in the command (TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS). When a command fails (the responseCode is not TPM_RC_SUCCESS), then the *tag* parameter in the response shall be TPM_ST_NO_SESSIONS.

A special case exists when the command *tag* parameter is not an allowed value (TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS). For this case, it is assumed that the system software is attempting to send a command formatted for a TPM 1.2 but the TPM is not capable of executing TPM 1.2 commands. So that the TPM 1.2 compatible software will have a recognizable response, the TPM sets *tag* to TPM_ST_RSP_COMMAND, *responseSize* to 00 00 00 $0A_{16}$ and *responseCode* to TPM_RC_BAD_TAG. This is the same response as the TPM 1.2 fatal error for TPM_BADTAG.

6.2 Response Codes

The normal response for any command is TPM_RC_SUCCESS. Any other value indicates that the command did not complete and the state of the TPM is unchanged. An exception to this general rule is that the logic associated with dictionary attack protection is allowed to be modified when an authorization failure occurs.

Commands have response codes that are specific to that command, and those response codes are enumerated in the detailed actions of each command. The codes associated with the unmarshaling of parameters are documented Table 3. Another set of response code values are not command specific and indicate a problem that is not specific to the command. That is, if the indicated problem is remedied, the same command could be resubmitted and may complete normally.

The response codes that are not command specific are listed and described in

Table 4.

The reference code for the command actions may have code that generates specific response codes associated with a specific check but the listing of responses may not have that response code listed.

 Copyright © TCG 2006-2020

Table 4 — Command-Independent Response Codes

Response Code	Meaning
TPM_RC_CANCELED	This response code may be returned by a TPM that supports command cancel. When the TPM receives an indication that the current command should be cancelled, the TPM may complete the command or return this code. If this code is returned, then the TPM state is not changed and the same command may be retried.
TPM_RC_CONTEXT_GAP	This response code can be returned for commands that manage session contexts. It indicates that the gap between the lowest numbered active session and the highest numbered session is at the limits of the session tracking logic. The remedy is to load the session context with the lowest number so that its tracking number can be updated.
TPM_RC_LOCKOUT	This response indicates that authorizations for objects subject to DA protection are not allowed at this time because the TPM is in DA lockout mode. The remedy is to wait or to exeucte TPM2_DictionaryAttackLockoutReset().
TPM_RC_MEMORY	A TPM may use a common pool of memory for objects, sessions, and other purposes. When the TPM does not have enough memory available to perform the actions of the command, it may return TPM_RC_MEMORY. This indicates that the TPM resource manager may flush either sessions or objects in order to make memory available for the command execution. A TPM may choose to return TPM_RC_OBJECT_MEMORY or TPM_RC_SESSION_MEMORY if it needs contexts of a particular type to be flushed.
TPM_RC_NV_RATE	This response code indicates that the TPM is rate-limiting writes to the NV memory in order to prevent wearout. This response is possible for any command that explicity writes to NV or commands that incidentally use NV such as a command that uses authorization session that may need to update the dictionary attack logic.
TPM_RC_NV_UNAVAILABLE	This response code is similar to TPM_RC_NV_RATE but indicates that access to NV memory is currently not available and the command is not allowed to proceed until it is. This would occur in a system where the NV memory used by the TPM is not exclusive to the TPM and is a shared system resource.
TPM_RC_OBJECT_HANDLES	This response code indicates that the TPM has exhausted its handle space and no new objects can be loaded unless the TPM is rebooted. This does not occur in the reference implementation because of the way that object handles are allocated. However, other implementations are allowed to assign each object a unique handle each time the object is loaded. A TPM using this implementation would be able to load 2^{24} objects before the object space is exhausted.
TPM_RC_OBJECT_MEMORY	This response code can be returned by any command that causes the TPM to need an object 'slot'. The most common case where this might be returned is when an object is loaded (TPM2_Load, TPM2_CreatePrimary(), or TPM2_ContextLoad()). However, the TPM implementation is allowed to use object slots for other reasons. In the reference implementation, the TPM copies a referenced persistent object into RAM for the duration of the commannd. If all the slots are previously occupied, the TPM may return this value. A TPM is allowed to use object slots for other purposes and return this value. The remedy when this response is returned is for the TPM resource manager to flush a transient object.
TPM_RC_REFERENCE_Hx	This response code indicates that a handle in the handle area of the command is not associated with a loaded object. The value of 'x' is in the range 0 to 6 with a value of 0 indicating the 1st handle and 6 representing the 7th. Upper values are provided for future use. The TPM resource manager needs to find the correct object and load it. It may then adjust the handle and retry the command. NOTE Usually, this error indicates that the TPM resource manager has a corrupted database.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

Response Code	Meaning
TPM_RC_REFERENCE_Sx	This response code indicates that a handle in the session area of the command is not associated with a loaded session. The value of 'x' is in the range 0 to 6 with a value of 0 indicating the 1st session handle and 6 representing the 7th. Upper values are provided for future use. The TPM resource manager needs to find the correct session and load it. It may then retry the command. NOTE Usually, this error indicates that the TPM resource manager has a corrupted database.
TPM_RC_RETRY	the TPM was not able to start the command
TPM_RC_SESSION_HANDLES	This response code indicates that the TPM does not have a handle to assign to a new session. This respose is only returned by TPM2_StartAuthSession(). It is listed here because the command is not in error and the TPM resource manager can remedy the situation by flushing a session (TPM2_FlushContext().
TPM_RC_SESSION_MEMORY	This response code can be returned by any command that causes the TPM to need a session 'slot'. The most common case where this might be returned is when a session is loaded (TPM2_StartAuthSession() or TPM2_ContextLoad()). However, the TPM implementation is allowed to use object slots for other purposes. The remedy when this response is returned is for the TPM resource manager to flush a transient object.
TPM_RC_SUCCESS	Normal completion for any command. If the responseCode is TPM_RC_SUCCESS, then the rest of the response has the format indicated in the response schematic. Otherwise, the response is a 10 octet value indicating an error.
TPM_RC_TESTING	This response code indicates that the TPM is performing tests and cannot respond to the request at this time. The command may be retried.
TPM_RC_YIELDED	the TPM has suspended operation on the command; forward progress was made and the command may be retried. See TPM 2.0 Part 1, “Multi-tasking.” NOTE This cannot occur on the reference implementation.

7 Implementation Dependent

The actions code for each command makes assumptions about the behavior of various sub-systems. There are many possible implementations of the subsystems that would achieve equivalent results. The actions code is not written to anticipate all possible implementations of the sub-systems. Therefore, it is the responsibility of the implementer to ensure that the necessary changes are made to the actions code when the sub-system behavior changes.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

8 Detailed Actions Assumptions

8.1 Introduction

The C code in the Detailed Actions for each command is written with a set of assumptions about the processing performed before the action code is called and the processing that will be done after the action code completes.

8.2 Pre-processing

Before calling the command actions code, the following actions have occurred.

- Verification that the handles in the handle area reference entities that are resident on the TPM.
- NOTE If a handle is in the parameter portion of the command, the associated entity does not have to be loaded, but the handle is required to be the correct type.
- If use of a handle requires authorization, the Password, HMAC, or Policy session associated with the handle has been verified.
- If a command parameter was encrypted using parameter encryption, it was decrypted before being unmarshaled.
- If the command uses handles or parameters, the calling stack contains a pointer to a data structure (*in*) that holds the unmarshaled values for the handles and command parameters. If the response has handles or parameters, the calling stack contains a pointer to a data structure (*out*) to hold the handles and response parameters generated by the command.
- All parameters of the *in* structure have been validated and meet the requirements of the parameter type as defined in TPM 2.0 Part 2.
- Space set aside for the out structure is sufficient to hold the largest *out* structure that could be produced by the command

8.3 Post Processing

When the function implementing the command actions completes,

- response parameters that require parameter encryption will be encrypted after the command actions complete;
- audit and session contexts will be updated if the command response is TPM_RC_SUCCESS; and
- the command header and command response parameters will be marshaled to the response buffer.

 Copyright © TCG 2006-2020

9 Start-up

9.1 Introduction

This clause contains the commands used to manage the startup and restart state of a TPM.

9.2 _TPM_Init

9.2.1 General Description

_TPM_Init initializes a TPM.

Initialization actions include testing code required to execute the next expected command. If the TPM is in FUM, the next expected command is TPM2_FieldUpgradeData(); otherwise, the next expected command is TPM2_Startup().

NOTE 1 If the TPM performs self-tests after receiving _TPM_Init() and the TPM enters Failure mode before receiving TPM2_Startup() or TPM2_FieldUpgradeData(), then the TPM may be able to accept TPM2_GetTestResult() or TPM2_GetCapability().

The means of signaling _TPM_Init shall be defined in the platform-specific specifications that define the physical interface to the TPM. The platform shall send this indication whenever the platform starts its boot process and only when the platform starts its boot process.

There shall be no software method of generating this indication that does not also reset the platform and begin execution of the CRTM.

NOTE 2 In the reference implementation, this signal causes an internal flag (*s_initialized*) to be CLEAR. While this flag is CLEAR, the TPM will only accept the next expected command described above.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

9.2.2 Detailed Actions

[[_TPM_Init]]

9.3 TPM2_Startup

9.3.1 General Description

TPM2_Startup() is always preceded by _TPM_Init, which is the physical indication that TPM initialization is necessary because of a system-wide reset. TPM2_Startup() is only valid after _TPM_Init. Additional TPM2_Startup() commands are not allowed after it has completed successfully. If a TPM requires TPM2_Startup() and another command is received, or if the TPM receives TPM2_Startup() when it is not required, the TPM shall return TPM_RC_INITIALIZE.

NOTE 1 See 9.2.1 for other command options for a TPM supporting field upgrade mode.

NOTE 2 _TPM_Hash_Start, _TPM_Hash_Data, and _TPM_Hash_End are not commands and a platform-specific specification may allow these indications between _TPM_Init and TPM2_Startup().

If in Failure mode, the TPM shall accept TPM2_GetTestResult() and TPM2_GetCapability() even if TPM2_Startup() is not completed successfully or processed at all.

A platform-specific specification may restrict the localities at which TPM2_Startup() may be received.

A Shutdown/Startup sequence determines the way in which the TPM will operate in response to TPM2_Startup(). The three sequences are:

1) TPM Reset – This is a Startup(CLEAR) preceded by either Shutdown(CLEAR) or no TPM2_Shutdown(). On TPM Reset, all variables go back to their default initialization state.

   NOTE 3 Only those values that are specified as having a default initialization state are changed by TPM Reset. Persistent values that have no default initialization state are not changed by this command. Values such as seeds have no default initialization state and only change due to specific commands.

2) TPM Restart – This is a Startup(CLEAR) preceded by Shutdown(STATE). This preserves much of the previous state of the TPM except that PCR and the controls associated with the Platform hierarchy are all returned to their default initialization state;

3) TPM Resume – This is a Startup(STATE) preceded by Shutdown(STATE). This preserves the previous state of the TPM including the static Root of Trust for Measurement (S-RTM) PCR and the platform controls other than the *phEnable*.

If a TPM receives Startup(STATE) and that was not preceded by Shutdown(STATE), the TPM shall return TPM_RC_VALUE.

If, during TPM Restart or TPM Resume, the TPM fails to restore the state saved at the last Shutdown(STATE), the TPM shall enter Failure Mode and return TPM_RC_FAILURE.

On any TPM2_Startup(),

- *phEnable* shall be SET;
- all transient contexts (objects, sessions, and sequences) shall be flushed from TPM memory;

NOTE 4 See Part 1 Time for a description of the TPMS_TIME_INFO.*time* behaviour.

- use of *lockoutAuth* shall be enabled if *lockoutRecovery* is zero.

Additional actions are performed based on the Shutdown/Startup sequence.

 Copyright © TCG 2006-2020

On TPM Reset:

- *platformAuth* and *platformPolicy* shall be set to the Empty Buffer,
- For each NV Index with TPMA_NV_WRITEDEFINE CLEAR or TPMA_NV_WRITTEN CLEAR, TPMA_NV_WRITELOCKED shall be CLEAR,
- For each NV Index with TPMA_NV_ORDERLY SET, TPMA_NV_WRITTEN shall be CLEAR unless the type is TPM_NT_COUNTER,
- On a disorderly reset, advance the orderly counters,
- For each NV Index with TPMA_NV_CLEAR_STCLEAR SET, TPMA_NV_WRITTEN shall be CLEAR,
- tracking data for saved session contexts shall be set to its initial value,
- the object context sequence number is reset to zero,
- a new context encryption key shall be generated,
- TPMS_CLOCK_INFO.*restartCount* shall be reset to zero,
- TPMS_CLOCK_INFO.*resetCount* shall be incremented,
- the PCR Update Counter shall be clear to zero,

  NOTE 5 Because the PCR update counter may be incremented when a PCR is reset, the PCR resets performed as part of this command can result in the PCR update counter being non-zero at the end of this command.

- *phEnableNV, shEnable* and *ehEnable* shall be SET, and
- PCR in all banks are reset to their default initial conditions as determined by the relevant platform-specific specification and the H-CRTM state (for exceptions, see TPM 2.0 Part 1*, H-CRTM before TPM2_Startup() and TPM2_Startup without H-CRTM*),
- For each ACT the timeout is reset to zero, the *signaled* attribute is set to CLEAR (if *preserveSignaled* is CLEAR), and the *authPolicy* is set to the Empty Buffer and its hashAlg is set to TPM_ALG_NULL.

  NOTE 6 PCR may be initialized any time between _TPM_Init and the end of TPM2_Startup(). PCR that are preserved by TPM Resume will need to be restored during TPM2_Startup().

  NOTE 7 See "Initializing PCR" in TPM 2.0 Part 1 for a description of the default initial conditions for a PCR.

 Copyright © TCG 2006-2020

On TPM Restart:

- TPMS_CLOCK_INFO.*restartCount* shall be incremented,
- *phEnableNV, shEnable* and *ehEnable* shall be SET,
- *platformAuth* and *platformPolicy* shall be set to the Empty Buffer,
- For each NV index with TPMA_NV_WRITEDEFINE CLEAR or TPMA_NV_WRITTEN CLEAR, TPMA_NV_WRITELOCKED shall be CLEAR,
- For each NV index with TPMA_NV_CLEAR_STCLEAR SET, TPMA_NV_WRITTEN shall be CLEAR, and
- PCR in all banks are reset to their default initial conditions.
- If an H-CRTM Event Sequence is active, extend the PCR designated by the platform-specific specification.
- For each ACT the timeout is reset to zero, the *signaled* attribute is set to CLEAR (if *preserveSignaled* is CLEAR), and the *authPolicy* is set to the Empty Buffer and its hashAlg is set to TPM_ALG_NULL.

On TPM Resume:

- the H-CRTM startup method is the same for this TPM2_Startup() as for the previous TPM2_Startup(); (TPM_RC_LOCALITY)
- TPMS_CLOCK_INFO.*restartCount* shall be incremented; and
- PCR that are specified in a platform-specific specification to be preserved on TPM Resume are restored to their saved state and other PCR are set to their initial value as determined by a platform-specific specification. For constraints, see TPM 2.0 Part 1, *H-CRTM before TPM2_Startup() and TPM2_Startup without H-CRTM.*
- The ACT timeout, the ACT *signaled* attribute and the ACT specific *authPolicy* values are preserved.

Other TPM state may change as required to meet the needs of the implementation.

If the *startupType* is TPM_SU_STATE and the TPM requires TPM_SU_CLEAR, then the TPM shall return TPM_RC_VALUE.

NOTE 8 The TPM will require TPM_SU_CLEAR when no shutdown was performed or after Shutdown(CLEAR).

NOTE 9 If *startupType* is neither TPM_SU_STATE nor TPM_SU_CLEAR, then the unmarshaling code returns TPM_RC_VALUE.

 Copyright © TCG 2006-2020

9.3.2 Command and Response

Table 5 — TPM2_Startup Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Startup {NV}
TPM_SU	startupType	TPM_SU_CLEAR or TPM_SU_STATE

Table 6 — TPM2_Startup Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

9.3.3 Detailed Actions

[[Startup]]

 Copyright © TCG 2006-2020

9.4 TPM2_Shutdown

9.4.1 General Description

This command is used to prepare the TPM for a power cycle. The *shutdownType* parameter indicates how the subsequent TPM2_Startup() will be processed.

For a *shutdownType* of any type, the volatile portion of Clock is saved to NV memory and the orderly shutdown indication is SET. NV Indexes with the TPMA_NV_ORDERLY attribute will be updated.

For a *shutdownType* of TPM_SU_STATE, the following additional items are saved:

- tracking information for saved session contexts;
- the session context counter;
- PCR that are designated as being preserved by TPM2_Shutdown(TPM_SU_STATE);
- the PCR Update Counter;
- flags associated with supporting the TPMA_NV_WRITESTCLEAR and TPMA_NV_READSTCLEAR attributes;
- the counter value and authPolicy for each ACT; and

  NOTE If a counter has not been updated since the last TPM2_Startup(), then the saved value will be one half of the current counter value.

- the command audit digest and count.

The following items shall not be saved and will not be in TPM memory after the next TPM2_Startup:

- TPM-memory-resident session contexts;
- TPM-memory-resident transient objects; or
- TPM-memory-resident hash contexts created by TPM2_HashSequenceStart().

Some values may be either derived from other values or saved to NV memory.

This command saves TPM state but does not change the state other than the internal indication that the context has been saved. The TPM shall continue to accept commands. If a subsequent command changes TPM state saved by this command, then the effect of this command is nullified. The TPM MAY nullify this command for any subsequent command rather than check whether the command changed state saved by this command. If this command is nullified. and if no TPM2_Shutdown() occurs before the next TPM2_Startup(), then the next TPM2_Startup() shall be TPM2_Startup(CLEAR).

 Copyright © TCG 2006-2020

9.4.2 Command and Response

Table 7 — TPM2_Shutdown Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Shutdown {NV}
TPM_SU	shutdownType	TPM_SU_CLEAR or TPM_SU_STATE

Table 8 — TPM2_Shutdown Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

9.4.3 Detailed Actions

[[Shutdown]]

10 Testing

10.1 Introduction

Compliance to standards for hardware security modules may require that the TPM test its functions before the results that depend on those functions may be returned. The TPM may perform operations using testable functions before those functions have been tested as long as the TPM returns no value that depends on the correctness of the testable function.

EXAMPLE TPM2_PCR_Extend() may be executed before the hash algorithms have been tested. However, until the hash algorithms have been tested, the contents of a PCR may not be used in any command if that command may result in a value being returned to the TPM user. This means that TPM2_PCR_Read() or TPM2_PolicyPCR() could not complete until the hashes have been checked but other TPM2_PCR_Extend() commands may be executed even though the operation uses previous PCR values.

If a command is received that requires return of a value that depends on untested functions, the TPM shall test the required functions before completing the command.

Once the TPM has received TPM2_SelfTest() and before completion of all tests, the TPM is required to return TPM_RC_TESTING for any command that uses a function that requires a test.

If a self-test fails at any time, the TPM will enter Failure mode. While in Failure mode, the TPM will return TPM_RC_FAILURE for any command other than TPM2_GetTestResult() and TPM2_GetCapability(). The TPM will remain in Failure mode until the next _TPM_Init.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

10.2 TPM2_SelfTest

10.2.1 General Description

This command causes the TPM to perform a test of its capabilities. If the *fullTest* is YES, the TPM will test all functions. If *fullTest* = NO, the TPM will only test those functions that have not previously been tested.

If any tests are required, the TPM shall either

- return TPM_RC_TESTING and begin self-test of the required functions, or

  NOTE 1 If *fullTest* is NO, and all functions have been tested, the TPM shall return TPM_RC_SUCCESS.

- perform the tests and return the test result when complete. On failure, the TPM shall return TPM_RC_FAILURE.

If the TPM uses option a), the TPM shall return TPM_RC_TESTING for any command that requires use of a testable function, even if the functions required for completion of the command have already been tested.

NOTE 2 This command may cause the TPM to continue processing after it has returned the response. So that software can be notified of the completion of the testing, the interface may include controls that would allow the TPM to generate an interrupt when the "background" processing is complete. This would be in addition to the interrupt that may be available for signaling normal command completion. It is not necessary that there be two interrupts, but the interface should provide a way to indicate the nature of the interrupt (normal command or deferred command).

NOTE 3 The PC Client platform specific TPM, in response to *fullTest* YES, will not return TPM_RC_TESTING. It will block until all tests are complete.

 Copyright © TCG 2006-2020

10.2.2 Command and Response

Table 9 — TPM2_SelfTest Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_SelfTest {NV}
TPMI_YES_NO	fullTest	YES if full test to be performed NO if only test of untested functions required

Table 10 — TPM2_SelfTest Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

10.2.3 Detailed Actions

[[SelfTest]]

10.3 TPM2_IncrementalSelfTest

10.3.1 General Description

This command causes the TPM to perform a test of the selected algorithms.

NOTE 1 The *toTest* list indicates the algorithms that software would like the TPM to test in anticipation of future use. This allows tests to be done so that a future commands will not be delayed due to testing.

The implementation may treat algorithms on the *toTest* list as either 'test each completely' or 'test this combination.'

EXAMPLE If the *toTest* list includes AES and CTR mode, it may be interpreted as a request to test only AES in CTR mode. Alternatively, it may be interpreted as a request to test AES in all modes and CTR mode for all symmetric algorithms.

If *toTest* contains an algorithm that has already been tested, it will not be tested again.

NOTE 2 The only way to force retesting of an algorithm is with TPM2_SelfTest(*fullTest* = YES).

The TPM will return in *toDoList* a list of algorithms that are yet to be tested. This list is not the list of algorithms that are scheduled to be tested but the algorithms/functions that have not been tested. Only the algorithms on the *toTest* list are scheduled to be tested by this command.

NOTE 3 An algorithm remains on the *toDoList* while any part of it remains untested.

EXAMPLE A symmetric algorithm remains untested until it is tested with all its modes.

Making *toTest* an empty list allows the determination of the algorithms that remain untested without triggering any testing.

If *toTest* is not an empty list, the TPM shall return TPM_RC_SUCCESS for this command and then return TPM_RC_TESTING for any subsequent command (including TPM2_IncrementalSelfTest()) until the requested testing is complete.

NOTE 4 If *toDoList* is empty, then no additional tests are required and TPM_RC_TESTING will not be returned in subsequent commands and no additional delay will occur in a command due to testing.

NOTE 5 If none of the algorithms listed in *toTest* is in the *toDoList*, then no tests will be performed.

NOTE 6 The TPM cannot return TPM_RC_TESTING for the first call to this command even when testing is not complete, because response parameters can only returned with the TPM_RC_SUCCESS return code.

If all the parameters in this command are valid, the TPM returns TPM_RC_SUCCESS and the *toDoList* (which may be empty).

NOTE 7 An implementation may perform all requested tests before returning TPM_RC_SUCCESS, or it may return TPM_RC_SUCCESS for this command and then return TPM_RC_TESTING for all subsequence commands (including TPM2_IncrementatSelfTest()) until the requested tests are complete.

 Copyright © TCG 2006-2020

10.3.2 Command and Response

Table 11 — TPM2_IncrementalSelfTest Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_IncrementalSelfTest {NV}
TPML_ALG	toTest	list of algorithms that should be tested

Table 12 — TPM2_IncrementalSelfTest Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPML_ALG	toDoList	list of algorithms that need testing

 Copyright © TCG 2006-2020

10.3.3 Detailed Actions

[[IncrementalSelfTest]]

10.4 TPM2_GetTestResult

10.4.1 General Description

This command returns manufacturer-specific information regarding the results of a self-test and an indication of the test status.

If TPM2_SelfTest() has not been executed and a testable function has not been tested, *testResult* will be TPM_RC_NEEDS_TEST. If TPM2_SelfTest() has been received and the tests are not complete, *testResult* will be TPM_RC_TESTING.

If testing of all functions is complete without functional failures, *testResult* will be TPM_RC_SUCCESS. If any test failed, *testResult* will be TPM_RC_FAILURE.

This command will operate when the TPM is in Failure mode so that software can determine the test status of the TPM and so that diagnostic information can be obtained for use in failure analysis. If the TPM is in Failure mode, then *tag* is required to be TPM_ST_NO_SESSIONS or the TPM shall return TPM_RC_FAILURE.

NOTE The reference implementation may return a 32-bit value *s_failFunction*. This simply gives a unique value to each of the possible places where a failure could occur. It is not intended to provide a pointer to the function. __func__ is a pointer to a character string but the failure mode code can only return 32-bit values. It is expected that the manufacturer can disambiguate this value if a customer's TPM goes into failure mode.

 Copyright © TCG 2006-2020

10.4.2 Command and Response

Table 13 — TPM2_GetTestResult Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_GetTestResult

Table 14 — TPM2_GetTestResult Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_MAX_BUFFER	outData	test result data contains manufacturer-specific information
TPM_RC	testResult	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

10.4.3 Detailed Actions

[[GetTestResult]]

11 Session Commands

11.1 TPM2_StartAuthSession

11.1.1 General Description

This command is used to start an authorization session using alternative methods of establishing the session key (*sessionKey*). The session key is then used to derive values used for authorization and for encrypting parameters.

This command allows injection of a secret into the TPM using either asymmetric or symmetric encryption. The type of *tpmKey* determines how the value in *encryptedSalt* is encrypted. The decrypted secret value is used to compute the *sessionKey*.

NOTE 1 If *tpmKey* Is TPM_RH_NULL, then *encryptedSalt* is required to be an Empty Buffer.

The label value of “SECRET” (see “Terms and Definitions” in TPM 2.0 Part 1) is used in the recovery of the secret value.

The TPM generates the *sessionKey* from the recovered secret value.

No authorization is required for *tpmKey* or *bind*.

NOTE 2 The justification for using *tpmKey* without providing authorization is that the result of using the key is not available to the caller, except indirectly through the *sessionKey*. This does not represent a point of attack on the value of the key. If the caller attempts to use the session without knowing the *sessionKey* value, it is an authorization failure that will trigger the dictionary attack logic.

The entity referenced with the *bind* parameter contributes an authorization value to the *sessionKey* generation process.

If both *tpmKey* and *bind* are TPM_RH_NULL, then *sessionKey* is set to the Empty Buffer. If *tpmKey* is not TPM_RH_NULL, then *encryptedSalt* is used in the computation of *sessionKey*. If *bind* is not TPM_RH_NULL, the *authValue* of *bind* is used in the *sessionKey* computation.

If *symmetric* specifies a block cipher, then TPM_ALG_CFB is the only allowed value for the *mode* field in the *symmetric* parameter (TPM_RC_MODE).

This command starts an authorization session and returns the session handle along with an initial *nonceTPM* in the response.

If the TPM does not have a free slot for an authorization session, it shall return TPM_RC_SESSION_HANDLES.

If the TPM implements a “gap” scheme for assigning *contextID* values, then the TPM shall return TPM_RC_CONTEXT_GAP if creating the session would prevent recycling of old saved contexts (See “Context Management” in TPM 2.0 Part 1).

If *tpmKey* is not TPM_ALG_NULL then *encryptedSalt* shall be a TPM2B_ENCRYPTED_SECRET of the proper type for *tpmKey*. The TPM shall return TPM_RC_HANDLE if the sensitive portion of *tpmKey* is not loaded. The TPM shall return TPM_RC_VALUE if:

a) *tpmKey* references an RSA key and
   1) the size of *encryptedSalt* is not the same as the size of the public modulus of *tpmKey*,
   2) *encryptedSalt* has a value that is greater than the public modulus of *tpmKey*,
   3) *encryptedSalt* is not a properly encoded OAEP value, or
   4) the decrypted *salt* value is larger than the size of the digest produced by the *nameAlg* of *tpmKey*; or

 Copyright © TCG 2006-2020

NOTE 3 The asymScheme of the key object is ignored in this case and TPM_ALG_OAEP is used, even if asymScheme is set to TPM_ALG_NULL.

b) *tpmKey* references an ECC key and *encryptedSalt*

1) does not contain a TPMS_ECC_POINT or

2) is not a point on the curve of *tpmKey*;

NOTE 4 When ECC is used, the point multiply process produces a value (Z) that is used in a KDF to produce the final secret value. The size of the secret value is an input parameter to the KDF and the result will be set to be the size of the digest produced by the *nameAlg* of *tpmKey*.

The TPM shall return TPM_RC_KEY if *tpmkey* does not reference an asymmetric key. The TPM shall return TPM_RC_VALUE if the scheme of the key is not TPM_ALG_OAEP or TPM_ALG_NULL. The TPM shall return TPM_RC_ATTRIBUTES if tpmKey does not have the *decrypt* attribute SET.

NOTE While TPM_RC_VALUE is preferred, TPM_RC_SCHEME is acceptable.

If *bind* references a transient object, then the TPM shall return TPM_RC_HANDLE if the sensitive portion of the object is not loaded.

For all session types, this command will cause initialization of the *sessionKey* and may establish binding between the session and an object (the *bind* object). If *sessionType* is TPM_SE_POLICY or TPM_SE_TRIAL, the additional session initialization is:

- set *policySession→policyDigest* to a Zero Digest (the digest size for *policySession→policyDigest* is the size of the digest produced by *authHash*);
- authorization may be given at any locality;
- authorization may apply to any command code;
- authorization may apply to any command parameters or handles;
- the authorization has no time limit;
- an authValue is not needed when the authorization is used;
- the session is not bound;
- the session is not an audit session; and
- the time at which the policy session was created is recorded.

Additionally, if *sessionType* is TPM_SE_TRIAL, the session will not be usable for authorization but can be used to compute the *authPolicy* for an object.

NOTE 5 Although this command changes the session allocation information in the TPM, it does not invalidate a saved context. That is, TPM2_Shutdown() is not required after this command in order to re-establish the orderly state of the TPM. This is because the created context will occupy an available slot in the TPM and sessions in the TPM do not survive any TPM2_Startup(). However, if a created session is context saved, the orderly state does change.

The TPM shall return TPM_RC_SIZE if *nonceCaller* is less than 16 octets or is greater than the size of the digest produced by *authHash*.

 Copyright © TCG 2006-2020

11.1.2 Command and Response

Table 15 — TPM2_StartAuthSession Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit, decrypt, or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_StartAuthSession
TPMI_DH_OBJECT+	tpmKey	handle of a loaded decrypt key used to encrypt *salt* may be TPM_RH_NULL Auth Index: None
TPMI_DH_ENTITY+	bind	entity providing the *authValue* may be TPM_RH_NULL Auth Index: None
TPM2B_NONCE	nonceCaller	initial *nonceCaller,* sets nonceTPM size for the session shall be at least 16 octets
TPM2B_ENCRYPTED_SECRET	encryptedSalt	value encrypted according to the type of *tpmKey* If *tpmKey* is TPM_RH_NULL, this shall be the Empty Buffer.
TPM_SE	sessionType	indicates the type of the session; simple HMAC or policy (including a trial policy)
TPMT_SYM_DEF+	symmetric	the algorithm and key size for parameter encryption may select TPM_ALG_NULL
TPMI_ALG_HASH	authHash	hash algorithm to use for the session Shall be a hash algorithm supported by the TPM and not TPM_ALG_NULL

Table 16 — TPM2_StartAuthSession Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_SH_AUTH_SESSION	sessionHandle	handle for the newly created session
TPM2B_NONCE	nonceTPM	the initial nonce from the TPM, used in the computation of the *sessionKey*

 Copyright © TCG 2006-2020

11.1.3 Detailed Actions

[[StartAuthSession]]

11.2 TPM2_PolicyRestart

11.2.1 General Description

This command allows a policy authorization session to be returned to its initial state. This command is used after the TPM returns TPM_RC_PCR_CHANGED. That response code indicates that a policy will fail because the PCR have changed after TPM2_PolicyPCR() was executed. Restarting the session allows the authorizations to be replayed because the session restarts with the same *nonceTPM*. If the PCR are valid for the policy, the policy may then succeed.

This command does not reset the policy ID or the policy start time.

 Copyright © TCG 2006-2020

11.2.2 Command and Response

Table 17 — TPM2_PolicyRestart Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyRestart
TPMI_SH_POLICY	sessionHandle	the handle for the policy session

Table 18 — TPM2_PolicyRestart Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

11.2.3 Detailed Actions

[[PolicyRestart]]

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

12 Object Commands

12.1 TPM2_Create

12.1.1 General Description

This command is used to create an object that can be loaded into a TPM using TPM2_Load(). If the command completes successfully, the TPM will create the new object and return the object's creation data (*creationData*), its public area (*outPublic*), and its encrypted sensitive area (*outPrivate*). Preservation of the returned data is the responsibility of the caller. The object will need to be loaded (TPM2_Load()) before it may be used. The only difference between the *inPublic* TPMT_PUBLIC template and the *outPublic* TPMT_PUBLIC object is in the *unique* field.

NOTE 1 This command may require temporary use of a transient resource, even though the object does not remain loaded after the command. See Part 1 Transient Resources.

TPM2B_PUBLIC template (*inPublic*) contains all of the fields necessary to define the properties of the new object. The setting for these fields is defined in "Public Area Template" in Part 1 of this specification and in "TPMA_OBJECT" in Part 2 of this specification. The size of the *unique* field shall not be checked for consistency with the other object parameters.

NOTE 2 For interoperability, the *unique* field should not be set to a value that is larger than allowed by object parameters, so that the unmarshaling will not fail. A size of zero is recommended. After unmarshaling, the TPM does not use the input *unique* field. It is, however, used in TPM2_CreatePrimary() and TPM2_CreateLoaded.

EXAMPLE 1 A TPM_ALG_RSA object with a *keyBits* of 2048 in the object's parameters should have a *unique* field that is no larger than 256 bytes.

EXAMPLE 2 TPM_ALG_KEYEDHASH or a TPM_ALG_SYMCIPHER object should have a *unique* field that is no larger than the digest produced by the object's *nameAlg*.

The *parentHandle* parameter shall reference a loaded decryption key that has both the public and sensitive area loaded.

When defining the object, the caller provides a template structure for the object in a TPM2B_PUBLIC structure (*inPublic*), an initial value for the object's *authValue (inSensitive.userAuth),* and, if the object is a symmetric object, an optional initial data value (*inSensitive.data*). The TPM shall validate the consistency of the attributes of *inPublic* according to the Creation rules in "TPMA_OBJECT" in TPM 2.0 Part 2.

The *inSensitive* parameter may be encrypted using parameter encryption.

The methods in this clause are used by both TPM2_Create() and TPM2_CreatePrimary(). When a value is indicated as being TPM-generated, the value is filled in by bits from the RNG if the command is TPM2_Create() and with values from **KDFa**() if the command is TPM2_CreatePrimary(). The parameters of each creation value are specified in TPM 2.0 Part 1.

The *sensitiveDataOrigin* attribute of *inPublic* shall be SET if *inSensitive.data* is an Empty Buffer and CLEAR if *inSensitive.data* is not an Empty Buffer or the TPM shall return TPM_RC_ATTRIBUTES.

If the Object is a not a *keyedHash* object, and the *sign* and *encrypt* attributes are CLEAR, the TPM shall return TPM_RC_ATTRIBUTES.

The TPM will create new data for the sensitive area and compute a TPMT_PUBLIC.*unique* from the sensitive area based on the object type:

a) For a symmetric key:

 Copyright © TCG 2006-2020

1) If *inSensitive*.sensitive.*data* is the Empty Buffer, a TPM-generated key value is placed in the new object's TPMT_SENSITIVE.*sensitive.sym*. The size of the key will be determined by *inPublic.publicArea.parameters*.

2) If *inSensitive*.sensitive.*data* is not the Empty Buffer, the TPM will validate that the size of *inSensitive.data* is no larger than the key size indicated in the *inPublic template* (TPM_RC_SIZE) and copy the *inSensitive.data* to TPMT_SENSITIVE.*sensitive.sym* of the new object.

3) A TPM-generated obfuscation value is placed in TPMT_SENSITIVE.*sensitive.seedValue*. The size of the obfuscation value is the size of the digest produced by the nameAlg in *inPublic*. This value prevents the public *unique* value from leaking information about the *sensitive* area.

4) The TPMT_PUBLIC.*unique.sym* value for the new object is then generated, as shown in equation (1) below, by hashing the key and obfuscation values in the TPMT_SENSITIVE with the *nameAlg* of the object.

$$unique \coloneqq \mathbf{H}_{nameAlg}(sensitive.seedValue.buffer \mid\mid sensitive.any.buffer) \quad (1)$$

b) If the Object is an asymmetric key:

1) If *inSensitive.sensitive.data* is not the Empty Buffer, then the TPM shall return TPM_RC_VALUE.

2) A TPM-generated private key value is created with the size determined by the parameters of inPublic.publicArea.parameters.

3) If the key is a Storage Key, a TPM-generated TPMT_SENSITIVE.*seedValue* value is created; otherwise, TPMT_SENSITIVE.*seedValue.size* is set to zero.

NOTE 3 An Object that is not a storage key has no child Objects to encrypt, so it does not need a symmetric key.

4) The public *unique* value is computed from the private key according to the methods of the key type.

5) If the key is an ECC key and the scheme required by the curveID is not the same as *scheme* in the public area of the template, then the TPM shall return TPM_RC_SCHEME.

6) If the key is an ECC key and the KDF required by the curveID is not the same as *kdf* in the pubic area of the template, then the TPM shall return TPM_RC_KDF.

NOTE 4 There is currently no command in which the caller may specify the KDF to be used with an ECC decryption key. Since there is no use for this capability, the reference implementation requires that the *kdf* in the template be set to TPM_ALG_NULL or TPM_RC_KDF is returned.

c) If the Object is a *keyedHash* object:

1) If *inSensitive.sensitive.data* is an Empty Buffer, and both *sign* and *decrypt* are CLEAR in the attributes of *inPublic*, the TPM shall return TPM_RC_ATTRIBUTES. This would be a data object with no data.

NOTE 5 Revisions 134 and earlier reference code did not check the error case of *sensitiveDataOrigin* SET and an Empty Buffer. Thus, some TPM implementations may also not have included this error check.

2) If *sign* and *decrypt* are both CLEAR, or if *sign* and *decrypt* are both SET and the *scheme* in the public area of the template is not TPM_ALG_NULL, the TPM shall return TPM_RC_SCHEME.

NOTE 6 Revisions 138 and earlier did not enforce this error case.

3) If *inSensitive.sensitive.data* is not an Empty Buffer, the TPM will copy the *inSensitive.sensitive.data* to TPMT_SENSITIVE.*sensitive.bits* of the new object.

NOTE 7 The size of inSensitive.sensitive.data is limited to be no larger than MAX_SYM_DATA.

 Copyright © TCG 2006-2020

4) If *inSensitive.sensitive.data* is an Empty Buffer, a TPM-generated key value that is the size of the digest produced by the *nameAlg* in *inPublic* is placed in TPMT_SENSITIVE.*sensitive.bits*.
5) A TPM-generated obfuscation value that is the size of the digest produced by the *nameAlg* of *inPublic* is placed in TPMT_SENSITIVE.*seedValue*.
6) The TPMT_PUBLIC.*unique.keyedHash* value for the new object is then generated, as shown in equation (1) above, by hashing the key and obfuscation values in the TPMT_SENSITIVE with the *nameAlg* of the object.

For TPM2_Load(), the TPM will apply normal symmetric protections to the created TPMT_SENSITIVE to create *outPublic*.

NOTE 8 The encryption key is derived from the symmetric seed in the sensitive area of the parent.

In addition to *outPublic* and *outPrivate*, the TPM will build a TPMS_CREATION_DATA structure for the object. TPMS_CREATION_DATA.*outsideInfo* is set to *outsideInfo*. This structure is returned in *creationData*. Additionally, the digest of this structure is returned in *creationHash*, and, finally, a TPMT_TK_CREATION is created so that the association between the creation data and the object may be validated by TPM2_CertifyCreation().

If the object being created is a Storage Key and *fixedParent* is SET in the attributes of *inPublic*, then the symmetric algorithms and parameters of *inPublic* are required to match those of the parent. The algorithms that must match are *inPublic.nameAlg*, and the values in *inPublic.parameters* that select the symmetric scheme. If *inPublic.nameAlg* does not match, the TPM shall return TPM_RC_HASH.If the symmetric scheme of the key does not match, the parent, the TPM shall return TPM_RC_SYMMETRIC. The TPM shall not use different response code to differentiate between mismatches of the components of *inPublic.parameters*. However, after this verification, when using the scheme to encrypt child objects, the TPM ignores the symmetric mode and uses TPM_ALG_CFB.

NOTE 9 The symmetric scheme is a TPMT_SYM_DEF_OBJECT. In a symmetric block ciphier, it is at *inPublic.parameters.symDetail.sym* and in an asymmetric object is at *inPublic.parameters.asymDetail.symmetric*.

NOTE 10 Prior to revision 01.34, the parent asymmetric algorithms were also checked for *fixedParent* storage keys.

 Copyright © TCG 2006-2020

12.1.2 Command and Response

Table 19 — TPM2_Create Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Create
TPMI_DH_OBJECT	@parentHandle	handle of parent for new object Auth Index: 1 Auth Role: USER
TPM2B_SENSITIVE_CREATE	inSensitive	the sensitive data
TPM2B_PUBLIC	inPublic	the public template
TPM2B_DATA	outsideInfo	data that will be included in the creation data for this object to provide permanent, verifiable linkage between this object and some object owner data
TPML_PCR_SELECTION	creationPCR	PCR that will be used in creation data

Table 20 — TPM2_Create Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PRIVATE	outPrivate	the private portion of the object
TPM2B_PUBLIC	outPublic	the public portion of the created object
TPM2B_CREATION_DATA	creationData	contains a TPMS_CREATION_DATA
TPM2B_DIGEST	creationHash	digest of *creationData* using *nameAlg* of *outPublic*
TPMT_TK_CREATION	creationTicket	ticket used by TPM2_CertifyCreation() to validate that the creation data was produced by the TPM

 Copyright © TCG 2006-2020

12.1.3 Detailed Actions

[[Create]]

12.2 TPM2_Load

12.2.1 General Description

This command is used to load objects into the TPM. This command is used when both a TPM2B_PUBLIC and TPM2B_PRIVATE are to be loaded. If only a TPM2B_PUBLIC is to be loaded, the TPM2_LoadExternal command is used.

NOTE 1 Loading an object is not the same as restoring a saved object context.

The object's TPMA_OBJECT attributes will be checked according to the rules defined in "TPMA_OBJECT" in TPM 2.0 Part 2 of this specification. If the Object is a not a *keyedHash* object, and the *sign* and *encrypt* attributes are CLEAR, the TPM shall return TPM_RC_ATTRIBUTES.

Objects loaded using this command will have a Name. The Name is the concatenation of *nameAlg* and the digest of the public area using the *nameAlg.*

NOTE 2 *nameAlg* is a parameter in the public area of the inPublic structure.

If *inPrivate.size* is zero, the load will fail.

After *inPrivate.buffer* is decrypted using the symmetric key of the parent, the integrity value shall be checked before the sensitive area is used, or unmarshaled.

NOTE 3 Checking the integrity before the data is used prevents attacks on the sensitive area by fuzzing the data and looking at the differences in the response codes.

The command returns a handle for the loaded object and the Name that the TPM computed for *inPublic.public* (that is, the digest of the TPMT_PUBLIC structure in *inPublic*).

NOTE 4 The TPM-computed Name is provided as a convenience to the caller for those cases where the caller does not implement the hash algorithms specified in the *nameAlg* of the object.

NOTE 5 The returned handle is associated with the object until the object is flushed (TPM2_FlushContext) or until the next TPM2_Startup.

For all objects, the size of the key in the sensitive area shall be consistent with the key size indicated in the public area or the TPM shall return TPM_RC_KEY_SIZE.

Before use, a loaded object shall be checked to validate that the public and sensitive portions are properly linked, cryptographically. Use of an object includes use in any policy command. If the parts of the object are not properly linked, the TPM shall return TPM_RC_BINDING. If a weak symmetric key is in the sensitive portion, the TPM shall return TPM_RC_KEY.

EXAMPLE 1 For a symmetric object, the unique value in the public area shall be the digest of the sensitive key and the obfuscation value.

EXAMPLE 2 For a two-prime RSA key, the remainder when dividing the public modulus by the private key shall be zero and it shall be possible to form a private exponent from the two prime factors of the public modulus.

EXAMPLE 3 For an ECC key, the public point shall be f(x) where x is the private key.

 Copyright © TCG 2006-2020

12.2.2 Command and Response

Table 21 — TPM2_Load Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Load
TPMI_DH_OBJECT	@parentHandle	TPM handle of parent key; shall not be a reserved handle Auth Index: 1 Auth Role: USER
TPM2B_PRIVATE	inPrivate	the private portion of the object
TPM2B_PUBLIC	inPublic	the public portion of the object

Table 22 — TPM2_Load Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM_HANDLE	objectHandle	handle of type TPM_HT_TRANSIENT for the loaded object
TPM2B_NAME	name	Name of the loaded object

 Copyright © TCG 2006-2020

12.2.3 Detailed Actions

[[Load]]

12.3 TPM2_LoadExternal

12.3.1 General Description

This command is used to load an object that is not a Protected Object into the TPM. The command allows loading of a public area or both a public and sensitive area.

NOTE 1 Typical use for loading a public area is to allow the TPM to validate an asymmetric signature. Typical use for loading both a public and sensitive area is to allow the TPM to be used as a crypto accelerator.

Load of a public external object area allows the object to be associated with a hierarchy so that the correct algorithms may be used when creating tickets. The *hierarchy* parameter provides this association. If the public and sensitive portions of the object are loaded, *hierarchy* is required to be TPM_RH_NULL.

NOTE 2 If both the public and private portions of an object are loaded, the object is not allowed to appear to be part of a hierarchy.

The object's TPMA_OBJECT attributes will be checked according to the rules defined in "TPMA_OBJECT" in TPM 2.0 Part 2. In particular, *fixedTPM*, *fixedParent*, and *restricted* shall be CLEAR if *inPrivate* is not the Empty Buffer.

NOTE 3 The duplication status of a public key needs to be able to be the same as the full key which may be resident on a different TPM. If both the public and private parts of the key are loaded, then it is not possible for the key to be either *fixedTPM* or *fixedParent*, since, its private area would not be available in the clear to load.

Objects loaded using this command will have a Name. The Name is the *nameAlg* of the object concatenated with the digest of the public area using the *nameAlg.* The Qualified Name for the object will be the same as its Name. The TPM will validate that the *authPolicy* is either the size of the digest produced by *nameAlg* or the Empty Buffer.

NOTE 4 If *nameAlg* is TPM_ALG_NULL, then the Name is the Empty Buffer. When the authorization value for an object with no Name is computed, no Name value is included in the HMAC. To ensure that these unnamed entities are not substituted, they should have an *authValue* that is statistically unique.

NOTE 5 The digest size for TPM_ALG_NULL is zero.

If the *nameAlg* is TPM_ALG_NULL, the TPM shall not verify the cryptographic binding between the public and sensitive areas, but the TPM will validate that the size of the key in the sensitive area is consistent with the size indicated in the public area. If it is not, the TPM shall return TPM_RC_KEY_SIZE.

NOTE 6 For an ECC object, the TPM will verify that the public key is on the curve of the key before the public area is used.

If *nameAlg* is not TPM_ALG_NULL, then the same consistency checks between *inPublic* and *inPrivate* are made as for TPM2_Load().

NOTE 7 Consistency checks are necessary because an object with a Name needs to have the public and sensitive portions cryptographically bound so that an attacker cannot mix pubic and sensitive areas.

The command returns a handle for the loaded object and the Name that the TPM computed for *inPublic.public* (that is, the TPMT_PUBLIC structure in *inPublic*).

NOTE 8 The TPM-computed Name is provided as a convenience to the caller for those cases where the caller does not implement the hash algorithm specified in the *nameAlg* of the object.

 Copyright © TCG 2006-2020

The *hierarchy* parameter associates the external object with a hierarchy. External objects are flushed when their associated hierarchy is disabled. If *hierarchy* is TPM_RH_NULL, the object is part of no hierarchy, and there is no implicit flush.

If *hierarchy* is TPM_RH_NULL or *nameAlg* is TPM_ALG_NULL, a ticket produced using the object shall be a NULL Ticket.

EXAMPLE If a key is loaded with hierarchy set to TPM_RH_NULL, then TPM2_VerifySignature() will produce a NULL Ticket of the required type.

External objects are Temporary Objects. The saved external object contexts shall be invalidated at the next TPM Reset.

If a weak symmetric key is in the sensitive area, the TPM shall return TPM_RC_KEY.

For an RSA key, the private exponent is computed using the two prime factors of the public modulus. One of the primes is P, and the second prime (Q) is found by dividing the public modulus by P. A TPM may return an error (TPM_RC_BINDING) if the bit size of P and Q are not the same."

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

12.3.2 Command and Response

Table 23 — TPM2_LoadExternal Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit, encrypt, or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_LoadExternal
TPM2B_SENSITIVE	inPrivate	the sensitive portion of the object (optional)
TPM2B_PUBLIC+	inPublic	the public portion of the object
TPMI_RH_HIERARCHY+	hierarchy	hierarchy with which the object area is associated

Table 24 — TPM2_LoadExternal Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM_HANDLE	objectHandle	handle of type TPM_HT_TRANSIENT for the loaded object
TPM2B_NAME	name	name of the loaded object

 Copyright © TCG 2006-2020

12.3.3 Detailed Actions

[[LoadExternal]]

12.4 TPM2_ReadPublic

12.4.1 General Description

This command allows access to the public area of a loaded object.

Use of the *objectHandle* does not require authorization.

NOTE Since the caller is not likely to know the public area of the object associated with *objectHandle*, it would not be possible to include the Name associated with *objectHandle* in the *cpHash* computation.

If *objectHandle* references a sequence object, the TPM shall return TPM_RC_SEQUENCE.

Copyright © TCG 2006-2020

12.4.2 Command and Response

Table 25 — TPM2_ReadPublic Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ReadPublic
TPMI_DH_OBJECT	objectHandle	TPM handle of an object Auth Index: None

Table 26 — TPM2_ReadPublic Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PUBLIC	outPublic	structure containing the public area of an object
TPM2B_NAME	name	name of the object
TPM2B_NAME	qualifiedName	the Qualified Name of the object

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

12.4.3 Detailed Actions

[[ReadPublic]]

12.5 TPM2_ActivateCredential

12.5.1 General Description

This command enables the association of a credential with an object in a way that ensures that the TPM has validated the parameters of the credentialed object.

If both the public and private portions of *activateHandle* and *keyHandle* are not loaded, then the TPM shall return TPM_RC_AUTH_UNAVAILABLE.

If *keyHandle* is not a Storage Key, then the TPM shall return TPM_RC_TYPE.

Authorization for *activateHandle* requires the ADMIN role.

The key associated with *keyHandle* is used to recover a seed from secret, which is the encrypted seed. The Name of the object associated with activateHandle and the recovered seed are used in a KDF to recover the symmetric key. The recovered seed (but not the Name) is used in a KDF to recover the HMAC key.

The HMAC is used to validate that the *credentialBlob* is associated with *activateHandle* and that the data in *credentialBlob* has not been modified. The linkage to the object associated with activateHandle is achieved by including the Name in the HMAC calculation.

If the integrity checks succeed, *credentialBlob* is decrypted and returned as *certInfo*.

NOTE The output *certInfo* parameter is an application defined value. It is typically a symmetric key or seed that is used to decrypt a certificate. See the TPM2_MakeCredential *credential* input parameter.

 Copyright © TCG 2006-2020

12.5.2 Command and Response

Table 27 — TPM2_ActivateCredential Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ActivateCredential
TPMI_DH_OBJECT	@activateHandle	handle of the object associated with certificate in *credentialBlob* Auth Index: 1 Auth Role: ADMIN
TPMI_DH_OBJECT	@keyHandle	loaded key used to decrypt the TPMS_SENSITIVE in *credentialBlob* Auth Index: 2 Auth Role: USER
TPM2B_ID_OBJECT	credentialBlob	the credential
TPM2B_ENCRYPTED_SECRET	secret	*keyHandle* algorithm-dependent encrypted seed that protects *credentialBlob*

Table 28 — TPM2_ActivateCredential Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	certInfo	the decrypted certificate information the data should be no larger than the size of the digest of the *nameAlg* associated with *keyHandle*

 Copyright © TCG 2006-2020

12.5.3 Detailed Actions

[[ActivateCredential]]

 Copyright © TCG 2006-2020

12.6 TPM2_MakeCredential

12.6.1 General Description

This command allows the TPM to perform the actions required of a Certificate Authority (CA) in creating a TPM2B_ID_OBJECT containing an activation credential.

NOTE The input *credential* parameter is an application defined value. It is typically a symmetric key or seed that is used to encrypt a certificate. See the TPM2_ActivateCredential *certInfo* output parameter.

The TPM will produce a TPM2B_ID_OBJECT according to the methods in "Credential Protection" in TPM 2.0 Part 1.

The loaded public area referenced by *handle* is required to be the public area of a Storage key, otherwise, the credential cannot be properly sealed.

This command does not use any TPM secrets nor does it require authorization. It is a convenience function, using the TPM to perform cryptographic calculations that could be done externally.

 Copyright © TCG 2006-2020

12.6.2 Command and Response

Table 29 — TPM2_MakeCredential Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit, encrypt, or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_MakeCredential
TPMI_DH_OBJECT	handle	loaded public area, used to encrypt the sensitive area containing the credential key Auth Index: None
TPM2B_DIGEST	credential	the credential information
TPM2B_NAME	objectName	Name of the object to which the credential applies

Table 30 — TPM2_MakeCredential Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ID_OBJECT	credentialBlob	the credential
TPM2B_ENCRYPTED_SECRET	secret	*handle* algorithm-dependent data that wraps the key that encrypts *credentialBlob*

 Copyright © TCG 2006-2020

12.6.3 Detailed Actions

[[MakeCredential]]

12.7 TPM2_Unseal

12.7.1 General Description

This command returns the data in a loaded Sealed Data Object.

NOTE 1 A random, TPM-generated, Sealed Data Object may be created by the TPM with TPM2_Create() or TPM2_CreatePrimary() using the template for a Sealed Data Object.

NOTE 2 TPM 1.2 hard coded PCR authorization. TPM 2.0 PCR authorization requires a policy.

The returned value may be encrypted using authorization session encryption.

If either *restricted*, *decrypt,* or *sign* is SET in the attributes of *itemHandle,* then the TPM shall return TPM_RC_ATTRIBUTES. If the *type* of *itemHandle* is not TPM_ALG_KEYEDHASH, then the TPM shall return TPM_RC_TYPE.

 Copyright © TCG 2006-2020

12.7.2 Command and Response

Table 31 — TPM2_Unseal Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Unseal
TPMI_DH_OBJECT	@itemHandle	handle of a loaded data object Auth Index: 1 Auth Role: USER

Table 32 — TPM2_Unseal Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_SENSITIVE_DATA	outData	unsealed data Size of *outData* is limited to be no more than 128 octets.

 Copyright © TCG 2006-2020

12.7.3 Detailed Actions

[[Unseal]]

12.8 TPM2_ObjectChangeAuth

12.8.1 General Description

This command is used to change the authorization secret for a TPM-resident object.

If successful, a new private area for the TPM-resident object associated with *objectHandle* is returned, which includes the new authorization value.

This command does not change the authorization of the TPM-resident object on which it operates. Therefore, the old authValue (of the TPM-resident object) is used when generating the response HMAC key if required.

NOTE 1 The returned *outPrivate* will need to be loaded before the new authorization will apply.

NOTE 2 The TPM-resident object may be persistent and changing the authorization value of the persistent object could prevent other users from accessing the object. This is why this command does not change the TPM-resident object.

EXAMPLE If a persistent key is being used as a Storage Root Key and the authorization of the key is a well-known value so that the key can be used generally, then changing the authorization value in the persistent key would deny access to other users.

This command may not be used to change the authorization value for an NV Index or a Primary Object.

NOTE 3 If an NV Index is to have a new authorization, it is done with TPM2_NV_ChangeAuth().

NOTE 4 If a Primary Object is to have a new authorization, it needs to be recreated (TPM2_CreatePrimary()).

 Copyright © TCG 2006-2020

12.8.2 Command and Response

Table 33 — TPM2_ObjectChangeAuth Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ObjectChangeAuth
TPMI_DH_OBJECT	@objectHandle	handle of the object Auth Index: 1 Auth Role: ADMIN
TPMI_DH_OBJECT	parentHandle	handle of the parent Auth Index: None
TPM2B_AUTH	newAuth	new authorization value

Table 34 — TPM2_ObjectChangeAuth Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PRIVATE	outPrivate	private area containing the new authorization value

 Copyright © TCG 2006-2020

12.8.3 Detailed Actions

[[ObjectChangeAuth]]

12.9 TPM2_CreateLoaded

12.9.1 General Description

This command creates an object and loads it in the TPM. This command allows creation of any type of object (Primary, Ordinary, or Derived) depending on the type of *parentHandle.* If *parentHandle* references a Primary Seed, then a Primary Object is created; if *parentHandle* references a Storage Parent, then an Ordinary Object is created; and if *parentHandle* references a Derivation Parent, then a Derived Object is generated.

The input validation is the same as for TPM2_Create() and TPM2_CreatePrimary() with one exception: when *parentHandle* references a Derivation Parent, then *sensitiveDataOrigin* in *inPublic* is required to be CLEAR.

Note 1 In the general descriptions of TPM2_Create() and TPM2_CreatePrimary() the validations refer to a TPMT_PUBLIC structure that is in *inPublic.* For TPM2_CreateLoaded(), *inPublic* is a TPM2B_TEMPLATE that may contain a TPMT_PUBLIC that is used for object creation. For object derivation, the *unique* field can contain a *label* and *context* that are used in the derivation process. To allow both the TPMT_PUBLIC and the derivation variation, a TPM2B_TEMPLATE is used. When referring to the checks in TPM2_Create() and TPM2_CreatePrimary(), TPM2B_TEMPLATE should be assumed to contain a TPMT_PUBLIC.

If *parentHandle* references a Derivation Parent, then the TPM may return TPM_RC_TYPE if the key type to be generated is an RSA key.

If *parentHandle* references a Derivation Parent or a Primary Seed, then *outPrivate* will be an Empty Buffer.

NOTE 2 Returning outPrivate would imply that the returned primary or derived object can be loaded and it cannot. It can only be re-derived.

A primary key cannot be loaded is because loading a key is a way to attack the protections of a key (e.g. using DPA). A saved context for a primary object is protected. The TPM will go into failure mode if the integrity of a saved context is good but the fingerprint doesn't decrypt. It is not possible to have these protections on loaded objects because this would be a simple way for an attacker to put the TPM into failure mode Saved contexts are assumed to be under control of the driver but loaded objects are not.

If all objects were derived from their parents then, load could not be used as an attack. However, that would preclude importation of objects and key hierarchies.

NOTE 3 Unlike TPM2_Create() and TPM2_CreatePrimary(), this command does not return creation data. If creation data is needed, then TPM2_Create() or TPM2_CreatePrimary() should be used.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

12.9.2 Command and Response

Table 35 — TPM2_CreateLoaded Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_CreateLoade
TPMI_DH_PARENT+	@parentHandle	Handle of a transient storage key, a persistent storage key, TPM_RH_ENDORSEMENT, TPM_RH_OWNER, TPM_RH_PLATFORM+{PP}, or TPM_RH_NULL Auth Index: 1 Auth Role: USER
TPM2B_SENSITIVE_CREATE	inSensitive	the sensitive data, see TPM 2.0 Part 1 Sensitive Values
TPM2B_TEMPLATE	inPublic	the public template

Table 36 — TPM2_CreateLoaded Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM_HANDLE	objectHandle	handle of type TPM_HT_TRANSIENT for created object
TPM2B_PRIVATE	outPrivate	the sensitive area of the object (optional)
TPM2B_PUBLIC	outPublic	the public portion of the created object
TPM2B_NAME	name	the name of the created object

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

12.9.3 Detailed Actions

[[CreateLoaded]]

 Copyright © TCG 2006-2020

13 Duplication Commands

13.1 TPM2_Duplicate

13.1.1 General Description

This command duplicates a loaded object so that it may be used in a different hierarchy. The new parent key for the duplicate may be on the same or different TPM or TPM_RH_NULL. Only the public area of *newParentHandle* is required to be loaded.

NOTE 1 Since the new parent may only be extant on a different TPM, it is likely that the new parent's sensitive area could not be loaded in the TPM from which *objectHandle* is being duplicated.

If *encryptedDuplication* is SET in the object being duplicated, then the TPM shall return TPM_RC_SYMMETRIC if *symmetricAlg.algorithm* is TPM_ALG_NULL or TPM_RC_HIERARCHY if *newParentHandle* is TPM_RH_NULL.

The authorization for this command shall be with a policy session.

If *fixedParent* of *objectHandle→attributes* is SET, the TPM shall return TPM_RC_ATTRIBUTES. If *objectHandle→nameAlg* is TPM_ALG_NULL, the TPM shall return TPM_RC_TYPE.

The *policySession→commandCode* parameter in the policy session is required to be TPM_CC_Duplicate to indicate that authorization for duplication has been provided. This indicates that the policy that is being used is a policy that is for duplication, and not a policy that would approve another use. That is, authority to use an object does not grant authority to duplicate the object.

The policy is likely to include cpHash in order to restrict where duplication can occur. If TPM2_PolicyCpHash() has been executed as part of the policy, the *policySession→cpHash* is compared to the cpHash of the command.

If TPM2_PolicyDuplicationSelect() has been executed as part of the policy, the *policySession→nameHash* is compared to

$$\mathbf{H}_{policyAlg}(objectHandle{\rightarrow}Name \mid\mid newParentHandle{\rightarrow}Name) \quad (2)$$

If the compared hashes are not the same, then the TPM shall return TPM_RC_POLICY_FAIL.

NOTE 2 It is allowed that policySesion→nameHash and policySession→cpHash share the same memory space.

NOTE 3 A duplication policy is not required to have either TPM2_PolicyDuplicationSelect() or TPM2_PolicyCpHash() as part of the policy. If neither is present, then the duplication policy may be satisfied with a policy that only contains TPM2_PolicyCommandCode(*code* = TPM_CC_Duplicate).

The TPM shall follow the process of encryption defined in the "Duplication" subclause of "Protected Storage Hierarchy" in TPM 2.0 Part 1.

 Copyright © TCG 2006-2020

13.1.2 Command and Response

Table 37 — TPM2_Duplicate Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Duplicate
TPMI_DH_OBJECT	@objectHandle	loaded object to duplicate Auth Index: 1 Auth Role: DUP
TPMI_DH_OBJECT+	newParentHandle	shall reference the public area of an asymmetric key Auth Index: None
TPM2B_DATA	encryptionKeyIn	optional symmetric encryption key The size for this key is set to zero when the TPM is to generate the key. This parameter may be encrypted.
TPMT_SYM_DEF_OBJECT+	symmetricAlg	definition for the symmetric algorithm to be used for the inner wrapper may be TPM_ALG_NULL if no inner wrapper is applied

Table 38 — TPM2_Duplicate Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DATA	encryptionKeyOut	If the caller provided an encryption key or if *symmetricAlg* was TPM_ALG_NULL, then this will be the Empty Buffer; otherwise, it shall contain the TPM-generated, symmetric encryption key for the inner wrapper.
TPM2B_PRIVATE	duplicate	private area that may be encrypted by *encryptionKeyIn*; and may be doubly encrypted
TPM2B_ENCRYPTED_SECRET	outSymSeed	seed protected by the asymmetric algorithms of new parent (NP)

 Copyright © TCG 2006-2020

13.1.3 Detailed Actions

[[Duplicate]]

13.2 TPM2_Rewrap

13.2.1 General Description

This command allows the TPM to serve in the role as a Duplication Authority. If proper authorization for use of the *oldParent* is provided, then an HMAC key and a symmetric key are recovered from *inSymSeed* and used to integrity check and decrypt *inDuplicate.* A new protection seed value is generated according to the methods appropriate for *newParent* and the blob is re-encrypted and a new integrity value is computed. The re-encrypted blob is returned in *outDuplicate* and the symmetric key returned in *outSymKey*.

In the rewrap process, L is "DUPLICATE" (see TPM 2.0 Part 1, *Terms and Definitions*).

If *inSymSeed* has a zero length, then *oldParent* is required to be TPM_RH_NULL and no decryption of *inDuplicate* takes place.

If *newParent* is TPM_RH_NULL, then no encryption is performed on *outDuplicate. outSymSeed* will have a zero length. See TPM 2.0 Part 2 *encryptedDuplication*.

 Copyright © TCG 2006-2020

13.2.2 Command and Response

Table 39 — TPM2_Rewrap Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Rewrap
TPMI_DH_OBJECT+	@oldParent	parent of object Auth Index: 1 Auth Role: User
TPMI_DH_OBJECT+	newParent	new parent of the object Auth Index: None
TPM2B_PRIVATE	inDuplicate	an object encrypted using symmetric key derived from *inSymSeed*
TPM2B_NAME	name	the Name of the object being rewrapped
TPM2B_ENCRYPTED_SECRET	inSymSeed	the seed for the symmetric key and HMAC key needs *oldParent* private key to recover the seed and generate the symmetric key

Table 40 — TPM2_Rewrap Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PRIVATE	outDuplicate	an object encrypted using symmetric key derived from *outSymSeed*
TPM2B_ENCRYPTED_SECRET	outSymSeed	seed for a symmetric key protected by *newParent* asymmetric key

 Copyright © TCG 2006-2020

13.2.3 Detailed Actions

[[Rewrap]]

 Copyright © TCG 2006-2020

13.3 TPM2_Import

13.3.1 General Description

This command allows an object to be encrypted using the symmetric encryption values of a Storage Key. After encryption, the object may be loaded and used in the new hierarchy. The imported object (*duplicate*) may be singly encrypted, multiply encrypted, or unencrypted.

If *fixedTPM* or *fixedParent* is SET in *objectPublic*, the TPM shall return TPM_RC_ATTRIBUTES.

If *encryptedDuplication* is SET in the object referenced by *parentHandle* and *encryptedDuplication* is CLEAR in *objectPublic,* the TPM may return TPM_RC_ATTRIBUTES.

If encryptedDuplication is SET in objectPublic, then *inSymSeed* and *encryptionKey* shall not be Empty buffers (TPM_RC_ATTRIBUTES). Recovery of the sensitive data of the object occurs in the TPM in a multi--step process in the following order:

a) If *inSymSeed* has a non-zero size:

1) The asymmetric parameters and private key of *parentHandle* are used to recover the seed used in the creation of the HMAC key and encryption keys used to protect the duplication blob.

NOTE 1 When recovering the seed from *inSymSeed*, *L* is "DUPLICATE".

2) The integrity value in *duplicate.buffer.integrityOuter* is used to verify the integrity of the data blob, which is the remainder of *duplicate.buffer* (TPM_RC_INTEGRITY).

NOTE 2 The data blob will contain a TPMT_SENSITIVE and may contain a TPM2B_DIGEST for the *innerIntegrity.*

3) The symmetric key recovered in 1) is used to decrypt the data blob.

NOTE 3 Checking the integrity before the data is used prevents attacks on the sensitive area by fuzzing the data and looking at the differences in the response codes.

b) If *encryptionKey* is not an Empty Buffer:

1) Use *encryptionKey* to decrypt the inner blob.

2) Use the TPM2B_DIGEST at the start of the inner blob to verify the integrity of the inner blob (TPM_RC_INTEGRITY).

c) Unmarshal the sensitive area

NOTE 4 It is not necessary to validate that the sensitive area data is cryptographically bound to the public area other than that the Name of the public area is included in the HMAC. However, if the binding is not validated by this command, the binding must be checked each time the object is loaded. For an object that is imported under a parent with *fixedTPM* SET, binding need only be checked at import. If the parent has *fixedTPM* CLEAR, then the binding needs to be checked each time the object is loaded, or before the TPM performs an operation for which the binding affects the outcome of the operation (for example, TPM2_PolicySigned() or TPM2_Certify()).

Similarly, if the new parent's *fixedTPM* is set, the *encryptedDuplication* state need only be checked at import.

If the new parent is not *fixedTPM*, then that object will be loadable on any TPM (including SW versions) on which the new parent exists. This means that, each time an object is loaded under a parent that is not *fixedTPM*, it is necessary to validate all of the properties of that object. If the parent is *fixedTPM*, then the new private blob is integrity protected by the TPM that "owns" the parent. So, it is sufficient to validate the object's properties (attribute and public-private binding) on import and not again.

If a weak symmetric key is being imported, the TPM shall return TPM_RC_KEY.

After integrity checks and decryption, the TPM will create a new symmetrically encrypted private area using the encryption key of the parent.

NOTE 5 The symmetric re-encryption is the normal integrity generation and symmetric encryption applied to a child object.

NOTE 6 Revision 01.16 of this specification required the ECC private key in *duplicate* to be padded.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

13.3.2 Command and Response

Table 41 — TPM2_Import Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Import
TPMI_DH_OBJECT	@parentHandle	the handle of the new parent for the object Auth Index: 1 Auth Role: USER
TPM2B_DATA	encryptionKey	the optional symmetric encryption key used as the inner wrapper for *duplicate* If *symmetricAlg* is TPM_ALG_NULL, then this parameter shall be the Empty Buffer.
TPM2B_PUBLIC	objectPublic	the public area of the object to be imported This is provided so that the integrity value for *duplicate* and the object attributes can be checked. NOTE Even if the integrity value of the object is not checked on input, the object Name is required to create the integrity value for the imported object.
TPM2B_PRIVATE	duplicate	the symmetrically encrypted duplicate object that may contain an inner symmetric wrapper
TPM2B_ENCRYPTED_SECRET	inSymSeed	the seed for the symmetric key and HMAC key *inSymSeed* is encrypted/encoded using the algorithms of *newParent*.
TPMT_SYM_DEF_OBJECT+	symmetricAlg	definition for the symmetric algorithm to use for the inner wrapper If this algorithm is TPM_ALG_NULL, no inner wrapper is present and *encryptionKey* shall be the Empty Buffer.

Table 42 — TPM2_Import Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PRIVATE	outPrivate	the sensitive area encrypted with the symmetric key of *parentHandle*

 Copyright © TCG 2006-2020

13.3.3 Detailed Actions

[[Import]]

14 Asymmetric Primitives

14.1 Introduction

The commands in this clause provide low-level primitives for access to the asymmetric algorithms implemented in the TPM. Many of these commands are only allowed if the asymmetric key is an unrestricted key.

14.2 TPM2_RSA_Encrypt

14.2.1 General Description

This command performs RSA encryption using the indicated padding scheme according to IETF RFC 8017. If the *scheme* of *keyHandle* is TPM_ALG_NULL, then the caller may use *inScheme* to specify the padding scheme. If *scheme* of keyHandle is not TPM_ALG_NULL, then *inScheme* shall either be TPM_ALG_NULL or be the same as *scheme* (TPM_RC_SCHEME).

The key referenced by *keyHandle* is required to be an RSA key (TPM_RC_KEY).

The three types of allowed padding are:

1) TPM_ALG_OAEP – Data is OAEP padded as described in 7.1 of IETF RFC 8017 (PKCS#1). The only supported mask generation is MGF1.

2) TPM_ALG_RSAES – Data is padded as described in 7.2 of IETF RFC 8017 (PKCS#1).

3) TPM_ALG_NULL – Data is not padded by the TPM and the TPM will treat *message* as an unsigned integer and perform a modular exponentiation of *message* using the public exponent of the key referenced by *keyHandle*. This scheme is only used if both the *scheme* in the key referenced by *keyHandle* is TPM_ALG_NULL, and the *inScheme* parameter of the command is TPM_ALG_NULL. The input value cannot be larger than the public modulus of the key referenced by *keyHandle*.

Table 43 — Padding Scheme Selection

<table>
<tr><th>keyHandle→scheme</th><th>inScheme</th><th>padding scheme used</th></tr>
<tr><td rowspan="3">TPM_ALG_NULL</td><td>TPM_ALG_NULL</td><td>none</td></tr>
<tr><td>TPM_ALG_RSAES</td><td>RSAES</td></tr>
<tr><td>TPM_ALG_OAEP</td><td>OAEP</td></tr>
<tr><td rowspan="3">TPM_ALG_RSAES</td><td>TPM_ALG_NULL</td><td>RSAES</td></tr>
<tr><td>TPM_ALG_RSAES</td><td>RSAES</td></tr>
<tr><td>TPM_ALG_OAEP</td><td>error (TPM_RC_SCHEME)</td></tr>
<tr><td rowspan="3">TPM_ALG_OAEP</td><td>TPM_ALG_NULL</td><td>OAEP</td></tr>
<tr><td>TPM_ALG_RSAES</td><td>error (TPM_RC_SCHEME)</td></tr>
<tr><td>TPM_ALG_OAEP</td><td>OAEP</td></tr>
</table>

After padding, the data is RSAEP encrypted according to 5.1.1 of IETF RFC 8017 (PKCS#1).

If *inScheme* is used, and the scheme requires a hash algorithm it may not be TPM_ALG_NULL.

NOTE 1 Because only the public portion of the key needs to be loaded for this command, the caller can manipulate the attributes of the key in any way desired. As a result, the TPM shall not check the consistency of the attributes. The only property checking is that the key is an RSA key and that the padding scheme is supported.

The *message* parameter is limited in size by the padding scheme according to the following table:

Table 44 — Message Size Limits Based on Padding

Scheme	Maximum Message Length (*mLen*) in Octets	Comments
TPM_ALG_OAEP	$mLen \leq k - 2hLen - 2$	
TPM_ALG_RSAES	$mLen \leq k - 11$	
TPM_ALG_NULL	$mLen \leq k$	The numeric value of the message must be less than the numeric value of the public modulus (n).
NOTES 1) k ≔ the number of byes in the public modulus 2) $hLen$ ≔ the number of octets in the digest produced by the hash algorithm used in the process		

The *label* parameter is optional. If provided (*label.size* != 0) then the TPM shall return TPM_RC_VALUE if the last octet in *label* is not zero. The terminating octet of zero is included in the *label* used in the padding scheme.

NOTE 2 If the scheme does not use a label, the TPM will still verify that label is properly formatted if label is present.

NOTE 3 Specifications before version 1.54 stated that *label* is truncated after the first zero octet. Applications should not include embedded zero bytes for compatibility.

The function returns padded and encrypted value *outData*.

The *message* parameter in the command may be encrypted using parameter encryption.

NOTE 4 Only the public area of *keyHandle* is required to be loaded. A public key may be loaded with any desired scheme. If the scheme is to be changed, a different public area must be loaded.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

14.2.2 Command and Response

Table 45 — TPM2_RSA_Encrypt Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit, encrypt, or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_RSA_Encrypt
TPMI_DH_OBJECT	keyHandle	reference to public portion of RSA key to use for encryption Auth Index: None
TPM2B_PUBLIC_KEY_RSA	message	message to be encrypted NOTE 1 The data type was chosen because it limits the overall size of the input to no greater than the size of the largest RSA public key. This may be larger than allowed for *keyHandle*.
TPMT_RSA_DECRYPT+	inScheme	the padding scheme to use if *scheme* associated with *keyHandle* is TPM_ALG_NULL
TPM2B_DATA	label	optional label *L* to be associated with the message Size of the buffer is zero if no label is present NOTE 2 See description of label above.

Table 46 — TPM2_RSA_Encrypt Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PUBLIC_KEY_RSA	outData	encrypted output

 Copyright © TCG 2006-2020

14.2.3 Detailed Actions

[[RSA_Encrypt]]

 Copyright © TCG 2006-2020

14.3 TPM2_RSA_Decrypt

14.3.1 General Description

This command performs RSA decryption using the indicated padding scheme according to IETF RFC 8017 ((PKCS#1).

The scheme selection for this command is the same as for TPM2_RSA_Encrypt() and is shown in Table 43.

The key referenced by *keyHandle* shall be an RSA key (TPM_RC_KEY) with *restricted* CLEAR and *decrypt* SET (TPM_RC_ATTRIBUTES).

This command uses the private key of *keyHandle* for this operation and authorization is required.

The TPM will perform a modular exponentiation of ciphertext using the private exponent associated with *keyHandle* (this is described in IETF RFC 8017 (PKCS#1), clause 5.1.2). It will then validate the padding according to the selected scheme. If the padding checks fail, TPM_RC_VALUE is returned. Otherwise, the data is returned with the padding removed. If no padding is used, the returned value is an unsigned integer value that is the result of the modular exponentiation of *cipherText* using the private exponent of *keyHandle*. The returned value may include leading octets zeros so that it is the same size as the public modulus. For the other padding schemes, the returned value will be smaller than the public modulus but will contain all the data remaining after padding is removed and this may include leading zeros if the original encrypted value contained leading zeros.

If a label is used in the padding process of the scheme during encryption, the *label* parameter is required to be present in the decryption process and *label* is required to be the same in both cases. If label is not the same, the decrypt operation is very likely to fail ((TPM_RC_VALUE). If *label* is present (*label.size* != 0), it shall be a byte stream whose last byte is zero or the TPM will return TPM_RC_VALUE.

NOTE 1 The size of *label* includes the terminating null.

The *message* parameter in the response may be encrypted using parameter encryption.

If *inScheme* is used, and the scheme requires a hash algorithm it may not be TPM_ALG_NULL.

If the scheme does not require a label, the value in *label* is not used but the size of the label field is checked for consistency with the indicated data type (TPM2B_DATA). That is, the field may not be larger than allowed for a TPM2B_DATA.

14.3.2 Command and Response

Table 47 — TPM2_RSA_Decrypt Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_RSA_Decrypt
TPMI_DH_OBJECT	@keyHandle	RSA key to use for decryption Auth Index: 1 Auth Role: USER
TPM2B_PUBLIC_KEY_RSA	cipherText	cipher text to be decrypted NOTE An encrypted RSA data block is the size of the public modulus.
TPMT_RSA_DECRYPT+	inScheme	the padding scheme to use if *scheme* associated with *keyHandle* is TPM_ALG_NULL
TPM2B_DATA	label	label whose association with the message is to be verified

Table 48 — TPM2_RSA_Decrypt Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PUBLIC_KEY_RSA	message	decrypted output

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

14.3.3 Detailed Actions

[[RSA_Decrypt]]

14.4 TPM2_ECDH_KeyGen

14.4.1 General Description

This command uses the TPM to generate an ephemeral key pair (d_e, Q_e where $Q_e \coloneqq [d_e]G$). It uses the private ephemeral key and a loaded public key (Q_S) to compute the shared secret value ($P \coloneqq [hd_e]Q_S$).

keyHandle shall refer to a loaded, ECC key (TPM_RC_KEY). The sensitive portion of this key need not be loaded.

The curve parameters of the loaded ECC key are used to generate the ephemeral key.

NOTE This function is the equivalent of encrypting data to another object's public key. The *seed* value is used in a KDF to generate a symmetric key and that key is used to encrypt the data. Once the data is encrypted and the symmetric key discarded, only the object with the private portion of the *keyHandle* will be able to decrypt it.

The *zPoint* in the response may be encrypted using parameter encryption.

 Copyright © TCG 2006-2020

14.4.2 Command and Response

Table 49 — TPM2_ECDH_KeyGen Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ECDH_KeyGen
TPMI_DH_OBJECT	keyHandle	Handle of a loaded ECC key public area. Auth Index: None

Table 50 — TPM2_ECDH_KeyGen Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ECC_POINT	zPoint	results of $P \coloneqq h[d_e]Q_s$
TPM2B_ECC_POINT	pubPoint	generated ephemeral public point (Q_e)

 Copyright © TCG 2006-2020

14.4.3 Detailed Actions

[[ECDH_KeyGen]]

14.5 TPM2_ECDH_ZGen

14.5.1 General Description

This command uses the TPM to recover the *Z* value from a public point (Q_B) and a private key (d_s). It will perform the multiplication of the provided *inPoint* (Q_B) with the private key (d_s) and return the coordinates of the resultant point ($Z = (x_Z, y_Z) \coloneqq [hd_s]Q_B$; where h is the cofactor of the curve).

keyHandle shall refer to a loaded, ECC key (TPM_RC_KEY) with the *restricted* attribute CLEAR and the *decrypt* attribute SET (TPM_RC_ATTRIBUTES).

NOTE While TPM_RC_ATTRIBUTES is preferred, TPM_RC_KEY is acceptable.

The *scheme* of the key referenced by *keyHandle* is required to be either TPM_ALG_ECDH or TPM_ALG_NULL (TPM_RC_SCHEME).

inPoint is required to be on the curve of the key referenced by *keyHandle* (TPM_RC_ECC_POINT).

The parameters of the key referenced by *keyHandle* are used to perform the point multiplication.

14.5.2 Command and Response

Table 51 — TPM2_ECDH_ZGen Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ECDH_ZGen
TPMI_DH_OBJECT	@keyHandle	handle of a loaded ECC key Auth Index: 1 Auth Role: USER
TPM2B_ECC_POINT	inPoint	a public key

Table 52 — TPM2_ECDH_ZGen Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ECC_POINT	outPoint	X and Y coordinates of the product of the multiplication $Z = (x_Z, y_Z) \coloneqq [hd_S]Q_B$

 Copyright © TCG 2006-2020

14.5.3 Detailed Actions

[[ECDH_ZGen]]

14.6 TPM2_ECC_Parameters

14.6.1 General Description

This command returns the parameters of an ECC curve identified by its TCG-assigned *curveID*.

The value returned is the same as that from the TCG Algorithm Registry, but may not be the same size.

EXAMPLE The value 01 may be returned as 00000001.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

14.6.2 Command and Response

Table 53 — TPM2_ECC_Parameters Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ECC_Parameters
TPMI_ECC_CURVE	curveID	parameter set selector

Table 54 — TPM2_ECC_Parameters Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMS_ALGORITHM_DETAIL_ECC	parameters	ECC parameters for the selected curve

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

14.6.3 Detailed Actions

[[ECC_Parameters]]

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

14.7 TPM2_ZGen_2Phase

14.7.1 General Description

This command supports two-phase key exchange protocols. The command is used in combination with TPM2_EC_Ephemeral(). TPM2_EC_Ephemeral() generates an ephemeral key and returns the public point of that ephemeral key along with a numeric value that allows the TPM to regenerate the associated private key.

The input parameters for this command are a static public key (*inQsU*), an ephemeral key (*inQeU*) from party B, and the *commitCounter* returned by TPM2_EC_Ephemeral(). The TPM uses the counter value to regenerate the ephemeral private key ($d_{e,V}$) and the associated public key ($Q_{e,V}$). *keyA* provides the static ephemeral elements $d_{s,V}$ and $Q_{s,V}$. This provides the two pairs of ephemeral and static keys that are required for the schemes supported by this command.

The TPM will compute $Z\ or\ Z_s$ and Z_e according to the selected scheme. If the scheme is not a two-phase key exchange scheme or if the scheme is not supported, the TPM will return TPM_RC_SCHEME.

It is an error if *inQsB* or *inQeB* are not on the curve of *keyA* (TPM_RC_ECC_POINT).

The two-phase key schemes that were assigned an algorithm ID as of the time of the publication of this specification are TPM_ALG_ECDH, TPM_ALG_ECMQV, and TPM_ALG_SM2.

If this command is supported, then support for TPM_ALG_ECDH is required. Support for TPM_ALG_ECMQV or TPM_ALG_SM2 is optional.

NOTE 1 If SM2 is supported and this command is supported, then the implementation is required to support the key exchange protocol of SM2, part 3.

For TPM_ALG_ECDH *outZ1* will be Zs and *outZ2* will Ze as defined in 6.1.1.2 of SP800-56A.

NOTE 2 An unrestricted decryption key using ECDH may be used in either TPM2_ECDH_ZGen() or TPM2_ZGen_2Phase as the computation done with the private part of *keyA* is the same in both cases.

For TPM_ALG_ECMQV or TPM_ALG_SM2 *outZ1* will be Z and *outZ2* will be an Empty Point.

NOTE 3 An Empty Point has two Empty Buffers as coordinates meaning the minimum *size* value for *outZ2* will be four.

If the input scheme is TPM_ALG_ECDH, then *outZ1* will be Z_s and outZ2 will be Z_e. For schemes like MQV (including SM2), *outZ1* will contain the computed value and *outZ2* will be an Empty Point.

NOTE 4 The Z values returned by the TPM are a full point and not just an x-coordinate.

If a computation of either Z produces the point at infinity, then the corresponding Z value will be an Empty Point.

 Copyright © TCG 2006-2020

14.7.2 Command and Response

Table 55 — TPM2_ZGen_2Phase Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ZGen_2Phase
TPMI_DH_OBJECT	@keyA	handle of an unrestricted decryption key ECC The private key referenced by this handle is used as $d_{S,A}$ Auth Index: 1 Auth Role: USER
TPM2B_ECC_POINT	inQsB	other party's static public key ($Q_{s,B} = (X_{s,B}, Y_{s,B})$)
TPM2B_ECC_POINT	inQeB	other party's ephemeral public key ($Q_{e,B} = (X_{e,B}, Y_{e,B})$)
TPMI_ECC_KEY_EXCHANGE	inScheme	the key exchange scheme
UINT16	counter	value returned by TPM2_EC_Ephemeral()

Table 56 — TPM2_ZGen_2Phase Response

Type	Name	Description
TPM_ST	tag	
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ECC_POINT	outZ1	X and Y coordinates of the computed value (scheme dependent)
TPM2B_ECC_POINT	outZ2	X and Y coordinates of the second computed value (scheme dependent)

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

14.7.3 Detailed Actions

[[ZGen_2Phase]]

15 Symmetric Primitives

15.1 Introduction

The commands in this clause provide low-level primitives for access to the symmetric algorithms implemented in the TPM that operate on blocks of data. These include symmetric encryption and decryption as well as hash and HMAC. All of the commands in this group are stateless. That is, they have no persistent state that is retained in the TPM when the command is complete.

For hashing, HMAC, and Events that require large blocks of data with retained state, the sequence commands are provided (see clause 17).

Some of the symmetric encryption/decryption modes use an IV. When an IV is used, it may be an initiation value or a chained value from a previous stage. The chaining for each mode is:

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

Table 57 — Symmetric Chaining Process

Mode	Chaining process
TPM_ALG_CTR	The TPM will increment the entire IV provided by the caller. The next count value will be returned to the caller as *ivOut*. This can be the input value to the next encrypt or decrypt operation. *ivIn* is required to be the size of a block encrypted by the selected algorithm and key combination. If the size of *ivIn* is not correct, the TPM shall return TPM_RC_SIZE. EXAMPLE 1 AES requires that *ivIn* be 128 bits (16 octets). *ivOut* will be the size of a cipher block and not the size of the last encrypted block. NOTE *ivOut* will be the value of the counter after the last block is encrypted. EXAMPLE 2 If *ivIn* were 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00_{16} and four data blocks were encrypted, *ivOut* will have a value of 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 04_{16}. All the bits of the IV are incremented as if it were an unsigned integer.
TPM_ALG_OFB	In Output Feedback (OFB), the output of the pseudo-random function (the block encryption algorithm) is XORed with a plaintext block to produce a ciphertext block. *ivOut* will be the value that was XORed with the last plaintext block. That value can be used as the *ivIn* for a next buffer. *ivIn* is required to be the size of a block encrypted by the selected algorithm and key combination. If the size of *ivIn* is not correct, the TPM shall return TPM_RC_SIZE. *ivOut* will be the size of a cipher block and not the size of the last encrypted block.
TPM_ALG_CBC	For Cipher Block Chaining (CBC), a block of ciphertext is XORed with the next plaintext block and that block is encrypted. The encrypted block is then input to the encryption of the next block. The last ciphertext block then is used as an IV for the next buffer. Even though the last ciphertext block is evident in the encrypted data, it is also returned in *ivOut*. *ivIn* is required to be the size of a block encrypted by the selected algorithm and key combination. If the size of *ivIn* is not correct, the TPM shall return TPM_RC_SIZE. *inData* is required to be an even multiple of the block encrypted by the selected algorithm and key combination. If the size of *inData* is not correct, the TPM shall return TPM_RC_SIZE.
TPM_ALG_CFB	Similar to CBC in that the last ciphertext block is an input to the encryption of the next block. *ivOut* will be the value that was XORed with the last plaintext block. That value can be used as the *ivIn* for a next buffer. *ivIn* is required to be the size of a block encrypted by the selected algorithm and key combination. If the size of *ivIn* is not correct, the TPM shall return TPM_RC_SIZE. *ivOut* will be the size of a cipher block and not the size of the last encrypted block.
TPM_ALG_ECB	Electronic Codebook (ECB) has no chaining. Each block of plaintext is encrypted using the key. ECB does not support chaining and *ivIn* shall be the Empty Buffer. *ivOut* will be the Empty Buffer. *inData* is required to be an even multiple of the block encrypted by the selected algorithm and key combination. If the size of *inData* is not correct, the TPM shall return TPM_RC_SIZE.

 Copyright © TCG 2006-2020

15.2 TPM2_EncryptDecrypt

15.2.1 General Description

NOTE 1 This command is deprecated, and TPM2_EncryptDecrypt2() is preferred. This should be reflected in platform-specific specifications.

This command performs symmetric encryption or decryption using the symmetric key referenced by keyHandle and the selected mode.

keyHandle shall reference a symmetric cipher object (TPM_RC_KEY) with the *restricted* attribute CLEAR (TPM_RC_ATTRIBUTES).

If the *decrypt* parameter of the command is TRUE, then the *decrypt* attribute of the key is required to be SET (TPM_RC_ATTRIBUTES). If the *decrypt* parameter of the command is FALSE, then the *sign* attribute of the key is required to be SET (TPM_RC_ATTRIBUTES).

NOTE 2 A key may have both *decrypt* and *sign* SET.

If the mode of the key is not TPM_ALG_NULL, then that is the only mode that can be used with the key and the caller is required to set *mode* either to TPM_ALG_NULL or to the same mode as the key (TPM_RC_MODE). If the mode of the key is TPM_ALG_NULL, then the caller may set *mode* to any valid symmetric encryption/decryption mode but may not select TPM_ALG_NULL (TPM_RC_MODE).

If the TPM allows this command to be canceled before completion, then the TPM may produce incremental results and return TPM_RC_SUCCESS rather than TPM_RC_CANCELED. In such case, *outData* may be less than *inData*.

NOTE 3 If all the data is encrypted/decrypted, the size of outData will be the same as inData.

 Copyright © TCG 2006-2020

15.2.2 Command and Response

Table 58 — TPM2_EncryptDecrypt Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_EncryptDecrypt
TPMI_DH_OBJECT	@keyHandle	the symmetric key used for the operation Auth Index: 1 Auth Role: USER
TPMI_YES_NO	decrypt	if YES, then the operation is decryption; if NO, the operation is encryption
TPMI_ALG_CIPHER_MODE+	mode	symmetric encryption/decryption mode this field shall match the default mode of the key or be TPM_ALG_NULL.
TPM2B_IV	ivIn	an initial value as required by the algorithm
TPM2B_MAX_BUFFER	inData	the data to be encrypted/decrypted

Table 59 — TPM2_EncryptDecrypt Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_MAX_BUFFER	outData	encrypted or decrypted output
TPM2B_IV	ivOut	chaining value to use for IV in next round

 Copyright © TCG 2006-2020

15.2.3 Detailed Actions

[[EncryptDecrypt]]

 Copyright © TCG 2006-2020

15.3 TPM2_EncryptDecrypt2

15.3.1 General Description

This command is identical to TPM2_EncryptDecrypt(), except that the *inData* parameter is the first parameter. This permits *inData* to be parameter encrypted.

NOTE In platform specification updates, this command is preferred and TPM2_EncryptDecrypt() should be deprecated.

 Copyright © TCG 2006-2020

15.3.2 Comand and Response

Table 60 — TPM2_EncryptDecrypt2 Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_EncryptDecrypt2
TPMI_DH_OBJECT	@keyHandle	the symmetric key used for the operation Auth Index: 1 Auth Role: USER
TPM2B_MAX_BUFFER	inData	the data to be encrypted/decrypted
TPMI_YES_NO	decrypt	if YES, then the operation is decryption; if NO, the operation is encryption
TPMI_ALG_CIPHER_MODE+	mode	symmetric mode this field shall match the default mode of the key or be TPM_ALG_NULL.
TPM2B_IV	ivIn	an initial value as required by the algorithm

Table 61 — TPM2_EncryptDecrypt2 Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_MAX_BUFFER	outData	encrypted or decrypted output
TPM2B_IV	ivOut	chaining value to use for IV in next round

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

15.3.3 Detailed Actions

[[EncryptDecrypt2]]

15.4 TPM2_Hash

15.4.1 General Description

This command performs a hash operation on a data buffer and returns the results.

NOTE If the data buffer to be hashed is larger than will fit into the TPM's input buffer, then the sequence hash commands will need to be used.

If the results of the hash will be used in a signing operation that uses a restricted signing key, then the ticket returned by this command can indicate that the hash is safe to sign.

If the digest is not safe to sign, then the TPM will return a TPMT_TK_HASHCHECK with the hierarchy set to TPM_RH_NULL and *digest* set to the Empty Buffer.

If *hierarchy* is TPM_RH_NULL, then *digest* in the ticket will be the Empty Buffer.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

15.4.2 Command and Response

Table 62 — TPM2_Hash Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit, decrypt, or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Hash
TPM2B_MAX_BUFFER	data	data to be hashed
TPMI_ALG_HASH	hashAlg	algorithm for the hash being computed – shall not be TPM_ALG_NULL
TPMI_RH_HIERARCHY+	hierarchy	hierarchy to use for the ticket (TPM_RH_NULL allowed)

Table 63 — TPM2_Hash Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	outHash	results
TPMT_TK_HASHCHECK	validation	ticket indicating that the sequence of octets used to compute *outDigest* did not start with TPM_GENERATED_VALUE will be a NULL ticket if the digest may not be signed with a restricted key

 Copyright © TCG 2006-2020

15.4.3 Detailed Actions

[[HASH]]

 Copyright © TCG 2006-2020

15.5 TPM2_HMAC

15.5.1 General Description

This command performs an HMAC on the supplied data using the indicated hash algorithm.

NOTE 1 A TPM may implement either TPM2_HMAC() or TPM2_MAC() but not both, as they have the same command code and there is no way to distinguish them. A TPM that supports TPM2_MAC() will support any code that was written to use TPM2_HMAC(), but a TPM that supports TPM2_HMAC() will not support a MAC based on symmetric block ciphers.

The caller shall provide proper authorization for use of *handle*.

If the *sign* attribute is not SET in the key referenced by *handle* then the TPM shall return TPM_RC_KEY. If the key type is not TPM_ALG_KEYEDHASH then the TPM shall return TPM_RC_TYPE. If the key referenced by *handle* has the *restricted* attribute SET, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE 2 For symmetric signing with a restricted key, see TPM2_Sign.

If the default scheme of the key referenced by *handle* is not TPM_ALG_NULL, then the *hashAlg* parameter is required to be either the same as the key's default or TPM_ALG_NULL (TPM_RC_VALUE). If the default scheme of the key is TPM_ALG_NULL, then hashAlg is required to be a valid hash and not TPM_ALG_NULL (TPM_RC_VALUE) (see hash selection matrix in

Table 72).

NOTE 3 A key may only have both sign and decrypt SET if the key is unrestricted. When both sign and decrypt are set, there is no default scheme for the key and the hash algorithm must be specified.

 Copyright © TCG 2006-2020

15.5.2 Command and Response

Table 64 — TPM2_HMAC Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_HMAC
TPMI_DH_OBJECT	@handle	handle for the symmetric signing key providing the HMAC key Auth Index: 1 Auth Role: USER
TPM2B_MAX_BUFFER	buffer	HMAC data
TPMI_ALG_HASH+	hashAlg	algorithm to use for HMAC

Table 65 — TPM2_HMAC Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	outHMAC	the returned HMAC in a sized buffer

 Copyright © TCG 2006-2020

15.5.3 Detailed Actions

[[HMAC]]

15.6 TPM2_MAC

15.6.1 General Description

This command performs an HMAC or a block cipher MAC on the supplied data using the indicated algorithm.

NOTE 1 A TPM may implement either TPM2_HMAC() or TPM2_MAC() but not both as they have the same command code and there is no way to distinguish them. A TPM that supports TPM2_MAC() will support any code that was written to use TPM2_HMAC() but a TPM that supports TPM2_HMAC () will not support a MAC based on symmetric block ciphers.

The caller shall provide proper authorization for use of *handle*.

If the *sign* attribute is not SET in the key referenced by *handle* then the TPM shall return TPM_RC_KEY. If the key type is neither TPM_ALG_KEYEDHASH nor TPM_ALG_SYMCIPHER then the TPM shall return TPM_RC_TYPE. If the key referenced by *handle* has the *restricted* attribute SET, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE 2 For symmetric signing with a restricted key, see TPM2_Sign.

If the default scheme or mode of the key referenced by *handle* is not TPM_ALG_NULL, then the *inScheme* parameter is required to be either the same as the key's default or TPM_ALG_NULL (TPM_RC_VALUE).

If the default scheme of an HMAC key is TPM_ALG_NULL, then *inScheme* is required to be a valid hash and not TPM_ALG_NULL (TPM_RC_VALUE) (see algorithm selection matrix in

Table 75).

If the default mode of a symmetric cipher key is TPM_ALG_NULL, then *inScheme* is required to be a valid block cipher mode for authentication and not TPM_ALG_NULL (TPM_RC_VALUE)

NOTE 3 A key may only have both sign and decrypt SET if the key is unrestricted. When both sign and decrypt are set, there is no default scheme for the key and *inScheme* may not be TPM_ALG_NULL.

NOTE 4 TPM2_MAC() was added in revision 01.43.

 Copyright © TCG 2006-2020

15.6.2 Command and Response

Table 66 — TPM2_MAC Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_MAC
TPMI_DH_OBJECT	@handle	handle for the symmetric signing key providing the MAC key Auth Index: 1 Auth Role: USER
TPM2B_MAX_BUFFER	buffer	MAC data
TPMI_ALG_MAC_SCHEME+	inScheme	algorithm to use for MAC

Table 67 — TPM2_MAC Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	outMAC	the returned MAC in a sized buffer

 Copyright © TCG 2006-2020

15.6.3 Detailed Actions

[[MAC]]

16 Random Number Generator

16.1 TPM2_GetRandom

16.1.1 General Description

This command returns the next *bytesRequested* octets from the random number generator (RNG).

NOTE 1 It is recommended that a TPM implement the RNG in a manner that would allow it to return RNG octets such that, as long as the value of *bytesRequested* is not greater than the maximum digest size, the frequency of *bytesRequested* being more than the number of octets available is an infrequent occurrence.

If *bytesRequested* is more than will fit into a TPM2B_DIGEST on the TPM, no error is returned but the TPM will only return as much data as will fit into a TPM2B_DIGEST buffer for the TPM.

NOTE 2 TPM2B_DIGEST is large enough to hold the largest digest that may be produced by the TPM. Because that digest size changes according to the implemented hashes, the maximum amount of data returned by this command is TPM implementation-dependent.

16.1.2 Command and Response

Table 68 — TPM2_GetRandom Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_GetRandom
UINT16	bytesRequested	number of octets to return

Table 69 — TPM2_GetRandom Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	randomBytes	the random octets

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

16.1.3 Detailed Actions

[[GetRandom]]

16.2 TPM2_StirRandom

16.2.1 General Description

This command is used to add "additional information" to the RNG state.

NOTE The "additional information" is as defined in SP800-90A.

The *inData* parameter may not be larger than 128 octets.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

16.2.2 Command and Response

Table 70 — TPM2_StirRandom Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_StirRandom {NV}
TPM2B_SENSITIVE_DATA	inData	additional information

Table 71 — TPM2_StirRandom Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

16.2.3 Detailed Actions

[[StirRandom]]

17 Hash/HMAC/Event Sequences

17.1 Introduction

All of the commands in this group are to support sequences for which an intermediate state must be maintained. For a description of sequences, see “Hash, MAC, and Event Sequences” in TPM 2.0 Part 1.

A TPM may implement either TPM2_HMAC_Start() or TPM2_MAC_Start() but not both as they have the same command code and there is no way to distinguish them. A TPM that supports TPM2_MAC_Start() will support any code that was written to use TPM2_HMAC_Start() but a TPM that supports TPM2_HMAC_Start() will not support a MAC based on symmetric block ciphers.

17.2 TPM2_HMAC_Start

17.2.1 General Description

This command starts an HMAC sequence. The TPM will create and initialize an HMAC sequence structure, assign a handle to the sequence, and set the *authValue* of the sequence object to the value in *auth*.

NOTE 1 The structure of a sequence object is vendor-dependent.

The caller shall provide proper authorization for use of *handle*.

If the *sign* attribute is not SET in the key referenced by *handle* then the TPM shall return TPM_RC_KEY. If the key type is not TPM_ALG_KEYEDHASH then the TPM shall return TPM_RC_TYPE. If the key referenced by *handle* has the *restricted* attribute SET, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE 2 For symmetric signing with a restricted key, see TPM2_Sign.

If the default scheme of the key referenced by *handle* is not TPM_ALG_NULL, then the *hashAlg* parameter is required to be either the same as the key’s default or TPM_ALG_NULL (TPM_RC_VALUE). If the default scheme of the key is TPM_ALG_NULL, then hashAlg is required to be a valid hash and not TPM_ALG_NULL (TPM_RC_VALUE).

Table 72 — Hash Selection Matrix

handle→restricted **(key's restricted attribute)**	***handle→scheme*** **(hash algorithm from key's *scheme*)**	***hashAlg***	**hash used**
CLEAR (unrestricted)	TPM_ALG_NULL[(1)]	TPM_ALG_NULL	error[(1)] (TPM_RC_VALUE)
CLEAR	TPM_ALG_NULL	valid hash	*hashAlg*
CLEAR	valid hash	TPM_ALG_NULL or same as *handle→scheme*	*handle→scheme*
CLEAR	valid hash	valid hash	error (TPM_RC_VALUE) if *hashAlg* != *handle->scheme*
SET (restricted)	don't care	don't care	TPM_RC_ATTRIBUTES
NOTES: 1) A hash algorithm is required for the HMAC.			

NOTE 1 A TPM may implement either TPM2_HMAC_Start() or TPM2_MAC_Start() but not both, as they have the same command code and there is no way to distinguish them. A TPM that supports TPM2_MAC_Start() will support any code that was written to use TPM2_HMAC_Start(), but a TPM that supports TPM2_HMAC_Start() will not support a MAC based on symmetric block ciphers.

 Copyright © TCG 2006-2020

17.2.2 Command and Response

Table 73 — TPM2_HMAC_Start Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_HMAC_Start
TPMI_DH_OBJECT	@handle	handle of an HMAC key Auth Index: 1 Auth Role: USER
TPM2B_AUTH	auth	authorization value for subsequent use of the sequence
TPMI_ALG_HASH+	hashAlg	the hash algorithm to use for the HMAC

Table 74 — TPM2_HMAC_Start Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_DH_OBJECT	sequenceHandle	a handle to reference the sequence

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

17.2.3 Detailed Actions

[[HMAC_Start]]

17.3 TPM2_MAC_Start

17.3.1 General Description

This command starts a MAC sequence. The TPM will create and initialize a MAC sequence structure, assign a handle to the sequence, and set the *authValue* of the sequence object to the value in *auth*.

NOTE 1 The structure of a sequence object is vendor-dependent.

The caller shall provide proper authorization for use of *handle*.

If the *sign* attribute is not SET in the key referenced by *handle* then the TPM shall return TPM_RC_KEY. If the key type is not TPM_ALG_KEYEDHASH or TPM_ALG_SYMCIPHER then the TPM shall return TPM_RC_TYPE. If the key referenced by *handle* has the *restricted* attribute SET, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE 2 For symmetric signing with a restricted key, see TPM2_Sign.

If the default scheme of the key referenced by *handle* is not TPM_ALG_NULL, then the *inScheme* parameter is required to be either the same as the key's default or TPM_ALG_NULL (TPM_RC_VALUE). If the default scheme of the key is TPM_ALG_NULL, then *inScheme*is required to be a valid hash or symmetric MAC scheme and not TPM_ALG_NULL (TPM_RC_VALUE).

Table 75 — Algorithm Selection Matrix

handle→restricted **(key's restricted attribute)**	***handle→scheme*** **(algorithm from key's *scheme*)**	***inScheme***	**algorithm used**
CLEAR (unrestricted)	TPM_ALG_NULL(1)	TPM_ALG_NULL	error(1) (TPM_RC_VALUE)
CLEAR	TPM_ALG_NULL	valid hash or symmetric MAC	*inScheme*
CLEAR	not TPM_ALG_NULL	TPM_ALG_NULL or same as *handle→scheme*	*handle→scheme*
CLEAR	not TPM_ALG_NULL	not TPM_AGL_NULL	error (TPM_RC_VALUE) if*inScheme*!= *handle->scheme*
SET (restricted)	don't care	don't care	TPM_RC_ATTRIBUTES
NOTES: 1) A hash algorithm is required for the HMAC. 2) hashAlg shall be TPM_ALG_NULL for handle referencing a CMAC key.			

NOTE 3 For a TPM_ALG_SYMCIPHER key, the symmetric block cipher algorithm is part of the key definition.

NOTE 4 TPM2_MAC_Start() was added in revision 01.43.

 Copyright © TCG 2006-2020

17.3.2 Command and Response

Table 76 — TPM2_MAC_Start Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_MAC_Start
TPMI_DH_OBJECT	@handle	handle of a MAC key Auth Index: 1 Auth Role: USER
TPM2B_AUTH	auth	authorization value for subsequent use of the sequence
TPMI_ALG_MAC_SCHEME+	inScheme	the algorithm to use for the MAC

Table 77 — TPM2_MAC_Start Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_DH_OBJECT	sequenceHandle	a handle to reference the sequence

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

17.3.3 Detailed Actions

[[MAC_Start]]

17.4 TPM2_HashSequenceStart

17.4.1 General Description

This command starts a hash or an Event Sequence. If *hashAlg* is an implemented hash, then a hash sequence is started. If *hashAlg* is TPM_ALG_NULL, then an Event Sequence is started. If *hashAlg* is neither an implemented algorithm nor TPM_ALG_NULL, then the TPM shall return TPM_RC_HASH.

Depending on *hashAlg*, the TPM will create and initialize a Hash Sequence context or an Event Sequence context. Additionally, it will assign a handle to the context and set the *authValue* of the context to the value in *auth*. A sequence context for an Event (*hashAlg* = TPM_ALG_NULL) contains a hash context for each of the PCR banks implemented on the TPM.

 Copyright © TCG 2006-2020

17.4.2 Command and Response

Table 78 — TPM2_HashSequenceStart Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_HashSequenceStart
TPM2B_AUTH	auth	authorization value for subsequent use of the sequence
TPMI_ALG_HASH+	hashAlg	the hash algorithm to use for the hash sequence An Event Sequence starts if this is TPM_ALG_NULL.

Table 79 — TPM2_HashSequenceStart Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_DH_OBJECT	sequenceHandle	a handle to reference the sequence

 Copyright © TCG 2006-2020

17.4.3 Detailed Actions

[[HashSequenceStart]]

 Copyright © TCG 2006-2020

17.5 TPM2_SequenceUpdate

17.5.1 General Description

This command is used to add data to a hash or HMAC sequence. The amount of data in buffer may be any size up to the limits of the TPM.

NOTE 1 In all TPM, a *buffer* size of 1,024 octets is allowed.

Proper authorization for the sequence object associated with *sequenceHandle* is required. If an authorization or audit of this command requires computation of a *cpHash* and an *rpHash*, the Name associated with *sequenceHandle* will be the Empty Buffer.

If the command does not return TPM_RC_SUCCESS, the state of the sequence is unmodified.

If the sequence is intended to produce a digest that will be signed by a restricted signing key, then the first block of data shall contain sizeof(TPM_GENERATED) octets and the first octets shall not be TPM_GENERATED_VALUE.

NOTE 2 This requirement allows the TPM to validate that the first block is safe to sign without having to accumulate octets over multiple calls.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

17.5.2 Command and Response

Table 80 — TPM2_SequenceUpdate Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_SequenceUpdate
TPMI_DH_OBJECT	@sequenceHandle	handle for the sequence object Auth Index: 1 Auth Role: USER
TPM2B_MAX_BUFFER	buffer	data to be added to hash

Table 81 — TPM2_SequenceUpdate Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

17.5.3 Detailed Actions

[[SequenceUpdate]]

 Copyright © TCG 2006-2020

17.6 TPM2_SequenceComplete

17.6.1 General Description

This command adds the last part of data, if any, to a hash/HMAC sequence and returns the result.

NOTE 1 This command is not used to complete an Event Sequence. TPM2_EventSequenceComplete() is used for that purpose.

For a hash sequence, if the results of the hash will be used in a signing operation that uses a restricted signing key, then the ticket returned by this command can indicate that the hash is safe to sign.

If the *digest* is not safe to sign, then *validation* will be a TPMT_TK_HASHCHECK with the hierarchy set to TPM_RH_NULL and *digest* set to the Empty Buffer.

If *hierarchy* is TPM_RH_NULL, then *digest* in the ticket will be the Empty Buffer.

NOTE 2 Regardless of the contents of the first octets of the hashed message, if the first buffer sent to the TPM had fewer than sizeof(TPM_GENERATED) octets, then the TPM will operate as if *digest* is not safe to sign.

NOTE 3 The ticket is only required for a signing operation that uses a restricted signing key. It is always returned, but can be ignored if not needed.

If *sequenceHandle* references an Event Sequence, then the TPM shall return TPM_RC_MODE.

Proper authorization for the sequence object associated with *sequenceHandle* is required. If an authorization or audit of this command requires computation of a *cpHash* and an *rpHash*, the Name associated with *sequenceHandle* will be the Empty Buffer.

If this command completes successfully, the *sequenceHandle* object will be flushed.

 Copyright © TCG 2006-2020

17.6.2 Command and Response

Table 82 — TPM2_SequenceComplete Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_SequenceComplete {F}
TPMI_DH_OBJECT	@sequenceHandle	authorization for the sequence Auth Index: 1 Auth Role: USER
TPM2B_MAX_BUFFER	buffer	data to be added to the hash/HMAC
TPMI_RH_HIERARCHY+	hierarchy	hierarchy of the ticket for a hash

Table 83 — TPM2_SequenceComplete Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	result	the returned HMAC or digest in a sized buffer
TPMT_TK_HASHCHECK	validation	ticket indicating that the sequence of octets used to compute *outDigest* did not start with TPM_GENERATED_VALUE This is a NULL Ticket when the sequence is HMAC.

 Copyright © TCG 2006-2020

17.6.3 Detailed Actions

[[SequenceComplete]]

17.7 TPM2_EventSequenceComplete

17.7.1 General Description

This command adds the last part of data, if any, to an Event Sequence and returns the result in a digest list. If *pcrHandle* references a PCR and not TPM_RH_NULL, then the returned digest list is processed in the same manner as the digest list input parameter to TPM2_PCR_Extend(). That is, if a bank contains a PCR associated with *pcrHandle,* it is extended with the associated digest value from the list.

If *sequenceHandle* references a hash or HMAC sequence, the TPM shall return TPM_RC_MODE.

Proper authorization for the sequence object associated with *sequenceHandle* is required. If an authorization or audit of this command requires computation of a *cpHash* and an *rpHash*, the Name associated with *sequenceHandle* will be the Empty Buffer.

If this command completes successfully, the *sequenceHandle* object will be flushed.

NOTE: Unlike TPM2_PCR_Event(), a digest is always returned for each implemented hash algorithm. There is no option to only return digests for which *pcrHandle* is allocated.

 Copyright © TCG 2006-2020

17.7.2 Command and Response

Table 84 — TPM2_EventSequenceComplete Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_EventSequenceComplete {NV F}
TPMI_DH_PCR+	@pcrHandle	PCR to be extended with the Event data Auth Index: 1 Auth Role: USER
TPMI_DH_OBJECT	@sequenceHandle	authorization for the sequence Auth Index: 2 Auth Role: USER
TPM2B_MAX_BUFFER	buffer	data to be added to the Event

Table 85 — TPM2_EventSequenceComplete Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPML_DIGEST_VALUES	results	list of digests computed for the PCR

 Copyright © TCG 2006-2020

17.7.3 Detailed Actions

[[EventSequenceComplete]]

 Copyright © TCG 2006-2020

18 Attestation Commands

18.1 Introduction

The attestation commands cause the TPM to sign an internally generated data structure. The contents of the data structure vary according to the command.

If the *sign* attribute is not SET in the key referenced by *signHandle* then the TPM shall return TPM_RC_KEY.

All signing commands include a parameter (typically *inScheme*) for the caller to specify a scheme to be used for the signing operation. This scheme will be applied only if the scheme of the key is TPM_ALG_NULL or the key handle is TPM_RH_NULL. If the scheme for *signHandle* is not TPM_ALG_NULL, then *inScheme.scheme* shall be TPM_ALG_NULL or the same as *scheme* in the public area of the key. If the scheme for *signHandle* is TPM_ALG_NULL or the key handle is TPM_RH_NULL, then *inScheme* will be used for the signing operation and may not be TPM_ALG_NULL. The TPM shall return TPM_RC_SCHEME to indicate that the scheme is not appropriate.

For a signing key that is not restricted, the caller may specify the scheme to be used as long as the scheme is compatible with the family of the key (for example, TPM_ALG_RSAPSS cannot be selected for an ECC key). If the caller sets *scheme* to TPM_ALG_NULL, then the default scheme of the key is used. For a restricted signing key, the key's scheme cannot be TPM_ALG_NULL and cannot be overridden.

If the handle for the signing key (*signHandle*) is TPM_RH_NULL, then all of the actions of the command are performed and the attestation block is “signed” with the NULL Signature.

NOTE 1 This mechanism is provided so that additional commands are not required to access the data that might be in an attestation structure.

NOTE 2 When *signHandle* is TPM_RH_NULL, *scheme* is still required to be a valid signing scheme (may be TPM_ALG_NULL), but the scheme will have no effect on the format of the signature. It will always be the NULL Signature.

TPM2_NV_Certify() is an attestation command that is documented in 31.16. The remaining attestation commands are collected in the remainder of this clause.

Each of the attestation structures contains a TPMS_CLOCK_INFO structure and a firmware version number. These values may be considered privacy-sensitive, because they would aid in the correlation of attestations by different keys. To provide improved privacy, the *resetCount*, *restartCount*, and *firmwareVersion* numbers are obfuscated when the signing key is not in the Endorsement or Platform hierarchies.

The obfuscation value is computed by:

$$obfuscation \coloneqq \mathbf{KDFa}(signHandle \rightarrow nameAlg, shProof, \text{“OBFUSCATE”}, signHandle \rightarrow QN, 0, 128) \quad (3)$$

Of the returned 128 bits, 64 bits are added to the *versionNumber* field of the attestation structure; 32 bits are added to the *clockInfo.resetCount* and 32 bits are added to the *clockInfo.restartCount*. The order in which the bits are added is implementation-dependent.

NOTE 3 The obfuscation value for each signing key will be unique to that key in a specific location. That is, each version of a duplicated signing key will have a different obfuscation value.

When the signing key is TPM_RH_NULL, the data structure is produced but not signed; and the values in the signed data structure are obfuscated. When computing the obfuscation value for TPM_RH_NULL, the hash used for context integrity is used.

NOTE 4 The QN for TPM_RH_NULL is TPM_RH_NULL.

 Copyright © TCG 2006-2020

If the signing scheme of *signHandle* is an anonymous scheme, then the attestation blocks will not contain the Qualified Name of the *signHandle*.

Each of the attestation structures allows the caller to provide some qualifying data (*qualifyingData*). For most signing schemes, this value will be placed in the TPMS_ATTEST.*extraData* parameter that is then hashed and signed. However, for some schemes such as ECDAA, the *qualifyingData* is used in a different manner (for details, see “ECDAA” in TPM 2.0 Part 1).

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

18.2 TPM2_Certify

18.2.1 General Description

The purpose of this command is to prove that an object with a specific Name is loaded in the TPM. By certifying that the object is loaded, the TPM warrants that a public area with a given Name is self-consistent and associated with a valid sensitive area. If a relying party has a public area that has the same Name as a Name certified with this command, then the values in that public area are correct.

NOTE 1 See 18.1 for description of how the signing scheme is selected.

Authorization for *objectHandle* requires ADMIN role authorization. If performed with a policy session, the session shall have a policySession→*commandCode* set to TPM_CC_Certify. This indicates that the policy that is being used is a policy that is for certification, and not a policy that would approve another use. That is, authority to use an object does not grant authority to certify the object.

The object may be any object that is loaded with TPM2_Load() or TPM2_CreatePrimary(). An object that only has its public area loaded cannot be certified.

NOTE 2 The restriction occurs because the Name is used to identify the object being certified. If the TPM has not validated that the public area is associated with a matched sensitive area, then the public area may not represent a valid object and cannot be certified.

The certification includes the Name and Qualified Name of the certified object as well as the Name and the Qualified Name of the certifying object.

NOTE 3 If *signHandle* is TPM_RH_NULL, the TPMS_ATTEST structure is returned and *signature* is a NULL Signature.

 Copyright © TCG 2006-2020

18.2.2 Command and Response

Table 86 — TPM2_Certify Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Certify
TPMI_DH_OBJECT	@objectHandle	handle of the object to be certified Auth Index: 1 Auth Role: ADMIN
TPMI_DH_OBJECT+	@signHandle	handle of the key used to sign the attestation structure Auth Index: 2 Auth Role: USER
TPM2B_DATA	qualifyingData	user provided qualifying data
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL

Table 87 — TPM2_Certify Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPM2B_ATTEST	certifyInfo	the structure that was signed
TPMT_SIGNATURE	signature	the asymmetric signature over *certifyInfo* using the key referenced by *signHandle*

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

18.2.3 Detailed Actions

[[Certify]]

18.3 TPM2_CertifyCreation

18.3.1 General Description

This command is used to prove the association between an object and its creation data. The TPM will validate that the ticket was produced by the TPM and that the ticket validates the association between a loaded public area and the provided hash of the creation data (*creationHash*).

NOTE 1 See 18.1 for description of how the signing scheme is selected.

The TPM will create a test ticket using the Name associated with *objectHandle* and *creationHash* as:

$$\mathbf{HMAC}(proof, (\text{TPM_ST_CREATION} \mid\mid objectHandle{\rightarrow}Name \mid\mid creationHash)) \tag{4}$$

This ticket is then compared to creation ticket. If the tickets are not the same, the TPM shall return TPM_RC_TICKET.

If the ticket is valid, then the TPM will create a TPMS_ATTEST structure and place *creationHash* of the command in the *creationHash* field of the structure. The Name associated with *objectHandle* will be included in the attestation data that is then signed using the key associated with *signHandle*.

NOTE 2 If *signHandle* is TPM_RH_NULL, the TPMS_ATTEST structure is returned and *signature* is a NULL Signature.

objectHandle may be any object that is loaded with TPM2_Load() or TPM2_CreatePrimary().

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

18.3.2 Command and Response

Table 88 — TPM2_CertifyCreation Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_CertifyCreation
TPMI_DH_OBJECT+	@signHandle	handle of the key that will sign the attestation block Auth Index: 1 Auth Role: USER
TPMI_DH_OBJECT	objectHandle	the object associated with the creation data Auth Index: None
TPM2B_DATA	qualifyingData	user-provided qualifying data
TPM2B_DIGEST	creationHash	hash of the creation data produced by TPM2_Create() or TPM2_CreatePrimary()
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL
TPMT_TK_CREATION	creationTicket	ticket produced by TPM2_Create() or TPM2_CreatePrimary()

Table 89 — TPM2_CertifyCreation Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ATTEST	certifyInfo	the structure that was signed
TPMT_SIGNATURE	signature	the signature over *certifyInfo*

 Copyright © TCG 2006-2020

18.3.3 Detailed Actions

[[CertifyCreation]]

18.4 TPM2_Quote

18.4.1 General Description

This command is used to quote PCR values.

The TPM will hash the list of PCR selected by *PCRselect* using the hash algorithm in the selected signing scheme. If the selected signing scheme or the scheme hash algorithm is TPM_ALG_NULL, then the TPM shall return TPM_RC_SCHEME.

NOTE 1 See 18.1 for description of how the signing scheme is selected.

The digest is computed as the hash of the concatenation of all of the digest values of the selected PCR.

The concatenation of PCR is described in TPM 2.0 Part 1, *Selecting Multiple PCR.*

NOTE 2 If *signHandle* is TPM_RH_NULL, the TPMS_ATTEST structure is returned and *signature* is a NULL Signature.

NOTE 3 A TPM may optionally return TPM_RC_SCHEME if *signHandle* is TPM_RH_NULL.

NOTE 4 Unlike TPM 1.2, TPM2_Quote does not return the PCR values. See Part 1, “Attesting to PCR” for a discussion of this issue.

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

18.4.2 Command and Response

Table 90 — TPM2_Quote Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Quote
TPMI_DH_OBJECT+	@signHandle	handle of key that will perform signature Auth Index: 1 Auth Role: USER
TPM2B_DATA	qualifyingData	data supplied by the caller
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL
TPML_PCR_SELECTION	PCRselect	PCR set to quote

Table 91 — TPM2_Quote Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ATTEST	quoted	the quoted information
TPMT_SIGNATURE	signature	the signature over *quoted*

 Copyright © TCG 2006-2020

18.4.3 Detailed Actions

[[Quote]]

18.5 TPM2_GetSessionAuditDigest

18.5.1 General Description

This command returns a digital signature of the audit session digest.

NOTE 1 See 18.1 for description of how the signing scheme is selected.

If *sessionHandle* is not an audit session, the TPM shall return TPM_RC_TYPE.

NOTE 2 A session does not become an audit session until the successful completion of the command in which the session is first used as an audit session.

This command requires authorization from the privacy administrator of the TPM (expressed with Endorsement Authorization) as well as authorization to use the key associated with *signHandle*.

If this command is audited, then the audit digest that is signed will not include the digest of this command because the audit digest is only updated when the command completes successfully.

This command does not cause the audit session to be closed and does not reset the digest value.

NOTE 3 If sessionHandle is used as an audit session for this command, the command is audited in the same manner as any other command.

NOTE 4 If *signHandle* is TPM_RH_NULL, the TPMS_ATTEST structure is returned and *signature* is a NULL Signature.

 Copyright © TCG 2006-2020

18.5.2 Command and Response

Table 92 — TPM2_GetSessionAuditDigest Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_GetSessionAuditDigest
TPMI_RH_ENDORSEMENT	@privacyAdminHandle	handle of the privacy administrator (TPM_RH_ENDORSEMENT) Auth Index: 1 Auth Role: USER
TPMI_DH_OBJECT+	@signHandle	handle of the signing key Auth Index: 2 Auth Role: USER
TPMI_SH_HMAC	sessionHandle	handle of the audit session Auth Index: None
TPM2B_DATA	qualifyingData	user-provided qualifying data – may be zero-length
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL

Table 93 — TPM2_GetSessionAuditDigest Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ATTEST	auditInfo	the audit information that was signed
TPMT_SIGNATURE	signature	the signature over *auditInfo*

 Copyright © TCG 2006-2020

18.5.3 Detailed Actions

[[GetSessionAuditDigest]]

18.6 TPM2_GetCommandAuditDigest

18.6.1 General Description

This command returns the current value of the command audit digest, a digest of the commands being audited, and the audit hash algorithm. These values are placed in an attestation structure and signed with the key referenced by *signHandle*.

NOTE 1 See 18.1 for description of how the signing scheme is selected.

When this command completes successfully, and *signHandle* is not TPM_RH_NULL, the audit digest is cleared. If signHandle is TPM_RH_NULL, *signature* is the Empty Buffer and the audit digest is not cleared.

NOTE 2 The way that the TPM tracks that the digest is clear is vendor-dependent. The reference implementation resets the size of the digest to zero.

If this command is being audited, then the signed digest produced by the command will not include the command. At the end of this command, the audit digest will be extended with *cpHash* and the *rpHash* of the command, which would change the command audit digest signed by the next invocation of this command.

This command requires authorization from the privacy administrator of the TPM (expressed with Endorsement Authorization) as well as authorization to use the key associated with *signHandle*.

 Copyright © TCG 2006-2020

18.6.2 Command and Response

Table 94 — TPM2_GetCommandAuditDigest Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_GetCommandAuditDigest {NV}
TPMI_RH_ENDORSEMENT	@privacyHandle	handle of the privacy administrator (TPM_RH_ENDORSEMENT) Auth Index: 1 Auth Role: USER
TPMI_DH_OBJECT+	@signHandle	the handle of the signing key Auth Index: 2 Auth Role: USER
TPM2B_DATA	qualifyingData	other data to associate with this audit digest
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL

Table 95 — TPM2_GetCommandAuditDigest Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ATTEST	auditInfo	the auditInfo that was signed
TPMT_SIGNATURE	signature	the signature over *auditInfo*

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

18.6.3 Detailed Actions

[[GetCommandAuditDigest]]

18.7 TPM2_GetTime

18.7.1 General Description

This command returns the current values of *Time* and *Clock*.

NOTE 1 See 18.1 for description of how the signing scheme is selected.

The values of *Clock*, *resetCount* and *restartCount* appear in two places in *timeInfo*: once in TPMS_ATTEST.*clockInfo* and again in TPMS_ATTEST.*attested.time.clockInfo.* The firmware version number also appears in two places (TPMS_ATTEST.*firmwareVersion* and TPMS_ATTEST.*attested.time.firmwareVersion*). If *signHandle* is in the endorsement or platform hierarchies, both copies of the data will be the same. However, if *signHandle* is in the storage hierarchy or is TPM_RH_NULL, the values in TPMS_ATTEST.*clockInfo* and TPMS_ATTEST.*firmwareVersion* are obfuscated but the values in TPMS_ATTEST.*attested.time* are not.

NOTE 2 The purpose of this duplication is to allow an entity who is trusted by the privacy Administrator to correlate the obfuscated values with the clear-text values. This command requires Endorsement Authorization.

NOTE 3 If *signHandle* is TPM_RH_NULL, the TPMS_ATTEST structure is returned and *signature* is a NULL Signature.

 Copyright © TCG 2006-2020

18.7.2 Command and Response

Table 96 — TPM2_GetTime Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_GetTime
TPMI_RH_ENDORSEMENT	@privacyAdminHandle	handle of the privacy administrator (TPM_RH_ENDORSEMENT) Auth Index: 1 Auth Role: USER
TPMI_DH_OBJECT+	@signHandle	the *keyHandle* identifier of a loaded key that can perform digital signatures Auth Index: 2 Auth Role: USER
TPM2B_DATA	qualifyingData	data to tick stamp
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL

Table 97 — TPM2_GetTime Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPM2B_ATTEST	timeInfo	standard TPM-generated attestation block
TPMT_SIGNATURE	signature	the signature over *timeInfo*

 Copyright © TCG 2006-2020

18.7.3 Detailed Actions

[[GetTime]]

18.8 TPM2_CertifyX509

18.8.1 General Description

The purpose of this command is to generate an X.509 certificate that proves an object with a specific public key and attributes is loaded in the TPM. In contrast to TPM2_Certify, which uses a TCG-defined data structure to convey attestation information, TPM2_CertifyX509 encodes the attestation information in a DER-encoded X.509 certificate that is compliant with RFC5280 *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.*

As described in RFC, an X.509 certificate contains a collection of data that is hashed and signed. The full signature is the combination of the *to be signed* (TBS) data, a description of the signature algorithm, and the signature over the TBS data. The elements of the TBS data structure are DER-encoded values. They are:

1) Version [0] – integer value of 2 indicating version 3
2) Certificate Serial Number – integer value
3) Signature Algorithm Identifier – values (usually a collection of OIDs) identifying the algorithm used for the signature
4) Issuer Name – X.501 type *Name* to identify the entity that has authorized the use of *signHandle* to create the certificate.
5) Validity – two time values indicating the period during which the certificate is valid
6) Subject Name – X.501 type *Name* that identifies the entity that authorized the use of *objectHandle*
7) Subject Public Key Info – the public key associated with *objectHandle*,
8) Extensions [3] – a set of values that “provide methods for associating additional attributes with users or public keys and for managing relationships between CAs.”

NOTE 1: The numbers in square brackets (e.g., [0]) indicate application-specific tag values that are used to identify the type of the field.

NOTE 2: RFC 5280 describes two fields (issuerUniqueID and subjectUniqueID) but goes on to say: “CAs conforming to this profile MUST NOT generate certificates with unique identifiers.” The TPM does not allow them to be present.

The caller provides a partial certificate (*partialCertificate*) parameter that contains four or five of the elements enumerated above in a DER encoded SEQUENCE. They are:

1) Signature Algorithm Identifier (optional)
2) Issuer (mandatory)
3) Validity (mandatory)
4) Subject Name (mandatory)
5) Extensions (mandatory)

The fields are required to be in the order in which they are listed above.

NOTE 3: The TPM determines if the Signature Algorithm Identifier element is present by counting the elements.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

The optional Signature Algorithm Identifier may be provided by the caller. If it is not present, the TPM will generate the value based on the selected signing scheme. If the caller provides this value, then the TPM will use it in the completed TBS. The TPM will not validate that the provided values are compatible with the signing scheme. If the caller does not provide this field and the TPM does not have OID values for the signing scheme, then the TPM will return an error (TPM_RC_SCHEME).

NOTE 4: The TPM may implement signing schemes for which OIDs are not defined at the time the TPM was manufactured. Those schemes may still be used if the caller can provide the Signature Algorithm Identifier.

The Extensions element is required to contain a Key Usage extension. The TPM will extract the Key Usage values and verify that the attributes of *objectHandle* are consistent with the selected values (TPM_RC_ATTRIBUTES)(See Part 2, *TPMA_X509_KEY_USAGE*).

The Extensions element may contain a TPMA_OBJECT extension. If present, the TPM will extract the value and verify that the extension value exactly matches the TPMA_OBJECT of *objectKey* (TPM_RC_ATTRIBUTES). The element uses the TCG OID tcg-tpmaObject, 2.23.133.10.1.1.1. It is a SEQUENCE containing that OID and an OCTET STRING encapsulating a 4-byte BIT STRING holding the big endian TPMA_OBJECT.

signHandle is required to have the *sign* attribute SET (TPM_RC_KEY).

NOTE 5: See 18.1 for description of how the signing scheme is selected.

Authorization for *objectHandle* requires ADMIN role authorization. If performed with a policy session, the session shall have a policySession→*commandCode* set to TPM_CC_CertifyX509. This indicates that the policy that is being used is a policy that is for certification, and not a policy that would approve another use. That is, authority to use an object does not grant authority to certify the object.

If *objectHandle* does not have a sensitive area loaded, the TPM will return an error (TPM_RC_AUTH_UNAVAILABLE).

NOTE 6: The command requires that authorization be provided for use of *objectHandle.* An object that only has its *publicArea* loaded does not have an authorization value and the *authPolicy* has no meaning as the sensitive area is not present.

The TPM will create the Version, the Certificate Serial Number, the Subject Public Key Info, and, if not provided by the caller, the Signature Algorithm Identifier. These TPM-created values will be combined with the provided values to make a full TBSCerfificate structure (See RFC 5280, clause 4.1). The TPM will then sign the certificate using the selected signing scheme.

The TPM-created values will be returned in *addedToCertificate.* If the TPM creates the Signature Algorithm Identifier, it will be in *addedToCertificate* before the Subject Public Key Info. The TPM returns *tbsDigest* as a debugging aid.

NOTE 7: These returned fields allow the caller to unambiguously create a full RFC5280-defined TBSCertificate.

NOTE 8: This command was added in revision 01.53.

18.8.2 Command and Response

Table 98 — TPM2_CertifyX509 Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_CertifyX509
TPMI_DH_OBJECT	@objectHandle	handle of the object to be certified Auth Index: 1 Auth Role: ADMIN
TPMI_DH_OBJECT+	@signHandle	handle of the key used to sign the attestation structure Auth Index: 2 Auth Role: USER
TPM2B_DATA	reserved	shall be an Empty Buffer
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL
TPM2B_MAX_BUFFER	partialCertificate	a DER encoded partial certificate

Table 99 — TPM2_CertifyX509 Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPM2B_MAX_BUFFER	addedToCertificate	a DER encoded SEQUENCE containing the DER encoded fields added to partialCertificate to make it a complete RFC5280 TBSCertificate.
TPM2B_DIGEST	tbsDigest	the digest that was signed
TPMT_SIGNATURE	signature	The signature over *tbsDigest*

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

18.8.3 Detailed Actions

[[certifyX509]]

19 Ephemeral EC Keys

19.1 Introduction

The TPM generates keys that have different lifetimes. TPM keys in a hierarchy can be persistent for as long as the seed of the hierarchy is unchanged and these keys may be used multiple times. Other TPM-generated keys are only useful for a single operation. Some of these single-use keys are used in the command in which they are created. Examples of this use are TPM2_Duplicate() where an ephemeral key is created for a single pass key exchange with another TPM. However, there are other cases, such as anonymous attestation, where the protocol requires two passes where the public part of the ephemeral key is used outside of the TPM before the final command "consumes" the ephemeral key.

For these uses, TPM2_Commit() or TPM2_EC_Ephemeral() may be used to have the TPM create an ephemeral EC key and return the public part of the key for external use. Then in a subsequent command, the caller provides a reference to the ephemeral key so that the TPM can retrieve or recreate the associated private key.

When an ephemeral EC key is created, it is assigned a number and that number is returned to the caller as the identifier for the key. This number is not a handle. A handle is assigned to a key that may be context saved but these ephemeral EC keys may not be saved and do not have a full key context. When a subsequent command uses the ephemeral key, the caller provides the number of the ephemeral key. The TPM uses that number to either look up or recompute the associated private key. After the key is used, the TPM records the fact that the key has been used so that it cannot be used again.

As mentioned, the TPM can keep each assigned private ephemeral key in memory until it is used. However, this could consume a large amount of memory. To limit the memory size, the TPM is allowed to restrict the number of pending private keys – keys that have been allocated but not used.

NOTE The minimum number of ephemeral keys is determined by a platform specific specification

To further reduce the memory requirements for the ephemeral private keys, the TPM is allowed to use pseudo-random values for the ephemeral keys. Instead of keeping the full value of the key in memory, the TPM can use a counter as input to a KDF. Incrementing the counter will cause the TPM to generate a new pseudo-random value.

Using the counter to generate pseudo-random private ephemeral keys greatly simplifies tracking of key usage. When a counter value is used to create a key, a bit in an array may be set to indicate that the key use is pending. When the ephemeral key is consumed, the bit is cleared. This prevents the key from being used more than once.

Since the TPM is allowed to restrict the number of pending ephemeral keys, the array size can be limited. For example, a 128 bit array would allow 128 keys to be "pending".

The management of the array is described in greater detail in the *Split Operations* clause in Annex C of TPM 2.0 Part 1.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

19.2 TPM2_Commit

19.2.1 General Description

TPM2_Commit() performs the first part of an ECC anonymous signing operation. The TPM will perform the point multiplications on the provided points and return intermediate signing values. The *signHandle* parameter shall refer to an ECC key and the signing scheme must be anonymous (TPM_RC_SCHEME).

NOTE 1 Currently, TPM_ALG_ECDAA is the only defined anonymous scheme.

NOTE 2 This command cannot be used with a sign+decrypt key because that type of key is required to have a scheme of TPM_ALG_NULL.

For this command, *p1*, *s2* and *y2* are optional parameters. If *s2* is an Empty Buffer, then the TPM shall return TPM_RC_SIZE if *y2* is not an Empty Buffer.

The algorithm is specified in the TPM 2.0 Part 1 Annex for ECC, TPM2_Commit().

 Copyright © TCG 2006-2020

19.2.2 Command and Response

Table 100 — TPM2_Commit Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Commit
TPMI_DH_OBJECT	@signHandle	handle of the key that will be used in the signing operation Auth Index: 1 Auth Role: USER
TPM2B_ECC_POINT	P1	a point (M) on the curve used by *signHandle*
TPM2B_SENSITIVE_DATA	s2	octet array used to derive x-coordinate of a base point
TPM2B_ECC_PARAMETER	y2	y coordinate of the point associated with *s2*

Table 101 — TPM2_Commit Response

Type	Name	Description
TPM_ST	tag	see 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ECC_POINT	K	ECC point $K \coloneqq [d_s](x2, y2)$
TPM2B_ECC_POINT	L	ECC point $L \coloneqq [r](x2, y2)$
TPM2B_ECC_POINT	E	ECC point $E \coloneqq [r]P1$
UINT16	counter	least-significant 16 bits of *commitCount*

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

19.2.3 Detailed Actions

[[Commit]]

19.3 TPM2_EC_Ephemeral

19.3.1 General Description

TPM2_EC_Ephemeral() creates an ephemeral key for use in a two-phase key exchange protocol.

The TPM will use the commit mechanism to assign an ephemeral key r and compute a public point $Q \coloneqq [r]G$ where G is the generator point associated with *curveID*.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

19.3.2 Command and Response

Table 102 — TPM2_EC_Ephemeral Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_EC_Ephemeral
TPMI_ECC_CURVE	curveID	The curve for the computed ephemeral point

Table 103 — TPM2_EC_Ephemeral Response

Type	Name	Description
TPM_ST	tag	see 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ECC_POINT	Q	ephemeral public key $Q \coloneqq [r]G$
UINT16	counter	least-significant 16 bits of *commitCount*

 Copyright © TCG 2006-2020

19.3.3 Detailed Actions

[[EC_Ephemeral]]

 Copyright © TCG 2006-2020

20 Signing and Signature Verification

20.1 TPM2_VerifySignature

20.1.1 General Description

This command uses loaded keys to validate a signature on a message with the message digest passed to the TPM.

If the signature check succeeds, then the TPM will produce a TPMT_TK_VERIFIED. Otherwise, the TPM shall return TPM_RC_SIGNATURE.

If the key is in the NULL hierarchy, then *digest* in the ticket will be the Empty Buffer.

NOTE 1 A valid ticket may be used in subsequent commands to provide proof to the TPM that the TPM has validated the signature over the message using the key referenced by *keyHandle*.

If *keyHandle* references an asymmetric key, only the public portion of the key needs to be loaded. If *keyHandle* references a symmetric key, both the public and private portions need to be loaded.

NOTE 2 The sensitive area of the symmetric object is required to allow verification of the symmetric signature (the HMAC).

20.1.2 Command and Response

Table 104 — TPM2_VerifySignature Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_VerifySignature
TPMI_DH_OBJECT	keyHandle	handle of public key that will be used in the validation Auth Index: None
TPM2B_DIGEST	digest	digest of the signed message
TPMT_SIGNATURE	signature	signature to be tested

Table 105 — TPM2_VerifySignature Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMT_TK_VERIFIED	validation	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

20.1.3 Detailed Actions

[[VerifySignature]]

20.2 TPM2_Sign

20.2.1 General Description

This command causes the TPM to sign an externally provided hash with the specified symmetric or asymmetric signing key.

NOTE 1 If *keyhandle* references an unrestricted signing key, a digest can be signed using either this command or an HMAC command.

If *keyHandle* references a restricted signing key, then *validation* shall be provided, indicating that the TPM performed the hash of the data and *validation* shall indicate that hashed data did not start with TPM_GENERATED_VALUE.

NOTE 2 If the hashed data did start with TPM_GENERATED_VALUE, then the validation will be a NULL ticket.

The *x509sign* attribute of keyHandle may not be SET (TPM_RC_ATTRIBUTES).

If the scheme of *keyHandle* is not TPM_ALG_NULL, then *inScheme* shall either be the same scheme as *keyHandle* or TPM_ALG_NULL. If the *sign* attribute is not SET in the key referenced by *handle* then the TPM shall return TPM_RC_KEY.

If the scheme of *keyHandle* is TPM_ALG_NULL, the TPM will sign using *inScheme*; otherwise, it will sign using the scheme of *keyHandle*.

NOTE 3 When the signing scheme uses a hash algorithm, the algorithm is defined in the qualifying data of the scheme. This is the same algorithm that is required to be used in producing *digest*. The size of *digest* must match that of the hash algorithm in the scheme.

If *inScheme* is not a valid signing scheme for the type of keyHandle (or TPM_ALG_NULL), then the TPM shall return TPM_RC_SCHEME.

If the scheme of *keyHandle* is an anonymous *scheme*, then *inScheme* shall have the same scheme algorithm as *keyHandle* and *inScheme* will contain a counter value that will be used in the signing process.

EXAMPLE For ECDAA, *inScheme.details.ecdaa.count* will contain the count value.

If *validation* is provided, then the hash algorithm used in computing the digest is required to be the hash algorithm specified in the scheme of *keyHandle* (TPM_RC_TICKET).

If the *validation* parameter is not the Empty Buffer*,* then it will be checked even if the key referenced by *keyHandle* is not a restricted signing key.

NOTE 4 If *keyHandle* is both a sign and decrypt key, *keyHandle* will have a scheme of TPM_ALG_NULL. If *validation* is provided, then it must be a NULL validation ticket or the ticket validation will fail.

 Copyright © TCG 2006-2020

20.2.2 Command and Response

Table 106 — TPM2_Sign Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Sign
TPMI_DH_OBJECT	@keyHandle	Handle of key that will perform signing Auth Index: 1 Auth Role: USER
TPM2B_DIGEST	digest	digest to be signed
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *keyHandle* is TPM_ALG_NULL
TPMT_TK_HASHCHECK	validation	proof that digest was created by the TPM If *keyHandle* is not a restricted signing key, then this may be a NULL Ticket with *tag* = TPM_ST_CHECKHASH.

Table 107 — TPM2_Sign Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMT_SIGNATURE	signature	the signature

20.2.3 Detailed Actions

[[Sign]]

21 Command Audit

21.1 Introduction

If a command has been selected for command audit, the command audit status will be updated when that command completes successfully. The digest is updated as:

$$commandAuditDigest_{new} \coloneqq \mathbf{H}_{auditAlg}(commandAuditDigest_{old} \mathbin{||} cpHash \mathbin{||} rpHash) \quad (5)$$

where

$\boldsymbol{H}_{auditAlg}$	hash function using the algorithm of the audit sequence
$commandAuditDigest$	accumulated digest
$cpHash$	the command parameter hash
$rpHash$	the response parameter hash

auditAlg, the hash algorithm, is set using TPM2_SetCommandCodeAuditStatus().

TPM2_Shutdown() cannot be audited but TPM2_Startup() can be audited. If the *cpHash* of the TPM2_Startup() is TPM_SU_STATE, that would indicate that a TPM2_Shutdown() had been successfully executed.

TPM2_SetCommandCodeAuditStatus() is always audited, except when it is used to change *auditAlg*.

If the TPM is in Failure mode, command audit is not functional.

 Copyright © TCG 2006-2020

21.2 TPM2_SetCommandCodeAuditStatus

21.2.1 General Description

This command may be used by the Privacy Administrator or platform to change the audit status of a command or to set the hash algorithm used for the audit digest, but not both at the same time.

If the *auditAlg* parameter is a supported hash algorithm and not the same as the current algorithm, then the TPM will check both *setList* and *clearList* are empty (zero length). If so, then the algorithm is changed, and the audit digest is cleared. If *auditAlg* is TPM_ALG_NULL or the same as the current algorithm, then the algorithm and audit digest are unchanged and the *setList* and *clearList* will be processed.

NOTE 1 Because the audit digest is cleared, the audit counter will increment the next time that an audited command is executed.

Use of TPM2_SetCommandCodeAuditStatus() to change the list of audited commands is an audited event. If TPM_CC_SetCommandCodeAuditStatus is in *clearList,* the fact that it is in *clearList* is ignored.

NOTE 2 Use of this command to change the audit hash algorithm is not audited and the digest is reset when the command completes. The change in the audit hash algorithm is the evidence that this command was used to change the algorithm.

The commands in *setList* indicate the commands to be added to the list of audited commands and the commands in *clearList* indicate the commands that will no longer be audited. It is not an error if a command in *setList* is already audited or is not implemented. It is not an error if a command in *clearList* is not currently being audited or is not implemented.

If a command code is in both *setList* and *clearList*, then it will not be audited (that is, *setList* shall be processed first).

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

21.2.2 Command and Response

Table 108 — TPM2_SetCommandCodeAuditStatus Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_SetCommandCodeAuditStatus {NV}
TPMI_RH_PROVISION	@auth	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPMI_ALG_HASH+	auditAlg	hash algorithm for the audit digest; if TPM_ALG_NULL, then the hash is not changed
TPML_CC	setList	list of commands that will be added to those that will be audited
TPML_CC	clearList	list of commands that will no longer be audited

Table 109 — TPM2_SetCommandCodeAuditStatus Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

21.2.3 Detailed Actions

[[SetCommandCodeAuditStatus]]

Copyright © TCG 2006-2020

22 Integrity Collection (PCR)

22.1 Introduction

In TPM 1.2, an Event was hashed using SHA-1 and then the 20-octet digest was extended to a PCR using TPM_Extend(). This specification allows the use of multiple PCR at a given Index, each using a different hash algorithm. Rather than require that the external software generate multiple hashes of the Event with each being extended to a different PCR, the Event data may be sent to the TPM for hashing. This ensures that the resulting digests will properly reflect the algorithms chosen for the PCR even if the calling software is unable to implement the hash algorithm.

NOTE 1 There is continued support for software hashing of events with TPM2_PCR_Extend().

To support recording of an Event that is larger than the TPM input buffer, the caller may use the command sequence described in clause 17.

Change to a PCR requires authorization. The authorization may be with either an authorization value or an authorization policy. The platform-specific specifications determine which PCR may be controlled by policy. All other PCR are controlled by authorization.

If a PCR may be associated with a policy, then the algorithm ID of that policy determines whether the policy is to be applied. If the algorithm ID is not TPM_ALG_NULL, then the policy digest associated with the PCR must match the *policySession→policyDigest* in a policy session. If the algorithm ID is TPM_ALG_NULL, then no policy is present and the authorization requires an EmptyAuth.

If a platform-specific specification indicates that PCR are grouped, then all the PCR in the group use the same authorization policy or authorization value.

pcrUpdateCounter counter will be incremented on the successful completion of any command that modifies (Extends or resets) a PCR unless the platform-specific specification explicitly excludes the PCR from being counted.

NOTE 2 If a command causes PCR in multiple banks to change, the PCR Update Counter must be incremented once for each bank. The commands that extend PCR are: TPM2_PCR_Extend, TPM2_PCR_Event, and TPM2_EventSequenceComplete.

If a command resets PCR in multiple banks, the PCR Update Counter must be incremented only once. The commands that reset PCR are: TPM2_PCR_Reset, and TPM2_Startup.

A platform-specific specification may designate a set of PCR that are under control of the TCB. These PCR may not be modified without the proper authorization. Updates of these PCR shall not cause the PCR Update Counter to increment.

EXAMPLE Updates of the TCB PCR will not cause the PCR update counter to increment because these PCR are changed at the whim of the TCB and may not represent the trust state of the platform.

 Copyright © TCG 2006-2020

22.2 TPM2_PCR_Extend

22.2.1 General Description

This command is used to cause an update to the indicated PCR. The *digests* parameter contains one or more tagged digest values identified by an algorithm ID. For each digest, the PCR associated with *pcrHandle* is Extended into the bank identified by the tag (*hashAlg*).

EXAMPLE A SHA1 digest would be Extended into the SHA1 bank and a SHA256 digest would be Extended into the SHA256 bank.

For each list entry, the TPM will check to see if *pcrNum* is implemented for that algorithm. If so, the TPM shall perform the following operation:

$$PCR.digest_{new}[pcrNum][alg] \coloneqq \mathbf{H}_{alg}(PCR.digest_{old}[pcrNum][alg] \mid\mid data[alg].buffer) \quad (6)$$

where

$\mathbf{H}_{alg}()$	hash function using the hash algorithm associated with the PCR instance
$PCR.digest$	the digest value in a PCR
$pcrNum$	the PCR numeric selector (*pcrHandle*)
alg	the PCR algorithm selector for the digest
$data[alg].buffer$	the bank-specific data to be extended

If no digest value is specified for a bank, then the PCR in that bank is not modified.

NOTE 1 This allows consistent operation of the digests list for all of the Event recording commands.

If a digest is present and the PCR in that bank is not implemented, the digest value is not used.

NOTE 2 If the caller includes digests for algorithms that are not implemented, then the TPM will fail the call because the unmarshalling of *digests* will fail. Each of the entries in the list is a TPMT_HA, which is a hash algorithm followed by a digest. If the algorithm is not implemented, unmarshalling of the *hashAlg* will fail and the TPM will return TPM_RC_HASH.

If the TPM unmarshals the *hashAlg* of a list entry and the unmarshaled value is not a hash algorithm implemented on the TPM, the TPM shall return TPM_RC_HASH.

The *pcrHandle* parameter is allowed to reference TPM_RH_NULL. If so, the input parameters are processed but no action is taken by the TPM. This permits the caller to probe for implemented hash algorithms as an alternative to TPM2_GetCapability.

NOTE 3 This command allows a list of digests so that PCR in all banks may be updated in a single command. While the semantics of this command allow multiple extends to a single PCR bank, this is not the preferred use and the limit on the number of entries in the list make this use somewhat impractical.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

22.2.2 Command and Response

Table 110 — TPM2_PCR_Extend Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_Extend {NV}
TPMI_DH_PCR+	@pcrHandle	handle of the PCR Auth Handle: 1 Auth Role: USER
TPML_DIGEST_VALUES	digests	list of tagged digest values to be extended

Table 111 — TPM2_PCR_Extend Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.

 Copyright © TCG 2006-2020

22.2.3 Detailed Actions

[[PCR_Extend]]

 Copyright © TCG 2006-2020

22.3 TPM2_PCR_Event

22.3.1 General Description

This command is used to cause an update to the indicated PCR.

The data in *eventData* is hashed using the hash algorithm associated with each bank in which the indicated PCR has been allocated. After the data is hashed, the *digests* list is returned. If the *pcrHandle* references an implemented PCR and not TPM_RH_NULL, the *digests* list is processed as in TPM2_PCR_Extend().

A TPM shall support an *Event.size* of zero through 1,024 inclusive (*Event.size* is an octet count). An *Event.size* of zero indicates that there is no data but the indicated operations will still occur,

EXAMPLE 1 If the command implements PCR[2] in a SHA1 bank and a SHA256 bank, then an extend to PCR[2] will cause *eventData* to be hashed twice, once with SHA1 and once with SHA256. The SHA1 hash of *eventData* will be Extended to PCR[2] in the SHA1 bank and the SHA256 hash of *eventData* will be Extended to PCR[2] of the SHA256 bank.

On successful command completion, *digests* will contain the list of tagged digests of *eventData* that was computed in preparation for extending the data into the PCR. At the option of the TPM, the list may contain a digest for each bank, or it may only contain a digest for each bank in which *pcrHandle* is extant. If *pcrHandle* is TPM_RH_NULL, the TPM may return either an empty list or a digest for each bank.

EXAMPLE 2 Assume a TPM that implements a SHA1 bank and a SHA256 bank and that PCR[22] is only implemented in the SHA1 bank. If *pcrHandle* references PCR[22], then *digests* may contain either a SHA1 and a SHA256 digest or just a SHA1 digest.

22.3.2 Command and Response

Table 112 — TPM2_PCR_Event Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_Event {NV}
TPMI_DH_PCR+	@pcrHandle	Handle of the PCR Auth Handle: 1 Auth Role: USER
TPM2B_EVENT	eventData	Event data in sized buffer

Table 113 — TPM2_PCR_Event Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPML_DIGEST_VALUES	digests	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

22.3.3 Detailed Actions

[[PCR_Event]]

22.4 TPM2_PCR_Read

22.4.1 General Description

This command returns the values of all PCR specified in *pcrSelectionIn*.

The TPM will process the list of TPMS_PCR_SELECTION in *pcrSelectionIn* in order. Within each TPMS_PCR_SELECTION, the TPM will process the bits in the *pcrSelect* array in ascending PCR order (see TPM 2.0 Part 1, *Selecting Multiple PCR*). If a bit is SET, and the indicated PCR is present, then the TPM will add the digest of the PCR to the list of values to be returned in *pcrValues.*

The TPM will continue processing bits until all have been processed or until *pcrValues* would be too large to fit into the output buffer if additional values were added.

The returned *pcrSelectionOut* will have a bit SET in its *pcrSelect* structures for each value present in *pcrValues*.

The current value of the PCR Update Counter is returned in *pcrUpdateCounter*.

The returned list may be empty if none of the selected PCR are implemented.

NOTE If no PCR are returned from a bank, the selector for the bank will be present in *pcrSelectionOut*.

No authorization is required to read a PCR and any implemented PCR may be read from any locality.

 Copyright © TCG 2006-2020

22.4.2 Command and Response

Table 114 — TPM2_PCR_Read Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_Read
TPML_PCR_SELECTION	pcrSelectionIn	The selection of PCR to read

Table 115 — TPM2_PCR_Read Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
UINT32	pcrUpdateCounter	the current value of the PCR update counter
TPML_PCR_SELECTION	pcrSelectionOut	the PCR in the returned list
TPML_DIGEST	pcrValues	the contents of the PCR indicated in *pcrSelectOut->pcrSelection[]* as tagged digests

 Copyright © TCG 2006-2020

22.4.3 Detailed Actions

[[PCR_Read]]

 Copyright © TCG 2006-2020

22.5 TPM2_PCR_Allocate

22.5.1 General Description

This command is used to set the desired PCR allocation of PCR and algorithms. This command requires Platform Authorization.

The TPM will evaluate the request and, if sufficient memory is available for the requested allocation, the TPM will store the allocation request for use during the next _TPM_Init operation. The PCR allocation in place when this command is executed will be retained until the next _TPM_Init. If this command is received multiple times before a _TPM_Init, each one overwrites the previous stored allocation.

This command will only change the allocations of banks that are listed in *pcrAllocation*.

EXAMPLE 1 If a TPM supports SHA1 and SHA256, then it maintains an allocation for two banks (one of which could be empty). If *pcrAllocation* only has a selector for the SHA1 bank, then only the allocation of the SHA1 bank will be changed and the SHA256 bank will remain unchanged. To change the allocation of a TPM from 24 SHA1 PCR and no SHA256 PCR to 24 SHA256 PCR and no SHA1 PCR, the *pcrAllocation* would have to have two selections: one for the empty SHA1 bank and one for the SHA256 bank with 24 PCR.

If a bank is listed more than once, then the last selection in the *pcrAllocation* list is the one that the TPM will attempt to allocate.

NOTE 1 This does not mean to imply that *pcrAllocation.count* can exceed HASH_COUNT, the number of digests implemented in the TPM.

EXAMPLE 2 If HASH_COUNT is 2, *pcrAllocation* can specify SHA-256 twice, and the second one is used. However, if SHA_256 is specified three times, the unmarshaling may fail and the TPM may return an error.

This command shall not allocate more PCR in any bank than there are PCR attribute definitions. The PCR attribute definitions indicate how a PCR is to be managed – if it is resettable, the locality for update, etc. In the response to this command, the TPM returns the maximum number of PCR allowed for any bank.

When PCR are allocated, if DRTM_PCR is defined, the resulting allocation must have at least one bank with the D-RTM PCR allocated. If HCRTM_PCR is defined, the resulting allocation must have at least one bank with the HCRTM_PCR allocated. If not, the TPM returns TPM_RC_PCR.

The TPM may return TPM_RC_SUCCESS even though the request fails. This is to allow the TPM to return information about the size needed for the requested allocation and the size available. If the *sizeNeeded* parameter in the return is less than or equal to the *sizeAvailable* parameter, then the *allocationSuccess* parameter will be YES. Alternatively, if the request fails, The TPM may return TPM_RC_NO_RESULT.

NOTE 2 An example for this type of failure is a TPM that can only support one bank at a time and cannot support arbitrary distribution of PCR among banks.

After this command, TPM2_Shutdown() is only allowed to have a *startupType* equal to TPM_SU_CLEAR until after the next _TPM_Init.

NOTE 3 Even if this command does not cause the PCR allocation to change, the TPM cannot have its state saved. This is done in order to simplify the implementation. There is no need to optimize this command as it is not expected to be used more than once in the lifetime of the TPM (it can be used any number of times but there is no justification for optimization).

 Copyright © TCG 2006-2020

22.5.2 Command and Response

Table 116 — TPM2_PCR_Allocate Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_Allocate {NV}
TPMI_RH_PLATFORM	@authHandle	TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPML_PCR_SELECTION	pcrAllocation	the requested allocation

Table 117 — TPM2_PCR_Allocate Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_YES_NO	allocationSuccess	YES if the allocation succeeded
UINT32	maxPCR	maximum number of PCR that may be in a bank
UINT32	sizeNeeded	number of octets required to satisfy the request
UINT32	sizeAvailable	Number of octets available. Computed before the allocation.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

22.5.3 Detailed Actions

[[PCR_Allocate]]

22.6 TPM2_PCR_SetAuthPolicy

22.6.1 General Description

This command is used to associate a policy with a PCR or group of PCR. The policy determines the conditions under which a PCR may be extended or reset.

A policy may only be associated with a PCR that has been defined by a platform-specific specification as allowing a policy. If the TPM implementation does not allow a policy for *pcrNum*, the TPM shall return TPM_RC_VALUE.

A platform-specific specification may group PCR so that they share a common policy. In such case, a *pcrNum* that selects any of the PCR in the group will change the policy for all PCR in the group.

The policy setting is persistent and may only be changed by TPM2_PCR_SetAuthPolicy() or by TPM2_ChangePPS().

Before this command is first executed on a TPM or after TPM2_ChangePPS(), the access control on the PCR will be set to the default value defined in the platform-specific specification.

NOTE 1 It is expected that the typical default will be with the policy hash set to TPM_ALG_NULL and an Empty Buffer for the *authPolicy* value. This will allow an *EmptyAuth* to be used as the authorization value.

If the size of the data buffer in *authPolicy* is not the size of a digest produced by *hashAlg*, the TPM shall return TPM_RC_SIZE.

NOTE 2 If *hashAlg* is TPM_ALG_NULL, then the size is required to be zero.

This command requires platformAuth/platformPolicy.

NOTE 3 If the PCR is in multiple policy sets, the policy will be changed in only one set. The set that is changed will be implementation dependent.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

22.6.2 Command and Response

Table 118 — TPM2_PCR_SetAuthPolicy Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_SetAuthPolicy {NV}
TPMI_RH_PLATFORM	@authHandle	TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPM2B_DIGEST	authPolicy	the desired *authPolicy*
TPMI_ALG_HASH+	hashAlg	the hash algorithm of the policy
TPMI_DH_PCR	pcrNum	the PCR for which the policy is to be set

Table 119 — TPM2_PCR_SetAuthPolicy Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

22.6.3 Detailed Actions

[[PCR_SetAuthPolicy]]

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

22.7 TPM2_PCR_SetAuthValue

22.7.1 General Description

This command changes the *authValue* of a PCR or group of PCR.

An *authValue* may only be associated with a PCR that has been defined by a platform-specific specification as allowing an authorization value. If the TPM implementation does not allow an authorization for *pcrNum*, the TPM shall return TPM_RC_VALUE. A platform-specific specification may group PCR so that they share a common authorization value. In such case, a *pcrNum* that selects any of the PCR in the group will change the *authValue* value for all PCR in the group.

The authorization setting is set to EmptyAuth on each STARTUP(CLEAR) or by TPM2_Clear(). The authorization setting is preserved by SHUTDOWN(STATE).

 Copyright © TCG 2006-2020

22.7.2 Command and Response

Table 120 — TPM2_PCR_SetAuthValue Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_SetAuthValue
TPMI_DH_PCR	@pcrHandle	handle for a PCR that may have an authorization value set Auth Index: 1 Auth Role: USER
TPM2B_DIGEST	auth	the desired authorization value

Table 121 — TPM2_PCR_SetAuthValue Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

22.7.3 Detailed Actions

[[PCR_SetAuthValue]]

22.8 TPM2_PCR_Reset

22.8.1 General Description

If the attribute of a PCR allows the PCR to be reset and proper authorization is provided, then this command may be used to set the PCR in all banks to zero. The attributes of the PCR may restrict the locality that can perform the reset operation.

NOTE 1 The definition of TPMI_DH_PCR in TPM 2.0 Part 2 indicates that if pcrHandle is out of the allowed range for PCR, then the appropriate return value is TPM_RC_VALUE.

If *pcrHandle* references a PCR that cannot be reset, the TPM shall return TPM_RC_LOCALITY.

NOTE 2 TPM_RC_LOCALITY is returned because the reset attributes are defined on a per-locality basis.

 Copyright © TCG 2006-2020

22.8.2 Command and Response

Table 122 — TPM2_PCR_Reset Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_Reset {NV}
TPMI_DH_PCR	@pcrHandle	the PCR to reset Auth Index: 1 Auth Role: USER

Table 123 — TPM2_PCR_Reset Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

22.8.3 Detailed Actions

[[PCR_Reset]]

 Copyright © TCG 2006-2020

22.9 _TPM_Hash_Start

22.9.1 Description

This indication from the TPM interface indicates the start of an H-CRTM measurement sequence. On receipt of this indication, the TPM will initialize an H-CRTM Event Sequence context.

If no object memory is available for creation of the sequence context, the TPM will flush the context of an object so that creation of the sequence context will always succeed.

A platform-specific specification may allow this indication before TPM2_Startup().

NOTE If this indication occurs after TPM2_Startup(), it is the responsibility of software to ensure that an object context slot is available or to deal with the consequences of having the TPM select an arbitrary object to be flushed. If this indication occurs before TPM2_Startup() then all context slots are available.

 Copyright © TCG 2006-2020

22.9.2 Detailed Actions

[[_TPM_Hash_Start]]

 Copyright © TCG 2006-2020

22.10 _TPM_Hash_Data

22.10.1 Description

This indication from the TPM interface indicates arrival of one or more octets of data that are to be included in the H-CRTM Event Sequence sequence context created by the _TPM_Hash_Start indication. The context holds data for each hash algorithm for each PCR bank implemented on the TPM.

If no H-CRTM Event Sequence context exists, this indication is discarded and no other action is performed.

 Copyright © TCG 2006-2020

22.10.2 Detailed Actions

[[_TPM_Hash_Data]]

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

22.11 _TPM_Hash_End

22.11.1 Description

This indication from the TPM interface indicates the end of the H-CRTM measurement. This indication is discarded and no other action performed if the TPM does not contain an H-CRTM Event Sequence context.

NOTE 1 An H-CRTM Event Sequence context is created by _TPM_Hash_Start().

If the H-CRTM Event Sequence occurs after TPM2_Startup(), the TPM will set all of the PCR designated in the platform-specific specifications as resettable by this event to the value indicated in the platform specific specification and increment *restartCount*. The TPM will then Extend the Event Sequence digest/digests into the designated D-RTM PCR (PCR[17]).

$$\text{PCR}[17][hashAlg] \coloneqq \mathbf{H}_{hashAlg}(initial_value \,||\, \mathbf{H}_{hashAlg}(hash_data)) \quad (7)$$

where

$hashAlg$	hash algorithm associated with a bank of PCR
$initial_value$	initialization value specified in the platform-specific specification (should be 0…0)
$hash_data$	all the octets of data received in _TPM_Hash_Data indications

A _TPM_Hash_End indication that occurs after TPM2_Startup() will increment *pcrUpdateCounter* unless a platform-specific specification excludes modifications of PCR[DRTM] from causing an increment.

A platform-specific specification may allow an H-CRTM Event Sequence before TPM2_Startup(). If so, _TPM_Hash_End will complete the digest, initialize PCR[0] with a digest-size value of 4, and then extend the H-CRTM Event Sequence data into PCR[0].

$$\text{PCR}[0][hashAlg] \coloneqq \mathbf{H}_{hashAlg}(0...04 \,||\, \mathbf{H}_{hashAlg}(hash_data)) \quad (8)$$

NOTE 2 The entire sequence of _TPM_Hash_Start, _TPM_Hash_Data, and _TPM_Hash_End are required to complete before TPM2_Startup() or the sequence will have no effect on the TPM.

NOTE 3 PCR[0] does not need to be updated according to (8) until the end of TPM2_Startup().

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

22.11.2 Detailed Actions

[[_TPM_Hash_End]]

 Copyright © TCG 2006-2020

23 Enhanced Authorization (EA) Commands

23.1 Introduction

The commands in this clause 23 are used for policy evaluation. When successful, each command will update the *policySession→policyDigest* in a policy session context in order to establish that the authorizations required to use an object have been provided. Many of the commands will also modify other parts of a policy context so that the caller may constrain the scope of the authorization that is provided.

NOTE 1 Many of the terms used in this clause are described in detail in TPM 2.0 Part 1 and are not redefined in this clause.

The *policySession* parameter of the command is the handle of the policy session context to be modified by the command.

If the *policySession* parameter indicates a trial policy session, then the *policySession→policyDigest* will be updated and the indicated validations are not performed. However, any authorizations required to perform the policy command will be checked and dictionary attack logic invoked as necessary.

NOTE 2 If software is used to create policies, no authorization values are used. For example, TPM_PolicySecret requires an authorization in a trial policy session, but not in a policy calculation outside the TPM.

NOTE 3 A policy session is set to a trial policy by TPM2_StartAuthSession(*sessionType* = TPM_SE_TRIAL).

NOTE 4 Unless there is an unmarshaling error in the parameters of the command, these commands will return TPM_RC_SUCCESS when *policySession* references a trial session.

NOTE 5 Policy context other than the *policySession→policyDigest* may be updated for a trial policy but it is not required.

 Copyright © TCG 2006-2020

23.2 Signed Authorization Actions

23.2.1 Introduction

The TPM2_PolicySigned, TPM_PolicySecret, and TPM2_PolicyTicket commands use many of the same functions. This clause consolidates those functions to simplify the document and to ensure uniformity of the operations.

23.2.2 Policy Parameter Checks

These parameter checks will be performed when indicated in the description of each of the commands:

a) *nonceTPM* – If this parameter is not the Empty Buffer, and it does not match *policySession→nonceTPM*, then the TPM shall return TPM_RC_VALUE.

b) *expiration* – If this parameter is not zero, then:

   1) if *nonceTPM* is not an Empty Buffer, then the absolute value of *expiration* is converted to milliseconds and added to *policySession→startTime* to create the *timeout* value and proceed to c).

   2) If *nonceTPM* is an Empty Buffer, then the absolute value of *expiration* is converted to milliseconds and used as the *timeout* value and proceed to c).

   However, *timeout* can only be changed to a smaller value.

c) *timeout* – If *timeout* is less than the current value of *Time*, or the current *timeEpoch* is not the same as *policySession→timeEpoch, the* TPM shall return TPM_RC_EXPIRED

d) *cpHashA* – If this parameter is not an Empty Buffer

   NOTE 2 *cpHashA* is the hash of the command to be executed using this policy session in the authorization. The algorithm used to compute this hash is required to be the algorithm of the policy session.

   1) the TPM shall return TPM_RC_CPHASH if *policySession→cpHash* is set and the contents of *policySession→cpHash* are not the same as *cpHashA*; or

      NOTE 3 cpHash is the expected cpHash value held in the policy session context.

   2) the TPM shall return TPM_RC_SIZE if *cpHashA* is not the same size as *policySession→policyDigest*.

      NOTE 4 policySession→policyDigest is the size of the digest produced by the hash algorithm used to compute policyDigest.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.2.3 Policy Digest Update Function (PolicyUpdate())

This is the update process for *policySession→policyDigest* used by TPM2_PolicySigned(), TPM2_PolicySecret(), TPM2_PolicyTicket(), and TPM2_PolicyAuthorize(). The function prototype for the update function is:

$$\mathbf{PolicyUpdate}(commandCode, arg2, arg3) \quad (9)$$

where

$arg2$	a TPM2B_NAME
$arg3$	a TPM2B

These parameters are used to update *policySession→policyDigest* by

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \;||\; commandCode \;||\; arg2.name) \quad (10)$$

followed by

$$policyDigest_{new+1} \coloneqq \mathbf{H}_{\mathrm{policyAlg}}(policyDigest_{new} \;||\; arg3.buffer) \quad (11)$$

where

$\mathbf{H}_{policyAlg}()$	the hash algorithm chosen when the policy session was started

NOTE 1 If *arg3* is a TPM2B_NAME, then *arg3.buffer* will actually be an *arg3.name*.

NOTE 2 The *arg2.size* and *arg3.size* fields are not included in the hashes.

NOTE 3 **PolicyUpdate**() uses two hash operations because *arg2* and *arg3* are variable-sized and the concatenation of *arg2* and *arg3* in a single hash could produce the same digest even though *arg2* and *arg3* are different. For example, arg2 = 1 2 3 and arg3 = 4 5 6 would produce the same digest as arg2 = 1 2 and arg3 = 3 4 5 6. Processing of the arguments separately in different Extend operation ensures that the digest produced by **PolicyUpdate**() will be different if *arg2* and *arg3* are different.

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

23.2.4 Policy Context Updates

When a policy command modifies some part of the policy session context other than the *policySession→policyDigest*, the following rules apply.

- ***cpHash*** – this parameter may only be changed if it contains its initialization value (an Empty Buffer). If *cpHash* is not the Empty Buffer when a policy command attempts to update it, the TPM will return an error (TPM_RC_CPHASH) if the current and update values are not the same.
- ***timeOut*** – this parameter may only be changed to a smaller value. If a command attempts to update this value with a larger value (longer into the future), the TPM will discard the update value. This is not an error condition.
- ***commandCode*** – once set by a policy command, this value may not be changed except by TPM2_PolicyRestart(). If a policy command tries to change this to a different value, an error is returned (TPM_RC_POLICY_CC).
- ***pcrUpdateCounter*** – this parameter is updated by TPM2_PolicyPCR(). This value may only be set once during a policy. Each time TPM2_PolicyPCR() executes, it checks to see if *policySession→pcrUpdateCounter* has its default state, indicating that this is the first TPM2_PolicyPCR(). If it has its default value, then *policySession→pcrUpdateCounter* is set to the current value of *pcrUpdateCounter*. If *policySession→pcrUpdateCounter* does not have its default value and its value is not the same as *pcrUpdateCounter*, the TPM shall return TPM_RC_PCR_CHANGED.

  NOTE 1 If this parameter and *pcrUpdateCounter* are not the same, it indicates that PCR have changed since checked by the previous TPM2_PolicyPCR(). Since they have changed, the previous PCR validation is no longer valid.

- ***commandLocality*** – this parameter is the logical AND of all enabled localities. All localities are enabled for a policy when the policy session is created. TPM2_PolicyLocalities() selectively disables localities. Once use of a policy for a locality has been disabled, it cannot be enabled except by TPM2_PolicyRestart().
- ***isPPRequired*** – once SET, this parameter may only be CLEARed by TPM2_PolicyRestart().
- ***isAuthValueNeeded*** – once SET, this parameter may only be CLEARed by TPM2_PolicyPassword() or TPM2_PolicyRestart().
- ***isPasswordNeeded*** – once SET, this parameter may only be CLEARed by TPM2_PolicyAuthValue() or TPM2_PolicyRestart(),

NOTE 2 Both TPM2_PolicyAuthValue() and TPM2_PolicyPassword() change *policySession→policyDigest* in the same way. The different commands simply indicate to the TPM the format used for the *authValue* (HMAC or clear text). Both commands could be in the same policy. The final instance of these commands determines the format.

 Copyright © TCG 2006-2020

23.2.5 Policy Ticket Creation

For TPM2_PolicySigned() or TPM2_PolicySecret(), if the caller specified a negative value for *expiration*, then the TPM will return a ticket that includes a value indicating when the authorization expires. Otherwise, the TPM will return a NULL Ticket.

NOTE 1 If the *authHandle* in TPM2_PolicySecret() references a PIN Pass Index, then the command may succeed but a NULL Ticket will be returned.

The required computation for the digest in the authorization ticket is:

$$\mathbf{HMAC}_{contextAlg}(proof, (\text{TPM_ST_AUTH_xxx} \mathbin{||} cpHash \mathbin{||} policyRef \mathbin{||} authName \mathbin{||} timeout \mathbin{||} [timeEpoch] \mathbin{||} [resetCount])) \quad (12)$$

where

$\mathbf{HMAC}_{contextAlg}()$	an HMAC using the context integrity hash
$proof$	a TPM secret value associated with the hierarchy of the object associated with *authName*
TPM_ST_AUTH_xxx	either TPM_ST_AUTH_SIGNED or TPM_ST_AUTH_SECRET; used to ensure that the ticket is properly used
$cpHash$	optional hash of the authorized command
$policyRef$	optional reference to a policy value
$authName$	Name of the object that signed the authorization
$timeout$	implementation-specific value indicating when the authorization expires
$timeEpoch$	implementation-specific representation of the $timeEpoch$ at the time the ticket was created

NOTE 2 Not included if *timeout* is zero.

$resetCount$	implementation-specific representation of the TPM's *totalResetCount*

NOTE 3 Not included it *timeout* is zero or if *nonceTPM* was include in the authorization.

 Copyright © TCG 2006-2020

23.3 TPM2_PolicySigned

23.3.1 General Description

This command includes a signed authorization in a policy. The command ties the policy to a signing key by including the Name of the signing key in the *policyDigest*

If *policySession* is a trial session, the TPM will not check the signature and will update *policySession→policyDigest* as described in 23.2.3 as if a properly signed authorization was received, but no ticket will be produced.

If *policySession* is not a trial session, the TPM will validate *auth* and only perform the update if it is a valid signature over the fields of the command.

The authorizing entity will sign a digest of the authorization qualifiers: *nonceTPM*, *expiration*, *cpHashA*, and *policyRef*. The digest is computed as:

$$aHash \coloneqq \mathbf{H}_{authAlg}(nonceTPM \mid\mid expiration \mid\mid cpHashA \mid\mid policyRef) \quad (13)$$

where

$\mathbf{H}_{authAlg}()$ the hash associated with the auth parameter of this command

NOTE 1 Each signature and key combination indicates the scheme and each scheme has an associated hash.

$nonceTPM$ the nonceTPM parameter from the TPM2_StartAuthSession() response. If the authorization is not limited to this session, the size of this value is zero.

$expiration$ time limit on authorization set by authorizing object. This 32-bit value is set to zero if the expiration time is not being set.

$cpHashA$ digest of the command parameters for the command being approved using the hash algorithm of the policy session. Set to an Empty Digest if the authorization is not limited to a specific command.

NOTE 3 This is not the *cpHash* of this TPM2_PolicySigned() command.

$policyRef$ an opaque value determined by the authorizing entity. Set to the Empty Buffer if no value is present.

NOTE 4 The *nonceTPM*, *cpHashA*, and *policyRef* qualifiers used to compute *aHash* use the TPM2B buffer but do not prepend the size.

EXAMPLE The computation for an *aHash* if there are no restrictions is:

$$aHash \coloneqq \mathbf{H}_{authAlg}(00\ 00\ 00\ 00_{16})$$

which is the hash of an expiration time of zero.

The *aHash* is signed by the key associated with a key whose handle is *authObject*. The signature and signing parameters are combined to create the *auth* parameter.

The TPM will perform the parameter checks listed in 23.2.2

If the parameter checks succeed, the TPM will construct a test digest (*tHash*) over the provided parameters using the same formulation as shown in equation (13) above.

If *tHash* does not match the digest of the signed *aHash,* then the authorization fails and the TPM shall return TPM_RC_POLICY_FAIL and make no change to *policySession→policyDigest*.

 Copyright © TCG 2006-2020

When all validations have succeeded, *policySession→policyDigest* is updated by **PolicyUpdate**() (see 23.2.3).

$$\mathbf{PolicyUpdate}(\text{TPM_CC_PolicySigned}, authObject{\rightarrow}Name, policyRef) \tag{14}$$

authObject→Name is a TPM2B_NAME. *policySession* is updated as described in 23.2.4. The TPM will optionally produce a ticket as described in 23.2.5.

Authorization to use *authObject* is not required.

 Copyright © TCG 2006-2020

23.3.2 Command and Response

Table 124 — TPM2_PolicySigned Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit, encrypt, or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicySigned
TPMI_DH_OBJECT	authObject	handle for a key that will validate the signature Auth Index: None
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_NONCE	nonceTPM	the policy nonce for the session This can be the Empty Buffer.
TPM2B_DIGEST	cpHashA	digest of the command parameters to which this authorization is limited This is not the *cpHash* for this command but the *cpHash* for the command to which this policy session will be applied. If it is not limited, the parameter will be the Empty Buffer.
TPM2B_NONCE	policyRef	a reference to a policy relating to the authorization – may be the Empty Buffer Size is limited to be no larger than the nonce size supported on the TPM.
INT32	expiration	time when authorization will expire, measured in seconds from the time that *nonceTPM* was generated If *expiration* is non-negative, a NULL Ticket is returned. See 23.2.5.
TPMT_SIGNATURE	auth	signed authorization (not optional)

Table 125 — TPM2_PolicySigned Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_TIMEOUT	timeout	implementation-specific time value, used to indicate to the TPM when the ticket expires NOTE If *policyTicket* is a NULL Ticket, then this shall be the Empty Buffer.
TPMT_TK_AUTH	policyTicket	produced if the command succeeds and *expiration* in the command was non-zero; this ticket will use the TPMT_ST_AUTH_SIGNED structure tag. See 23.2.5

23.3.3 Detailed Actions

[[PolicySigned]]

23.4 TPM2_PolicySecret

23.4.1 General Description

This command includes a secret-based authorization to a policy. The caller proves knowledge of the secret value using an authorization session using the *authValue* associated with *authHandle*. A password session, an HMAC session, or a policy session containing TPM2_PolicyAuthValue() or TPM2_PolicyPassword() will satisfy this requirement.

If a policy session is used and use of the *authValue* of *authHandle* is not required, the TPM will return TPM_RC_MODE. That is, the session for *authHandle* must have either *isAuthValueNeeded* or *isPasswordNeeded* SET.

The secret is the *authValue* of the entity whose handle is *authHandle,* which may be any TPM entity with a handle and an associated *authValue*. This includes the reserved handles (for example, Platform, Storage, and Endorsement), NV Indexes, and loaded objects. *authEntity* is the entity referenced by *authHandle*. If *authEntity* references an Ordinary object, it must have *userWithAuth* SET.

NOTE 1 The *userWithAuth* requirement permits the implementation to use common authorization code.

If *authEntity* references a non-PIN Index. TPMA_NV_AUTHREAD is required to be SET in the Index. If *authEntity* references an NV PIN index, TPMA_NV_WRITTEN is required to be SET and *pinCount* must be less than *pinLimit*.

NOTE 2 The authorization value for a hierarchy cannot be used in this command if the hierarchy is disabled.

If the authorization check fails, then the normal dictionary attack logic is invoked.

If the authorization provided by the authorization session is valid, the command parameters are checked as described in 23.2.2.

When all validations have succeeded, *policySession→policyDigest* is updated by **PolicyUpdate**() (see 23.2.3).

$$\textbf{PolicyUpdate}(\text{TPM_CC_PolicySecret}, authEntity{\rightarrow}Name, policyRef) \qquad (15)$$

authEntity→Name is a TPM2B_NAME. *policySession* is updated as described in 23.2.4. The TPM will optionally produce a ticket as described in 23.2.5.

If the session is a trial session, *policySession→policyDigest* is updated if the authorization is valid.

NOTE 2 If an HMAC is used to convey the authorization, a separate session is needed for the authorization. Because the HMAC in that authorization will include a nonce that prevents replay of the authorization, the value of the *nonceTPM* parameter in this command is limited. It is retained mostly to provide processing consistency with TPM2_PolicySigned().

 Copyright © TCG 2006-2020

23.4.2 Command and Response

Table 126 — TPM2_PolicySecret Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicySecret
TPMI_DH_ENTITY	@authHandle	handle for an entity providing the authorization Auth Index: 1 Auth Role: USER
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_NONCE	nonceTPM	the policy nonce for the session This can be the Empty Buffer.
TPM2B_DIGEST	cpHashA	digest of the command parameters to which this authorization is limited This not the *cpHash* for this command but the *cpHash* for the command to which this policy session will be applied. If it is not limited, the parameter will be the Empty Buffer.
TPM2B_NONCE	policyRef	a reference to a policy relating to the authorization – may be the Empty Buffer Size is limited to be no larger than the nonce size supported on the TPM.
INT32	expiration	time when authorization will expire, measured in seconds from the time that *nonceTPM* was generated If *expiration* is non-negative, a NULL Ticket is returned. See 23.2.5.

Table 127 — TPM2_PolicySecret Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_TIMEOUT	timeout	implementation-specific time value used to indicate to the TPM when the ticket expires
TPMT_TK_AUTH	policyTicket	produced if the command succeeds and *expiration* in the command was non-zero (See 23.2.5). This ticket will use the TPMT_ST_AUTH_SECRET structure tag

23.4.3 Detailed Actions

[[PolicySecret]]

 Copyright © TCG 2006-2020

23.5 TPM2_PolicyTicket

23.5.1 General Description

This command is similar to TPM2_PolicySigned() except that it takes a ticket instead of a signed authorization. The ticket represents a validated authorization that had an expiration time associated with it.

The parameters of this command are checked as described in 23.2.2.

If the checks succeed, the TPM uses the *timeout*, *cpHashA*, *policyRef*, and *authName* to construct a ticket to compare with the value in *ticket*. If these tickets match, then the TPM will create a TPM2B_NAME (*objectName*) using *authName* and update the context of *policySession* by **PolicyUpdate**() (see 23.2.3).

$$\mathbf{PolicyUpdate}(commandCode, authName, policyRef) \tag{16}$$

If the structure tag of ticket is TPM_ST_AUTH_SECRET, then *commandCode* will be TPM_CC_PolicySecret. If the structure tag of ticket is TPM_ST_AUTH_SIGNED, then *commandCode* will be TPM_CC_PolicySIgned.

policySession is updated as described in 23.2.4.

 Copyright © TCG 2006-2020

23.5.2 Command and Response

Table 128 — TPM2_PolicyTicket Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyTicket
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_TIMEOUT	timeout	time when authorization will expire The contents are TPM specific. This shall be the value returned when ticket was produced.
TPM2B_DIGEST	cpHashA	digest of the command parameters to which this authorization is limited If it is not limited, the parameter will be the Empty Buffer.
TPM2B_NONCE	policyRef	reference to a qualifier for the policy – may be the Empty Buffer
TPM2B_NAME	authName	name of the object that provided the authorization
TPMT_TK_AUTH	ticket	an authorization ticket returned by the TPM in response to a TPM2_PolicySigned() or TPM2_PolicySecret()

Table 129 — TPM2_PolicyTicket Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.5.3 Detailed Actions

[[PolicyTicket]]

 Copyright © TCG 2006-2020

23.6 TPM2_PolicyOR

23.6.1 General Description

This command allows options in authorizations without requiring that the TPM evaluate all of the options. If a policy may be satisfied by different sets of conditions, the TPM need only evaluate one set that satisfies the policy. This command will indicate that one of the required sets of conditions has been satisfied.

PolicySession→policyDigest is compared against the list of provided values. If the current *policySession→policyDigest* does not match any value in the list, the TPM shall return TPM_RC_VALUE. Otherwise, the TPM will reset *policySession→policyDigest* to a Zero Digest. Then *policySession→policyDigest* is extended by the concatenation of TPM_CC_PolicyOR and the concatenation of all of the digests.

If *policySession* is a trial session, the TPM will assume that *policySession→policyDigest* matches one of the list entries and compute the new value of *policyDigest*.

The algorithm for computing the new value for *policyDigest* of *policySession* is:

a) Concatenate all the digest values in *pHashList*:

$$digests \coloneqq pHashList.digests[1].buffer \;||\; \ldots \;||\; pHashList.digests[n].buffer \qquad (17)$$

NOTE 1 The TPM will not return an error if the size of an entry is not the same as the size of the digest of the policy. However, that entry cannot match *policyDigest*.

b) Reset *policyDigest* to a Zero Digest.

c) Extend the command code and the hashes computed in step a) above:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \;||\; \text{TPM_CC_PolicyOR} \;||\; digests) \qquad (18)$$

NOTE 2 The computation in b) and c) above is equivalent to:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(0\ldots0 \;||\; \text{TPM_CC_PolicyOR} \;||\; digests)$$

A TPM shall support a list with at least eight tagged digest values.

NOTE 3 If policies are to be portable between TPMs, then they should not use more than eight values.

 Copyright © TCG 2006-2020

23.6.2 Command and Response

Table 130 — TPM2_PolicyOR Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyOR
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPML_DIGEST	pHashList	the list of hashes to check for a match

Table 131 — TPM2_PolicyOR Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

23.6.3 Detailed Actions

[[PolicyOR]]

23.7 TPM2_PolicyPCR

23.7.1 General Description

This command is used to cause conditional gating of a policy based on PCR. This command together with TPM2_PolicyOR() allows one group of authorizations to occur when PCR are in one state and a different set of authorizations when the PCR are in a different state.

The TPM will modify the *pcrs* parameter so that bits that correspond to unimplemented PCR are CLEAR. If *policySession* is not a trial policy session, the TPM will use the modified value of pcrs to select PCR values to hash according to TPM 2.0 Part 1, *Selecting Multiple PCR.* The hash algorithm of the policy session is used to compute a digest (*digestTPM*) of the selected PCR. If *pcrDigest* does not have a length of zero, then it is compared to *digestTPM*; and if the values do not match, the TPM shall return TPM_RC_VALUE and make no change to *policySession→policyDigest*. If the values match, or if the length of *pcrDigest* is zero, then *policySession→policyDigest* is extended by:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM_CC_PolicyPCR} \,||\, pcrs \,||\, digestTPM) \qquad (19)$$

where

$pcrs$	the pcrs parameter with bits corresponding to unimplemented PCR set to 0
$digestTPM$	the digest of the selected PCR using the hash algorithm of the policy session

NOTE 1 If the caller provides the expected PCR value, the intention is that the policy evaluation stop at that point if the PCR do not match. If the caller does not provide the expected PCR value, then the validity of the settings will not be determined until an attempt is made to use the policy for authorization. If the policy is constructed such that the PCR check comes before user authorization checks, this early termination would allow software to avoid unnecessary prompts for user input to satisfy a policy that would fail later due to incorrect PCR values.

After this command completes successfully, the TPM shall return TPM_RC_PCR_CHANGED if the policy session is used for authorization and the PCR are not known to be correct.

The TPM uses a "generation" number (*pcrUpdateCounter*) that is incremented each time PCR are updated (unless the PCR being changed is specified not to cause a change to this counter). The value of this counter is stored in the policy session context (*policySession→pcrUpdateCounter*) when this command is executed. When the policy is used for authorization, the current value of the counter is compared to the value in the policy session context and the authorization will fail if the values are not the same.

When this command is executed, *policySession→pcrUpdateCounter* is checked to see if it has been previously set (in the reference implementation, it has a value of zero if not previously set). If it has been set, it will be compared with the current value of *pcrUpdateCounter* to determine if any PCR changes have occurred. If the values are different, the TPM shall return TPM_RC_PCR_CHANGED.

NOTE 2 Since the pcrUpdateCounter is updated if any PCR is extended (except those specified not to do so), this means that the command will fail even if a PCR not specified in the policy is updated. This is an optimization for the purposes of conserving internal TPM memory. This would be a rare occurrence, and, if this should occur, the policy could be reset using the TPM2_PolicyRestart command and rerun.

If *policySession→pcrUpdateCounter* has not been set, then it is set to the current value of *pcrUpdateCounter*.

If this command is used for a trial *policySession*, *policySession→policyDigest* will be updated using the values from the command rather than the values from a digest of the TPM PCR. If the caller does not provide PCR settings (*pcrDigest* has a length of zero), the TPM may (and it is preferred to) use the

 Copyright © TCG 2006-2020

current TPM PCR settings (*digestTPM)* in the calculation for the new *policyDigest*. The TPM may return an error if the caller does not provide a PCR digest for a trial policy session but this is not the preferred behavior.

The TPM will not check any PCR and will compute:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyPCR} \mathbin{||} pcrs \mathbin{||} pcrDigest) \tag{20}$$

In this computation, pcrs is the input parameter without modification.

NOTE 3 The pcrs parameter is expected to match the configuration of the TPM for which the policy is being computed which may not be the same as the TPM on which the trial policy is being computed.

NOTE 4 Although no PCR are checked in a trial policy session, *pcrDigest* is expected to correspond to some useful PCR values. It is legal, but pointless, to have the TPM aid in calculating a *policyDigest* corresponding to PCR values that are not useful in practice.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.7.2 Command and Response

Table 132 — TPM2_PolicyPCR Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyPCR
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_DIGEST	pcrDigest	expected digest value of the selected PCR using the hash algorithm of the session; may be zero length
TPML_PCR_SELECTION	pcrs	the PCR to include in the check digest

Table 133 — TPM2_PolicyPCR Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

23.7.3 Detailed Actions

[[PolicyPCR]]

Copyright © TCG 2006-2020

23.8 TPM2_PolicyLocality

23.8.1 General Description

This command indicates that the authorization will be limited to a specific locality.

policySession→commandLocality is a parameter kept in the session context. When the policy session is started, this parameter is initialized to a value that allows the policy to apply to any locality.

If *locality* has a value greater than 31, then an extended locality is indicated. For an extended locality, the TPM will validate that *policySession→commandLocality* has not previously been set or that the current value of *policySession→commandLocality* is the same as *locality* (TPM_RC_RANGE).

When *locality* is not an extended locality, the TPM will validate that the *policySession→commandLocality* is not set to an extended locality value (TPM_RC_RANGE). If not the TPM will disable any locality not SET in the *locality* parameter. If the result of disabling localities results in no locality being enabled, the TPM will return TPM_RC_RANGE.

If no error occurred in the validation of *locality*, *policySession→policyDigest* is extended with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyLocality} \mathbin{||} locality) \quad (21)$$

Then *policySession→commandLocality* is updated to indicate which localities are still allowed after execution of TPM2_PolicyLocality().

When the policy session is used to authorize a command, the authorization will fail if the locality used for the command is not one of the enabled localities in *policySession→commandLocality*.

 Copyright © TCG 2006-2020

23.8.2 Command and Response

Table 134 — TPM2_PolicyLocality Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyLocality
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPMA_LOCALITY	locality	the allowed localities for the policy

Table 135 — TPM2_PolicyLocality Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

23.8.3 Detailed Actions

[[PolicyLocality]]

23.9 TPM2_PolicyNV

23.9.1 General Description

This command is used to cause conditional gating of a policy based on the contents of an NV Index. It is an immediate assertion. The NV index is validated during the TPM2_PolicyNV() command, not when the session is used for authorization.

The authorization to read the NV Index must succeed even if *policySession* is a trial policy session.

If *policySession* is a trial policy session, the TPM will update *policySession→policyDigest* as shown in equations (22) and (23) below and return TPM_RC_SUCCESS. It will not perform any further validation. The remainder of this general description would apply only if *policySession* is not a trial policy session.

An authorization session providing authorization to read the NV Index shall be provided.

If TPMA_NV_WRITTEN is not SET in the NV Index, the TPM shall return TPM_RC_NV_UNINITIALIZED. If TPMA_NV_READLOCKED of the NV Index is SET, then the TPM shall return TPM_RC_NV_LOCKED.

For an NV Index with the TPM_NT_COUNTER or TPM_NT_BITS attribute SET, the TPM may ignore the *offset* parameter and use an offset of 0. Therefore, it is recommended that the caller set the *offset* parameter to 0 for interoperability.

If *offset* and the *size* field of *data* add to a value that is greater than the *dataSize* field of the NV Index referenced by *nvIndex*, the TPM shall return an error (TPM_RC_NV_RANGE). The implementation may return an error (TPM_RC_VALUE) if it performs an additional check and determines that *offset* is greater than the *dataSize* field of the NV Index.

operandA begins at *offset* into the NV index contents and has a size equal to the size of *operandB*. The TPM will perform the indicated arithmetic check using *operandA* and *operandB*. If the check fails, the TPM shall return TPM_RC_POLICY and not change *policySession→policyDigest*. If the check succeeds, the TPM will hash the arguments:

$$args \coloneqq \mathbf{H}_{policyAlg}(operandB.buffer \mid\mid offset \mid\mid operation) \quad (22)$$

where

$\mathbf{H}_{policyAlg}()$	hash function using the algorithm of the policy session
$operandB$	the value used for the comparison
$offset$	offset from the start of the NV Index data to start the comparison
$operation$	the operation parameter indicating the comparison being performed

The value of args and the Name of the NV Index are extended to *policySession→policyDigest* by

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mid\mid \text{TPM_CC_PolicyNV} \mid\mid args \mid\mid nvIndex{\rightarrow}Name) \quad (23)$$

where

$\mathbf{H}_{policyAlg}()$	hash function using the algorithm of the policy session
$args$	value computed in equation (22)
$nvIndex{\rightarrow}Name$	the Name of the NV Index

The signed arithmetic operations are performed using twos-compliment.

Magnitude comparisons assume that the octet at offset zero in the referenced NV location and in *operandB* contain the most significant octet of the data.

 Copyright © TCG 2006-2020

23.9.2 Command and Response

Table 136 — TPM2_PolicyNV Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyNV
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index of the area to read Auth Index: None
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_OPERAND	operandB	the second operand
UINT16	offset	the octet offset in the NV Index for the start of operand A
TPM_EO	operation	the comparison to make

Table 137 — TPM2_PolicyNV Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.9.3 Detailed Actions

[[PolicyNV]]

 Copyright © TCG 2006-2020

23.10 TPM2_PolicyCounterTimer

23.10.1 General Description

This command is used to cause conditional gating of a policy based on the contents of the TPMS_TIME_INFO structure.

If *policySession* is a trial policy session, the TPM will update *policySession→policyDigest* as shown in equations (24) and (25) below and return TPM_RC_SUCCESS. It will not perform any validation. The remainder of this general description would apply only if *policySession* is not a trial policy session.

The TPM will perform the indicated arithmetic check on the indicated portion of the TPMS_TIME_INFO structure. If the check fails, the TPM shall return TPM_RC_POLICY and not change *policySession→policyDigest*. If the check succeeds, the TPM will hash the arguments:

$$args \coloneqq \mathbf{H}_{policyAlg}(operandB.buffer \mid\mid offset \mid\mid operation) \tag{24}$$

where

$\mathbf{H}_{policyAlg}()$	hash function using the algorithm of the policy session
$operandB.buffer$	the value used for the comparison
$offset$	offset from the start of the TPMS_TIME_INFO structure at which the comparison starts
$operation$	the operation parameter indicating the comparison being performed

NOTE There is no security related reason for the double hash.

The value of *args* is extended to *policySession→policyDigest* by

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mid\mid \text{TPM_CC_PolicyCounterTimer} \mid\mid args) \tag{25}$$

where

$\mathbf{H}_{policyAlg}()$	hash function using the algorithm of the policy session
$args$	value computed in equation (24)

The signed arithmetic operations are performed using twos-compliment. The indicated portion of the TPMS_TIME_INFO structure begins at *offset* and has a length of *operandB.size*. If the number of octets to be compared overflows the TPMS_TIME_INFO structure, the TPM returns TPM_RC_RANGE. If *offset* is greater than the size of the marshaled TPMS_TIME_INFO structure, the TPM returns TPM_RC_VALUE. The structure is marshaled into its canonical form with no padding. The TPM does not check for alignment of the offset with a TPMS_TIME_INFO structure member.

Magnitude comparisons assume that the octet at offset zero in the referenced location and in *operandB* contain the most significant octet of the data.

 Copyright © TCG 2006-2020

23.10.2 Command and Response

Table 138 — TPM2_PolicyCounterTimer Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyCounterTimer
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_OPERAND	operandB	the second operand
UINT16	offset	the octet offset in the TPMS_TIME_INFO structure for the start of operand A
TPM_EO	operation	the comparison to make

Table 139 — TPM2_PolicyCounterTimer Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.10.3 Detailed Actions

[[PolicyCounterTimer]]

23.11 TPM2_PolicyCommandCode

23.11.1 General Description

This command indicates that the authorization will be limited to a specific command code.

If *policySession→commandCode* has its default value, then it will be set to *code.* If *policySession→commandCode* does not have its default value, then the TPM will return TPM_RC_VALUE if the two values are not the same.

If *code* is not implemented, the TPM will return TPM_RC_POLICY_CC.

If the TPM does not return an error, it will update *policySession→policyDigest* by

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM_CC_PolicyCommandCode} \,||\, code) \qquad (26)$$

NOTE 1 If a previous TPM2_PolicyCommandCode() had been executed, then it is probable that the policy expression is improperly formed but the TPM does not return an error if *code* is the same.

NOTE 2 A TPM2_PolicyOR() would be used to allow an authorization to be used for multiple commands.

When the policy session is used to authorize a command, the TPM will fail the command if the *commandCode* of that command does not match *policySession→commandCode*.

This command, or TPM2_PolicyDuplicationSelect(), is required to enable the policy to be used for ADMIN role authorization.

EXAMPLE Before TPM2_Certify() can be executed, TPM2_PolicyCommandCode() with code set to TPM_CC_Certify is required.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.11.2 Command and Response

Table 140 — TPM2_PolicyCommandCode Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyCommandCode
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM_CC	code	the allowed *commandCode*

Table 141 — TPM2_PolicyCommandCode Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

23.11.3 Detailed Actions

[[PolicyCommandCode]]

 Copyright © TCG 2006-2020

23.12 TPM2_PolicyPhysicalPresence

23.12.1 General Description

This command indicates that physical presence will need to be asserted at the time the authorization is performed.

If this command is successful, *policySession→isPPRequired* will be SET to indicate that this check is required when the policy is used for authorization. Additionally, *policySession→policyDigest* is extended with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM_CC_PolicyPhysicalPresence}) \quad (27)$$

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

23.12.2 Command and Response

Table 142 — TPM2_PolicyPhysicalPresence Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyPhysicalPresence
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None

Table 143 — TPM2_PolicyPhysicalPresence Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.12.3 Detailed Actions

[[PolicyPhysicalPresence]]

23.13 TPM2_PolicyCpHash

23.13.1 General Description

This command is used to allow a policy to be bound to a specific command and command parameters.

TPM2_PolicySigned(), TPM2_PolicySecret(), and TPM2_PolicyTIcket() are designed to allow an authorizing entity to execute an arbitrary command as the *cpHashA* parameter of those commands is not included in *policySession→policyDigest*. TPM2_PolicyCommandCode() allows the policy to be bound to a specific Command Code so that only certain entities may authorize specific command codes. This command allows the policy to be restricted such that an entity may only authorize a command with a specific set of parameters.

If *policySession→cpHash* is already set and not the same as *cpHashA*, then the TPM shall return TPM_RC_CPHASH. If *cpHashA* does not have the size of the *policySession→policyDigest*, the TPM shall return TPM_RC_SIZE.

NOTE 1 If a previous TPM2_PolicyCpHash() had been executed, then it is probable that the policy expression is improperly formed but the TPM does not return an error if *cpHash* is the same.

If the *cpHashA* checks succeed, *policySession→cpHash* is set to *cpHashA* and *policySession→policyDigest* is updated with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyCpHash} \mathbin{||} cpHashA) \tag{28}$$

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.13.2 Command and Response

Table 144 — TPM2_PolicyCpHash Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyCpHash
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_DIGEST	cpHashA	the *cpHash* added to the policy

Table 145 — TPM2_PolicyCpHash Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

23.13.3 Detailed Actions

[[PolicyCpHash]]

 Copyright © TCG 2006-2020

23.14 TPM2_PolicyNameHash

23.14.1 General Description

This command allows a policy to be bound to a specific set of TPM entities without being bound to the parameters of the command. This is most useful for commands such as TPM2_Duplicate() and for TPM2_PCR_Event() when the referenced PCR requires a policy.

The *nameHash* parameter should contain the digest of the Names associated with the handles to be used in the authorized command.

EXAMPLE For the TPM2_Duplicate() command, two handles are provided. One is the handle of the object being duplicated and the other is the handle of the new parent. For that command, *nameHash* would contain:

$$nameHash \coloneqq \mathbf{H}_{policyAlg}(objectHandle{\rightarrow}Name \mid\mid newParentHandle{\rightarrow}Name)$$

If *policySession→cpHash* is already set, the TPM shall return TPM_RC_CPHASH. If the size of *nameHash* is not the size of *policySession→policyDigest*, the TPM shall return TPM_RC_SIZE. Otherwise, *policySession→cpHash* is set to *nameHash*.

If this command completes successfully, the *cpHash* of the authorized command will not be used for validation. Only the digest of the Names associated with the handles in the command will be used.

NOTE 1 This allows the space normally used to hold *policySession→cpHash* to be used for *policySession→nameHash* instead.

The *policySession→policyDigest* will be updated with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mid\mid \text{TPM_CC_PolicyNameHash} \mid\mid nameHash) \quad (29)$$

NOTE 2 This command can only be used with TPM2_PolicyAuthorize() or TPM2_PolicyAuthorizeNV. The owner of the object being duplicated provides approval for their object to be migrated to a specific new parent.

Without this approval, the Name of the Object would need to be known at the time that Object's policy is created. However, since the Name of the Object includes its policy, the Name is not known. The Name can be known by the authorizing entity.

 Copyright © TCG 2006-2020

23.14.2 Command and Response

Table 146 — TPM2_PolicyNameHash Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyNameHash
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_DIGEST	nameHash	the digest to be added to the policy

Table 147 — TPM2_PolicyNameHash Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.14.3 Detailed Actions

[[PolicyNameHash]]

23.15 TPM2_PolicyDuplicationSelect

23.15.1 General Description

This command allows qualification of duplication to allow duplication to a selected new parent.

If this command not used in conjunction with a PolicyAuthorize Command, then only the new parent is selected and *includeObject* should be CLEAR.

EXAMPLE When an object is created when the list of allowed duplication targets is known, the policy would be created with *includeObject* CLEAR.

NOTE 1 Only the new parent may be selected because, without TPM2_PolicyAuthorize(), the Name of the Object to be duplicated would need to be known at the time that Object's policy is created. However, since the Name of the Object includes its policy, the Name is not known. The Name can be known by the authorizing entity (a PolicyAuthorize Command) in which case *includeObject* may be SET.

If used in conjunction with TPM2_PolicyAuthorize(), then the authorizer of the new policy has the option of selecting just the new parent or of selecting both the new parent and the duplication Object.

NOTE 2 If the authorizing entity for an TPM2_PolicyAuthorize() only specifies the new parent, then that authorization may be applied to the duplication of any number of other Objects. If the authorizing entity specifies both a new parent and the duplicated Object, then the authorization only applies to that pairing of Object and new parent.

If either *policySession→cpHash* or *policySession→nameHash* has been previously set, the TPM shall return TPM_RC_CPHASH. Otherwise, *policySession→nameHash* will be set to:

$$nameHash \coloneqq \mathbf{H}_{policyAlg}(objectName.name \; || \; newParentName.name) \quad (30)$$

NOTE 3 It is allowed that policySesion→nameHash and policySession→cpHash share the same memory space.

NOTE 4 The Name in these equations uses Name.name, indicating that the UINT16 size is not included in the hash.

The *policySession→policyDigest* will be updated according to the setting of *includeObject*. If equal to YES, *policySession→policyDigest* is updated by:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \; || \; \text{TPM_CC_PolicyDuplicationSelect} \; || \; objectName.name \; || \; newParentName.name \; || \; includeObject) \quad (31)$$

If includeObject is NO, *policySession→policyDigest* is updated by:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \; || \; \text{TPM_CC_PolicyDuplicationSelect} \; || \; newParentName.name \; || \; includeObject) \quad (32)$$

NOTE 5 *policySession→nameHash* receives the digest of both Names so that the check performed in TPM2_Duplicate() may be the same regardless of which Names are included in *policySession→policyDigest*. This means that, when TPM2_PolicyDuplicationSelect() is executed, it is only valid for a specific pair of duplication object and new parent.

If the command succeeds, *policySession→commandCode* is set to TPM_CC_Duplicate.

NOTE 6 The normal use of this command is before a TPM2_PolicyAuthorize(). An authorized entity would approve a *policyDigest* that allowed duplication to a specific new parent. The authorizing entity may want to limit the authorization so that the approval allows only a specific object to be duplicated to the new parent. In that case, the authorizing entity would approve the *policyDigest* of *equation* (31).

 Copyright © TCG 2006-2020

23.15.2 Command and Response

Table 148 — TPM2_PolicyDuplicationSelect Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyDuplicationSelect
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_NAME	objectName	the Name of the object to be duplicated
TPM2B_NAME	newParentName	the Name of the new parent
TPMI_YES_NO	includeObject	if YES, the *objectName* will be included in the value in *policySession→policyDigest*

Table 149 — TPM2_PolicyDuplicationSelect Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.15.3 Detailed Actions

[[PolicyDuplicationSelect]]

23.16 TPM2_PolicyAuthorize

23.16.1 General Description

This command allows policies to change. If a policy were static, then it would be difficult to add users to a policy. This command lets a policy authority sign a new policy so that it may be used in an existing policy.

The authorizing entity signs a structure that contains

$$aHash \coloneqq \mathbf{H}_{aHashAlg}(approvedPolicy \mid\mid policyRef) \tag{33}$$

The *aHashAlg* is required to be the *nameAlg* of the key used to sign the *aHash*. The *aHash* value is then signed (symmetric or asymmetric) by *keySign*. That signature is then checked by the TPM in 20.1 TPM2_VerifySignature() which produces a ticket by

$$\mathbf{HMAC}(proof, (\text{TPM_ST_VERIFIED} \mid\mid aHash \mid\mid keySign{\rightarrow}Name)) \tag{34}$$

NOTE 1 The reason for the validation is because of the expectation that the policy will be used multiple times and it is more efficient to check a ticket than to load an object each time to check a signature.

The ticket is then used in TPM2_PolicyAuthorize() to validate the parameters.

The *keySign* parameter is required to be a valid object name using nameAlg other than TPM_ALG_NULL. If the first two octets of *keySign* are not a valid hash algorithm, the TPM shall return TPM_RC_HASH. If the remainder of the Name is not the size of the indicated digest, the TPM shall return TPM_RC_SIZE.

The TPM validates that the *approvedPolicy* matches the current value of *policySession→policyDigest* and if not, shall return TPM_RC_VALUE.

The TPM then validates that the parameters to TPM2_PolicyAuthorize() match the values used to generate the ticket. If so, the TPM will reset *policySession→policyDigest* to a Zero Digest. Then it will update *policySession→policyDigest* with **PolicyUpdate**() (see 23.2.3).

$$\mathbf{PolicyUpdate}(\text{TPM_CC_PolicyAuthorize}, keySign, policyRef) \tag{35}$$

If the ticket is not valid, the TPM shall return TPM_RC_POLICY.

If *policySession* is a trial session, *policySession→policyDigest* is extended as if the ticket is valid without actual verification.

NOTE 2 The unmarshaling process requires that a proper TPMT_TK_VERIFIED be provided for *checkTicket* but it may be a NULL Ticket. A NULL ticket is useful in a trial policy, where the caller uses the TPM to perform policy calculations but does not have a valid authorization ticket.

 Copyright © TCG 2006-2020

23.16.2 Command and Response

Table 150 — TPM2_PolicyAuthorize Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyAuthorize
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_DIGEST	approvedPolicy	digest of the policy being approved
TPM2B_NONCE	policyRef	a policy qualifier
TPM2B_NAME	keySign	Name of a key that can sign a policy addition
TPMT_TK_VERIFIED	checkTicket	ticket validating that *approvedPolicy* and *policyRef* were signed by *keySign*

Table 151 — TPM2_PolicyAuthorize Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.16.3 Detailed Actions

[[PolicyAuthorize]]

23.17 TPM2_PolicyAuthValue

23.17.1 General Description

This command allows a policy to be bound to the authorization value of the authorized entity.

When this command completes successfully, *policySession→isAuthValueNeeded* is SET to indicate that the *authValue* will be included in *hmacKey* when the authorization HMAC is computed for the command being authorized using this session. Additionally, *policySession→isPasswordNeeded* will be CLEAR.

NOTE If a policy does not use this command, then the *hmacKey* for the authorized command would only use *sessionKey.* If *sessionKey* is not present, then the *hmacKey* is an Empty Buffer and no HMAC would be computed.

If successful, *policySession→policyDigest* will be updated with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyAuthValue}) \quad (36)$$

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.17.2 Command and Response

Table 152 — TPM2_PolicyAuthValue Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyAuthValue
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None

Table 153 — TPM2_PolicyAuthValue Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

23.17.3 Detailed Actions

[[PolicyAuthValue]]

23.18 TPM2_PolicyPassword

23.18.1 General Description

This command allows a policy to be bound to the authorization value of the authorized object.

When this command completes successfully, *policySession→isPasswordNeeded* is SET to indicate that *authValue* of the authorized object will be checked when the session is used for authorization. The caller will provide the *authValue* in clear text in the *hmac* parameter of the authorization. The comparison of *hmac* to *authValue* is performed as if the authorization is a password.

NOTE 1 The parameter field in the policy session where the authorization value is provided is called *hmac*. If TPM2_PolicyPassword() is part of the sequence, then the field will contain a password and not an HMAC.

If successful, *policySession→policyDigest* will be updated with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyAuthValue}) \quad (37)$$

NOTE 2 This is the same extend value as used with TPM2_PolicyAuthValue so that the evaluation may be done using either an HMAC or a password with no change to the *authPolicy* of the object. The reason that two commands are present is to indicate to the TPM if the *hmac* field in the authorization will contain an HMAC or a password value.

When this command is successful, *policySession→isAuthValueNeeded* will be CLEAR.

23.18.2 Command and Response

Table 154 — TPM2_PolicyPassword Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyPassword
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None

Table 155 — TPM2_PolicyPassword Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.18.3 Detailed Actions

[[PolicyPassword]]

23.19 TPM2_PolicyGetDigest

23.19.1 General Description

This command returns the current *policyDigest* of the session. This command allows the TPM to be used to perform the actions required to pre-compute the *authPolicy* for an object.

 Copyright © TCG 2006-2020

23.19.2 Command and Response

Table 156 — TPM2_PolicyGetDigest Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyGetDigest
TPMI_SH_POLICY	policySession	handle for the policy session Auth Index: None

Table 157 — TPM2_PolicyGetDigest Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	policyDigest	the current value of the *policySession→policyDigest*

 Copyright © TCG 2006-2020

23.19.3 Detailed Actions

[[PolicyGetDigest]]

 Copyright © TCG 2006-2020

23.20 TPM2_PolicyNvWritten

23.20.1 General Description

This command allows a policy to be bound to the TPMA_NV_WRITTEN attributes. This is a deferred assertion. Values are stored in the policy session context and checked when the policy is used for authorization.

If *policySession→checkNVWritten* is CLEAR, it is SET and *policySession→nvWrittenState* is set to *writtenSet*. If *policySession→checkNVWritten* is SET, the TPM will return TPM_RC_VALUE if *policySession→nvWrittenState* and *writtenSet* are not the same.

If the TPM does not return an error, it will update *policySession→policyDigest* by

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyNvWritten} \mathbin{||} writtenSet) \quad (38)$$

When the policy session is used to authorize a command, the TPM will fail the command if *policySession→checkNVWritten* is SET and *nvIndex→attributes→TPMA_NV_WRITTEN does not match policySession→nvWrittenState.*

NOTE 1 A typical use case is a simple policy for the first write during manufacturing provisioning that would require TPMA_NV_WRITTEN CLEAR and a more complex policy for later use that would require TPMA_NV_WRITTEN SET.

NOTE 2 When an Index is written, it has a different authorization name than an Index that has not been written. It is possible to use this change in the NV Index to create a write-once Index.

 Copyright © TCG 2006-2020

23.20.2 Command and Response

Table 158 — TPM2_PolicyNvWritten Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyNvWritten
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPMI_YES_NO	writtenSet	YES if NV Index is required to have been written NO if NV Index is required not to have been written

Table 159 — TPM2_PolicyNvWritten Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.20.3 Detailed Actions

[[PolicyNvWRitten]]

23.21 TPM2_PolicyTemplate

23.21.1 General Description

This command allows a policy to be bound to a specific creation template. This is most useful for an object creation command such as TPM2_Create(), TPM2_CreatePrimary(), or TPM2_CreateLoaded().

The *templateHash* parameter should contain the digest of the template that will be required for the *inPublic* parameter of an Object creation command.

If *policySession→isTemplateHash* is SET and *policySession→cpHash* is not equal to *templateHash,* the TPM shall return TPM_RC_VALUE.

NOTE 1 Revision 01.38 of this specification permitted the TPM to return TPM_RC_CPHASH.

Otherwise, if *policySession→cpHash* is already set, the TPM shall return TPM_RC_CPHASH.

NOTE 2 Revision 01.38 of this specification permitted the TPM to return TPM_RC_VALUE.

If the size of *templateHash* is not the size of *policySession→policyDigest*, the TPM shall return TPM_RC_SIZE. Otherwise, *policySession→cpHash* is set to *templateHash*.

NOTE 3 The digest calculation includes the TPM2B buffer but not the TPM2B size.

If this command completes successfully, the *cpHash* of the authorized command will not be used for validation. Only the digest of the *inPublic* parameter will be used.

NOTE 4 This allows the space normally used to hold *policySession→cpHash* to be used for *policySession→templateHash* instead.

The *policySession→policyDigest* will be updated with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyTemplate} \mathbin{||} templateHash) \qquad (39)$$

 Copyright © TCG 2006-2020

23.21.2 Command and Response

Table 160 — TPM2_PolicyTemplate Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyTemplate
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_DIGEST	templateHash	the digest to be added to the policy

Table 161 — TPM2_PolicyTemplate Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

23.21.3 Detailed Actions

[[PolicyTemplate]]

23.22 TPM2_PolicyAuthorizeNV

23.22.1 General Description

This command provides a capability that is the equivalent of a revocable policy. With TPM2_PolicyAuthorize(), the authorization ticket never expires, so the authorization may not be withdrawn. With this command, the approved policy is kept in an NV Index location so that the policy may be changed as needed to render the old policy unusable.

NOTE 1 This command is useful for Objects but of limited value for other policies that are persistently stored in TPM NV, such as the OwnerPolicy.

An authorization session providing authorization to read the NV Index shall be provided.

The authorization to read the NV Index must succeed even if *policySession* is a trial policy session.

If *policySession* is a trial policy session, the TPM will update *policySession→policyDigest* as shown in equation (40) below and return TPM_RC_SUCCESS. It will not perform any further validation. The remainder of this general description would apply only if *policySession* is not a trial policy session.

NOTE 2 If read access is controlled by policy, the policy should include a branch that authorizes a TPM2_PolicyAuthorizeNV().

If TPMA_NV_WRITTEN is not SET in the Index referenced by *nvIndex*, the TPM shall return TPM_RC_NV_UNINITIALIZED. If TPMA_NV_READLOCKED of the NV Index is SET, then the TPM shall return TPM_RC_NV_LOCKED.

The *dataSize* of the NV Index referenced by *nvIndex* is required to be at least large enough to hold a properly formatted TPMT_HA (TPM_RC_INSUFFICIENT).

NOTE 3 A TPMT_HA contains a TPM_ALG_ID followed a digest that is consistent in size with the hash algorithm indicated by the TPM_ALG_ID.

It is an error (TPM_RC_HASH) if the first two octets of the Index are not a TPM_ALG_ID for a hash algorithm implemented on the TPM or if the indicated hash algorithm does not match *policySession→authHash*.

NOTE 4 The TPM_ALG_ID is stored in the first two octets in big endian format.

The TPM will compare *policySession→policyDigest* to the contents of the NV Index, starting at the first octet after the TPM_ALG_ID (the third octet) and return TPM_RC_VALUE if they are not the same.

NOTE 5 If the Index does not contain enough bytes for the compare, then TPM_RC_INSUFFICENT is generated as indicated above.

NOTE 6 The *dataSize* of the Index may be larger than is required for this command. This permits the Index to include metadata.

If the comparison is successful, the TPM will reset *policySession→policyDigest* to a Zero Digest. Then it will update *policySession→policyDigest* with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_PolicyAuthorizeNV} \mathbin{||} nvIndex{\rightarrow}Name) \quad (40)$$

 Copyright © TCG 2006-2020

23.22.2 Command and Response

Table 162 — TPM2_PolicyAuthorizeNV Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyAuthorizeNV
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index of the area to read Auth Index: None
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None

Table 163 — TPM2_PolicyAuthorizeNV Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

23.22.3 Detailed Actions

[[PolicyAuthorizeNV]]

24 Hierarchy Commands

24.1 TPM2_CreatePrimary

24.1.1 General Description

This command is used to create a Primary Object under one of the Primary Seeds or a Temporary Object under TPM_RH_NULL. The command uses a TPM2B_PUBLIC as a template for the object to be created. The size of the *unique* field shall not be checked for consistency with the other object parameters. The command will create and load a Primary Object. The sensitive area is not returned.

NOTE 1 Since the sensitive data is not returned, the key cannot be reloaded. It can either be made persistent or it can be recreated.

NOTE 2 For interoperability, the *unique* field should not be set to a value that is larger than allowed by object parameters, so that the unmarshaling will not fail.

NOTE 3 An Empty Buffer is a legal *unique* field value.

EXAMPLE 1 A TPM_ALG_RSA object with a *keyBits* of 2048 in the objects parameters should have a *unique* field that is no larger than 256 bytes.

EXAMPLE 2 A TPM_ALG_KEYEDHASH or a TPM_ALG_SYMCIPHER object should have a *unique* field this is no larger than the digest produced by the object's *nameAlg*.

Any type of object and attributes combination that is allowed by TPM2_Create() may be created by this command. The constraints on templates and parameters are the same as TPM2_Create() except that a Primary Storage Key and a Temporary Storage Key are not constrained to use the algorithms of their parents.

For setting of the attributes of the created object, *fixedParent*, *fixedTPM*, decrypt, and restricted are implied to be SET in the parent (a Permanent Handle). The remaining attributes are implied to be CLEAR.

The TPM will derive the object from the Primary Seed indicated in *primaryHandle* using an approved KDF. All of the bits of the template are used in the creation of the Primary Key. Methods for creating a Primary Object from a Primary Seed are described in TPM 2.0 Part 1 and implemented in TPM 2.0 Part 4.

If this command is called multiple times with the same *inPublic* parameter, *inSensitive.data*, and Primary Seed, the TPM shall produce the same Primary Object.

NOTE 4 If the Primary Seed is changed, the Primary Objects generated with the new seed shall be statistically unique even if the parameters of the call are the same.

This command requires authorization. Authorization for a Primary Object attached to the Platform Primary Seed (PPS) shall be provided by *platformAuth* or *platformPolicy*. Authorization for a Primary Object attached to the Storage Primary Seed (SPS) shall be provided by *ownerAuth* or *ownerPolicy*. Authorization for a Primary Key attached to the Endorsement Primary Seed (EPS) shall be provided by *endorsementAuth* or *endorsementPolicy*.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

24.1.2 Command and Response

Table 164 — TPM2_CreatePrimary Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_CreatePrimary
TPMI_RH_HIERARCHY+	@primaryHandle	TPM_RH_ENDORSEMENT, TPM_RH_OWNER, TPM_RH_PLATFORM+{PP}, or TPM_RH_NULL Auth Index: 1 Auth Role: USER
TPM2B_SENSITIVE_CREATE	inSensitive	the sensitive data, see TPM 2.0 Part 1 Sensitive Values
TPM2B_PUBLIC	inPublic	the public template
TPM2B_DATA	outsideInfo	data that will be included in the creation data for this object to provide permanent, verifiable linkage between this object and some object owner data
TPML_PCR_SELECTION	creationPCR	PCR that will be used in creation data

Table 165 — TPM2_CreatePrimary Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM_HANDLE	objectHandle	handle of type TPM_HT_TRANSIENT for created Primary Object
TPM2B_PUBLIC	outPublic	the public portion of the created object
TPM2B_CREATION_DATA	creationData	contains a TPMT_CREATION_DATA
TPM2B_DIGEST	creationHash	digest of *creationData* using *nameAlg* of *outPublic*
TPMT_TK_CREATION	creationTicket	ticket used by TPM2_CertifyCreation() to validate that the creation data was produced by the TPM
TPM2B_NAME	name	the name of the created object

 Copyright © TCG 2006-2020

24.1.3 Detailed Actions

[[CreatePrimary]]

Copyright © TCG 2006-2020

24.2 TPM2_HierarchyControl

24.2.1 General Description

This command enables and disables use of a hierarchy and its associated NV storage. The command allows *phEnable*, *phEnableNV*, *shEnable*, and *ehEnable* to be changed when the proper authorization is provided.

This command may be used to CLEAR *phEnable* and *phEnableNV* if *platformAuth/platformPolicy* is provided. *phEnable* may not be SET using this command.

This command may be used to CLEAR *shEnable* if either *platformAuth/platformPolicy* or *ownerAuth/ownerPolicy* is provided. *shEnable* may be SET if *platformAuth*/*platformPolicy* is provided.

This command may be used to CLEAR *ehEnable* if either *platformAuth*/*platformPolicy* or *endorsementAuth*/*endorsementPolicy* is provided. *ehEnable* may be SET if *platformAuth*/*platformPolicy* is provided.

When this command is used to CLEAR *phEnable*, *shEnable*, or *ehEnable*, the TPM will disable use of any persistent entity associated with the disabled hierarchy and will flush any transient objects associated with the disabled hierarchy.

When this command is used to CLEAR *shEnable,* the TPM will disable access to any NV index that has TPMA_NV_PLATFORMCREATE CLEAR (indicating that the NV Index was defined using Owner Authorization). As long as *shEnable* is CLEAR, the TPM will return an error in response to any command that attempts to operate upon an NV index that has TPMA_NV_PLATFORMCREATE CLEAR.

When this command is used to CLEAR *phEnableNV*, the TPM will disable access to any NV index that has TPMA_NV_PLATFORMCREATE SET (indicating that the NV Index was defined using Platform Authorization). As long as *phEnableNV* is CLEAR, the TPM will return an error in response to any command that attempts to operate upon an NV index that has TPMA_NV_PLATFORMCREATE SET.

 Copyright © TCG 2006-2020

24.2.2 Command and Response

Table 166 — TPM2_HierarchyControl Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_HierarchyControl {NV E}
TPMI_RH_HIERARCHY	@authHandle	TPM_RH_ENDORSEMENT, TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPMI_RH_ENABLES	enable	the enable being modified TPM_RH_ENDORSEMENT, TPM_RH_OWNER, TPM_RH_PLATFORM, or TPM_RH_PLATFORM_NV
TPMI_YES_NO	state	YES if the enable should be SET, NO if the enable should be CLEAR

Table 167 — TPM2_HierarchyControl Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

24.2.3 Detailed Actions

[[HierarchyControl]]

24.3 TPM2_SetPrimaryPolicy

24.3.1 General Description

This command allows setting of the authorization policy for the lockout (*lockoutPolicy*), the platform hierarchy (*platformPolicy*), the storage hierarchy (*ownerPolicy*), and the endorsement hierarchy (*endorsementPolicy*). On TPMs implementing Authenticated Countdown Timers (ACT), this command may also be used to set the authorization policy for an ACT.

The command requires an authorization session. The session shall use the current *authValue* or satisfy the current *authPolicy* for the referenced hierarchy, or the ACT.

The policy that is changed is the policy associated with *authHandle*.

If the enable associated with *authHandle* is not SET, then the associated authorization values (*authValue* or *authPolicy*) may not be used, and the TPM returns TPM_RC_HIERARCHY.

When *hashAlg* is not TPM_ALG_NULL, if the size of *authPolicy* is not consistent with the hash algorithm, the TPM returns TPM_RC_SIZE.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

24.3.2 Command and Response

Table 168 — TPM2_SetPrimaryPolicy Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_SetPrimaryPolicy {NV}
TPMI_RH_HIERARCHY_POLICY	@authHandle	TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_OWNER, TPMI_RH_ACT or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPM2B_DIGEST	authPolicy	an authorization policy digest; may be the Empty Buffer If *hashAlg* is TPM_ALG_NULL, then this shall be an Empty Buffer.
TPMI_ALG_HASH+	hashAlg	the hash algorithm to use for the policy If the *authPolicy* is an Empty Buffer, then this field shall be TPM_ALG_NULL.

Table 169 — TPM2_SetPrimaryPolicy Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

24.3.3 Detailed Actions

[[SetPrimaryPolicy]]

 Copyright © TCG 2006-2020

24.4 TPM2_ChangePPS

24.4.1 General Description

This replaces the current platform primary seed (PPS) with a value from the RNG and sets *platformPolicy* to the default initialization value (the Empty Buffer).

NOTE 1 A policy that is the Empty Buffer can match no policy.

NOTE 2 Platform Authorization is not changed.

All resident transient and persistent objects in the Platform hierarchy are flushed.

Saved contexts in the Platform hierarchy that were created under the old PPS will no longer be able to be loaded.

The policy hash algorithm for PCR is reset to TPM_ALG_NULL.

This command does not clear any NV Index values.

NOTE 3 Index values belonging to the Platform are preserved because the indexes may have configuration information that will be the same after the PPS changes. The Platform may remove the indexes that are no longer needed using TPM2_NV_UndefineSpace().

This command requires Platform Authorization.

24.4.2 Command and Response

Table 170 — TPM2_ChangePPS Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ChangePPS {NV E}
TPMI_RH_PLATFORM	@authHandle	TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER

Table 171 — TPM2_ChangePPS Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

24.4.3 Detailed Actions

[[ChangePPS]]

24.5 TPM2_ChangeEPS

24.5.1 General Description

This replaces the current endorsement primary seed (EPS) with a value from the RNG and sets the Endorsement hierarchy controls to their default initialization values: *ehEnable* is SET, *endorsementAuth* and *endorsementPolicy* are both set to the Empty Buffer. It will flush any resident objects (transient or persistent) in the Endorsement hierarchy and not allow objects in the hierarchy associated with the previous EPS to be loaded.

NOTE In the reference implementation, *ehProof* is a non-volatile value from the RNG. It is allowed that the *ehProof* be generated by a KDF using both the EPS and SPS as inputs. If generated with a KDF, the ehProof can be generated on an as-needed basis or made a non-volatile value.

This command requires Platform Authorization.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

24.5.2 Command and Response

Table 172 — TPM2_ChangeEPS Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ChangeEPS {NV E}
TPMI_RH_PLATFORM	@authHandle	TPM_RH_PLATFORM+{PP} Auth Handle: 1 Auth Role: USER

Table 173 — TPM2_ChangeEPS Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

24.5.3 Detailed Actions

[[ChangeEPS]]

24.6 TPM2_Clear

24.6.1 General Description

This command removes all TPM context associated with a specific Owner.

The clear operation will:

- flush resident objects (persistent and volatile) in the Storage and Endorsement hierarchies;
- delete any NV Index with TPMA_NV_PLATFORMCREATE == CLEAR;
- change the storage primary seed (SPS) to a new value from the TPM's random number generator (RNG),
- change *shProof* and *ehProof*,

NOTE 1 The proof values may be set from the RNG or derived from the associated new Primary Seed. If derived from the Primary Seeds, the derivation of *ehProof* shall use both the SPS and EPS. The computation shall use the SPS as an HMAC key and the derived value may then be a parameter in a second HMAC in which the EPS is the HMAC key. The reference design uses values from the RNG.

- SET *shEnable* and *ehEnable*;
- set *ownerAuth*, *endorsementAuth*, and *lockoutAuth* to the Empty Buffer;
- set *ownerPolicy, endorsementPolicy,* and *lockoutPolicy* to the Empty Buffer;
- set *Clock* to zero;
- set *resetCount* to zero;
- set *restartCount* to zero; and
- set *Safe* to YES.
- increment *pcrUpdateCounter*

NOTE 2 This permits an application to create a policy session that is invalidated on TPM2_Clear. The policy needs, ideally as the first term, TPM2_PolicyPCR(). The session is invalidated even if the PCR selection is empty.

This command requires Platform Authorization or Lockout Authorization. If TPM2_ClearControl() has disabled this command, the TPM shall return TPM_RC_DISABLED.

If this command is authorized using *lockoutAuth*, the HMAC in the response shall use the new *lockoutAuth* value (that is, the Empty Buffer) when computing the response HMAC.

 Copyright © TCG 2006-2020

24.6.2 Command and Response

Table 174 — TPM2_Clear Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Clear {NV E}
TPMI_RH_CLEAR	@authHandle	TPM_RH_LOCKOUT or TPM_RH_PLATFORM+{PP} Auth Handle: 1 Auth Role: USER

Table 175 — TPM2_Clear Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

24.6.3 Detailed Actions

[[Clear]]

24.7 TPM2_ClearControl

24.7.1 General Description

TPM2_ClearControl() disables and enables the execution of TPM2_Clear().

The TPM will SET the TPM's TPMA_PERMANENT.*disableClear* attribute if *disable* is YES and will CLEAR the attribute if *disable* is NO. When the attribute is SET, TPM2_Clear() may not be executed.

NOTE This is to simplify the logic of TPM2_Clear(). TPM2_ClearControl() can be called using Platform Authorization to CLEAR the *disableClear* attribute and then execute TPM2_Clear().

Lockout Authorization may be used to SET *disableClear* but not to CLEAR it.

Platform Authorization may be used to SET or CLEAR *disableClear*.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

24.7.2 Command and Response

Table 176 — TPM2_ClearControl Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ClearControl {NV}
TPMI_RH_CLEAR	@auth	TPM_RH_LOCKOUT or TPM_RH_PLATFORM+{PP} Auth Handle: 1 Auth Role: USER
TPMI_YES_NO	disable	YES if the *disableOwnerClear* flag is to be SET, NO if the flag is to be CLEAR.

Table 177 — TPM2_ClearControl Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

24.7.3 Detailed Actions

[[ClearControl]]

Copyright © TCG 2006-2020

24.8 TPM2_HierarchyChangeAuth

24.8.1 General Description

This command allows the authorization secret for a hierarchy or lockout to be changed using the current authorization value as the command authorization.

If *authHandle* is TPM_RH_PLATFORM, then *platformAuth* is changed. If *authHandle* is TPM_RH_OWNER, then o*wnerAuth* is changed. If *authHandle* is TPM_RH_ENDORSEMENT, then e*ndorsementAuth* is changed. If *authHandle* is TPM_RH_LOCKOUT, then *lockoutAuth* is changed. The HMAC in the response shall use the new authorization value when computing the response HMAC.

If *authHandle* is TPM_RH_PLATFORM, then Physical Presence may need to be asserted for this command to succeed (see 26.2, *TPM2_PP_Commands*).

The authorization value may be no larger than the digest produced by the hash algorithm used for context integrity.

EXAMPLE If SHA384 is used in the computation of the integrity values for saved contexts, then the largest authorization value is 48 octets.

 Copyright © TCG 2006-2020

24.8.2 Command and Response

Table 178 — TPM2_HierarchyChangeAuth Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_HierarchyChangeAuth {NV}
TPMI_RH_HIERARCHY_AUTH	@authHandle	TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPM2B_AUTH	newAuth	new authorization value

Table 179 — TPM2_HierarchyChangeAuth Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

24.8.3 Detailed Actions

[[HierarchyChangeAuth]]

25 Dictionary Attack Functions

25.1 Introduction

A TPM is required to have support for logic that will help prevent a dictionary attack on an authorization value. The protection is provided by a counter that increments when a password authorization or an HMAC authorization fails. When the counter reaches a predefined value, the TPM will not accept, for some time interval, further requests that require authorization and the TPM is in Lockout mode. While the TPM is in Lockout mode, the TPM will return TPM_RC_LOCKOUT if the command requires use of an object's or Index's *authValue* unless the authorization applies to an entry in the Platform hierarchy.

NOTE 1 Authorizations for objects and NV Index values in the Platform hierarchy are never locked out. However, a command that requires multiple authorizations will not be accepted when the TPM is in Lockout mode unless all of the authorizations reference objects and indexes in the Platform hierarchy.

If the TPM is continuously powered for the duration of *newRecoveryTime* and no authorization failures occur, the authorization failure counter will be decremented by one. This property is called "self-healing." Self-healing shall not cause the count of failed attempts to decrement below zero.

The count of failed attempts, the lockout interval, and self-healing interval are settable using TPM2_DictionaryAttackParameters(). The lockout parameters and the current value of the lockout counter can be read with TPM2_GetCapability().

Dictionary attack protection does not apply to an entity associated with a permanent handle (handle type == TPM_HT_PERMANENT) other than TPM_RH_LOCKOUT

25.2 TPM2_DictionaryAttackLockReset

25.2.1 General Description

This command cancels the effect of a TPM lockout due to a number of successive authorization failures. If this command is properly authorized, the lockout counter is set to zero.

Only one *lockoutAuth* authorization failure is allowed for this command during a *lockoutRecovery* interval (set using TPM2_DictionaryAttackParameters().

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

25.2.2 Command and Response

Table 180 — TPM2_DictionaryAttackLockReset Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_DictionaryAttackLockReset {NV}
TPMI_RH_LOCKOUT	@lockHandle	TPM_RH_LOCKOUT Auth Index: 1 Auth Role: USER

Table 181 — TPM2_DictionaryAttackLockReset Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

25.2.3 Detailed Actions

[[DictionaryAttackLockReset]]

 Copyright © TCG 2006-2020

25.3 TPM2_DictionaryAttackParameters

25.3.1 General Description

This command changes the lockout parameters.

The command requires Lockout Authorization.

The timeout parameters (*newRecoveryTime* and *lockoutRecovery*) indicate values that are measured with respect to the *Time* and not *Clock*.

NOTE Use of *Time* means that the TPM shall be continuously powered for the duration of a timeout.

If *newRecoveryTime* is zero, then DA protection is disabled. Authorizations are checked but authorization failures will not cause the TPM to enter lockout.

If *newMaxTries* is zero, the TPM will be in lockout and use of DA protected entities will be disabled.

If *lockoutRecovery* is zero, then the recovery interval is _TPM_Init followed by TPM2_Startup().

Only one *lockoutAuth* authorization failure is allowed for this command during a *lockoutRecovery* interval.

 Copyright © TCG 2006-2020

25.3.2 Command and Response

Table 182 — TPM2_DictionaryAttackParameters Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_DictionaryAttackParameters {NV}
TPMI_RH_LOCKOUT	@lockHandle	TPM_RH_LOCKOUT Auth Index: 1 Auth Role: USER
UINT32	newMaxTries	count of authorization failures before the lockout is imposed
UINT32	newRecoveryTime	time in seconds before the authorization failure count is automatically decremented A value of zero indicates that DA protection is disabled.
UINT32	lockoutRecovery	time in seconds after a *lockoutAuth* failure before use of *lockoutAuth* is allowed A value of zero indicates that a reboot is required.

Table 183 — TPM2_DictionaryAttackParameters Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

25.3.3 Detailed Actions

[[DictionaryAttackParameters]]

 Copyright © TCG 2006-2020

26 Miscellaneous Management Functions

26.1 Introduction

This clause contains commands that do not logically group with any other commands.

26.2 TPM2_PP_Commands

26.2.1 General Description

This command is used to determine which commands require assertion of Physical Presence (PP) in addition to *platformAuth*/*platformPolicy*.

This command requires that *auth* is TPM_RH_PLATFORM and that Physical Presence be asserted.

After this command executes successfully, the commands listed in *setList* will be added to the list of commands that require that Physical Presence be asserted when the handle associated with the authorization is TPM_RH_PLATFORM. The commands in *clearList* will no longer require assertion of Physical Presence in order to authorize a command.

If a command is not in either list, its state is not changed. If a command is in both lists, then it will no longer require Physical Presence (for example, *setList* is processed first).

Only commands with handle types of TPMI_RH_PLATFORM, TPMI_RH_PROVISION, TPMI_RH_CLEAR, or TPMI_RH_HIERARCHY can be gated with Physical Presence. If any other command is in either list, it is discarded.

When a command requires that Physical Presence be provided, then Physical Presence shall be asserted for either an HMAC or a Policy authorization.

NOTE 1 Physical Presence may be made a requirement of any policy.

NOTE 2 If the TPM does not implement this command, the command list is vendor specific. A platform-specific specification may require that the command list be initialized in a specific way.

TPM2_PP_Commands() always requires assertion of Physical Presence.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

26.2.2 Command and Response

Table 184 — TPM2_PP_Commands Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PP_Commands {NV}
TPMI_RH_PLATFORM	@auth	TPM_RH_PLATFORM+PP Auth Index: 1 Auth Role: USER + Physical Presence
TPML_CC	setList	list of commands to be added to those that will require that Physical Presence be asserted
TPML_CC	clearList	list of commands that will no longer require that Physical Presence be asserted

Table 185 — TPM2_PP_Commands Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

26.2.3 Detailed Actions

[[PP_Commands]]

26.3 TPM2_SetAlgorithmSet

26.3.1 General Description

This command allows the platform to change the set of algorithms that are used by the TPM. The *algorithmSet* setting is a vendor-dependent value.

If the changing of the algorithm set results in a change of the algorithms of PCR banks, then the TPM will need to be reset (_TPM_Init and TPM2_Startup(TPM_SU_CLEAR)) before the new PCR settings take effect. After this command executes successfully, if *startupType* in the next TPM2_Startup() is not TPM_SU_CLEAR, the TPM shall return TPM_RC_VALUE and may enter Failure mode.

Other than PCR, when an algorithm is no longer supported, the behavior of this command is vendor-dependent.

EXAMPLE Entities may remain resident. Persistent objects, transient objects, or sessions may be flushed. NV Indexes may be undefined. Policies may be erased.

NOTE The reference implementation does not have support for this command. In particular, it does not support use of this command to selectively disable algorithms. Proper support would require modification of the unmarshaling code so that each time an algorithm is unmarshaled, it would be verified as being enabled.

 Copyright © TCG 2006-2020

26.3.2 Command and Response

Table 186 — TPM2_SetAlgorithmSet Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_SetAlgorithmSet {NV}
TPMI_RH_PLATFORM	@authHandle	TPM_RH_PLATFORM Auth Index: 1 Auth Role: USER
UINT32	algorithmSet	a TPM vendor-dependent value indicating the algorithm set selection

Table 187 — TPM2_SetAlgorithmSet Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

26.3.3 Detailed Actions

[[SetAlgorithmSet]]

27 Field Upgrade

27.1 Introduction

This clause contains the commands for managing field upgrade of the firmware in the TPM. The field upgrade scheme may be used for replacement or augmentation of the firmware installed in the TPM.

EXAMPLE 1 If an algorithm is found to be flawed, a patch of that algorithm might be installed using the firmware upgrade process. The patch might be a replacement of a portion of the code or a complete replacement of the firmware.

EXAMPLE 2 If an additional set of ECC parameters is needed, the firmware process may be used to add the parameters to the TPM data set.

The field upgrade process uses two commands (TPM2_FieldUpgradeStart() and TPM2_FieldUpgradeData()). TPM2_FieldUpgradeStart() validates that a signature on the provided digest is from the TPM manufacturer and that proper authorization is provided using *platformPolicy*.

NOTE 1 The *platformPolicy* for field upgraded is defined by the PM and may include requirements that the upgrade be signed by the PM or the TPM owner and include any other constraints that are desired by the PM.

If the proper authorization is given, the TPM will retain the signed digest and enter the Field Upgrade mode (FUM). While in FUM, the TPM will accept TPM2_FieldUpgradeData() commands. It may accept other commands if it is able to complete them using the previously installed firmware. Otherwise, it will return TPM_RC_UPGRADE.

Each block of the field upgrade shall contain the digest of the next block of the field upgrade data. That digest shall be included in the digest of the previous block. The digest of the first block is signed by the TPM manufacturer. That signature and first block digest are the parameters for TPM2_FieldUpgradeStart(). The digest is saved in the TPM as the required digest for the next field upgrade data block and as the identifier of the field upgrade sequence.

For each field upgrade data block that is sent to the TPM by TPM2_FieldUpgradeData(), the TPM shall validate that the digest matches the required digest and if not, shall return TPM_RC_VALUE. The TPM shall extract the digest of the next expected block and return that value to the caller, along with the digest of the first data block of the update sequence.

The system may attempt to abandon the firmware upgrade by using a zero-length buffer in TPM2_FieldUpdateData(). If the TPM is able to resume operation using the firmware present when the upgrade started, then the TPM will indicate that it has abandon the update by setting the digest of the next block to the Empty Buffer. If the TPM cannot abandon the update, it will return the expected next digest.

The system may also attempt to abandon the update because of a power interruption. If the TPM is able to resume normal operations, then it will respond normally to TPM2_Startup(). If the TPM is not able to resume normal operations, then it will respond to any command but TPM2_FieldUpgradeData() with TPM_RC_UPGRADE.

After a _TPM_Init, system software may not be able to resume the field upgrade that was in process when the power interruption occurred. In such case, the TPM firmware may be reset to one of two other values:

- the original firmware that was installed at the factory (“initial firmware”); or
- the firmware that was in the TPM when the field upgrade process started (“previous firmware”).

The TPM retains the digest of the first block for these firmware images and checks to see if the first block after _TPM_Init matches either of those digests. If so, the firmware update process restarts and the original firmware may be loaded.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

NOTE 2 The TPM is required to accept the previous firmware as either a vendor-provided update or as recovered from the TPM using TPM2_FirmwareRead().

When the last block of the firmware upgrade is loaded into the TPM (indicated to the TPM by data in the data block in a TPM vendor-specific manner), the TPM will complete the upgrade process. If the TPM is able to resume normal operations without a reboot, it will set the hash algorithm of the next block to TPM_ALG_NULL and return TPM_RC_SUCCESS. If a reboot is required, the TPM shall return TPM_RC_REBOOT in response to the last TPM2_FieldUpgradeData() and all subsequent TPM commands until a _TPM_Init is received.

NOTE 3 Because no additional data is allowed when the response code is not TPM_RC_SUCCESS, the TPM returns TPM_RC_SUCCESS for all calls to TPM2_FieldUpgradeData() except the last. In this manner, the TPM is able to indicate the digest of the next block. If a _TPM_Init occurs while the TPM is in FUM, the next block may be the digest for the first block of the original firmware. If it is not, then the TPM will not accept the original firmware until the next _TPM_Init when the TPM is in FUM.

During the field upgrade process, either the one specified in this clause or a vendor proprietary field upgrade process, the TPM should preserve:

- Primary Seeds;
- Hierarchy *authValue*, *authPolicy*, and *proof* values;
- Lockout *authValue* and authorization failure count values;
- PCR authValue and authPolicy values;
- NV Index allocations and contents;
- Persistent object allocations and contents; and
- Clock.

NOTE 4 A platform manufacturer may provide a means to change preserved data to accommodate a case where a field upgrade fixes a flaw that might have compromised TPM secrets.

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

27.2 TPM2_FieldUpgradeStart

27.2.1 General Description

This command uses *platformPolicy* and a TPM Vendor Authorization Key to authorize a Field Upgrade Manifest.

If the signature checks succeed, the authorization is valid and the TPM will accept TPM2_FieldUpgradeData().

This signature is checked against the loaded key referenced by *keyHandle*. This key will have a Name that is the same as a value that is part of the TPM firmware data. If the signature is not valid, the TPM shall return TPM_RC_SIGNATURE.

NOTE A loaded key is used rather than a hard-coded key to reduce the amount of memory needed for this key data in case more than one vendor key is needed.

 Copyright © TCG 2006-2020

27.2.2 Command and Response

Table 188 — TPM2_FieldUpgradeStart Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_FieldUpgradeStart
TPMI_RH_PLATFORM	@authorization	TPM_RH_PLATFORM+{PP} Auth Index:1 Auth Role: ADMIN
TPMI_DH_OBJECT	keyHandle	handle of a public area that contains the TPM Vendor Authorization Key that will be used to validate *manifestSignature* Auth Index: None
TPM2B_DIGEST	fuDigest	digest of the first block in the field upgrade sequence
TPMT_SIGNATURE	manifestSignature	signature over *fuDigest* using the key associated with *keyHandle* (not optional)

Table 189 — TPM2_FieldUpgradeStart Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

27.2.3 Detailed Actions

[[FieldUpgradeStart]]

 Copyright © TCG 2006-2020

27.3 TPM2_FieldUpgradeData

27.3.1 General Description

This command will take the actual field upgrade image to be installed on the TPM. The exact format of *fuData* is vendor-specific. This command is only possible following a successful TPM2_FieldUpgradeStart(). If the TPM has not received a properly authorized TPM2_FieldUpgradeStart(), then the TPM shall return TPM_RC_FIELDUPGRADE.

The TPM will validate that the digest of *fuData* matches an expected value. If so, the TPM may buffer or immediately apply the update. If the digest of *fuData* does not match an expected value, the TPM shall return TPM_RC_VALUE.

 Copyright © TCG 2006-2020

27.3.2 Command and Response

Table 190 — TPM2_FieldUpgradeData Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_FieldUpgradeData {NV}
TPM2B_MAX_BUFFER	fuData	field upgrade image data

Table 191 — TPM2_FieldUpgradeData Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMT_HA+	nextDigest	tagged digest of the next block TPM_ALG_NULL if field update is complete
TPMT_HA	firstDigest	tagged digest of the first block of the sequence

 Copyright © TCG 2006-2020

27.3.3 Detailed Actions

[[FieldUpgradeData]]

 Copyright © TCG 2006-2020

27.4 TPM2_FirmwareRead

27.4.1 General Description

This command is used to read a copy of the current firmware installed in the TPM.

The presumption is that the data will be returned in reverse order so that the last block in the sequence would be the first block given to the TPM in case of a failure recovery. If the TPM2_FirmwareRead sequence completes successfully, then the data provided from the TPM will be sufficient to allow the TPM to recover from an abandoned upgrade of this firmware.

To start the sequence of retrieving the data, the caller sets *sequenceNumber* to zero. When the TPM has returned all the firmware data, the TPM will return the Empty Buffer as *fuData*.

The contents of *fuData* are opaque to the caller.

NOTE 1 The caller should retain the ordering of the update blocks so that the blocks sent to the TPM have the same size and inverse order as the blocks returned by a sequence of calls to this command.

NOTE 2 Support for this command is optional even if the TPM implements TPM2_FieldUpgradeStart() and TPM2_FieldUpgradeData().

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

27.4.2 Command and Response

Table 192 — TPM2_FirmwareRead Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_FirmwareRead
UINT32	sequenceNumber	the number of previous calls to this command in this sequence set to 0 on the first call

Table 193 — TPM2_FirmwareRead Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_MAX_BUFFER	fuData	field upgrade image data

 Copyright © TCG 2006-2020

27.4.3 Detailed Actions

[[FirmwareRead]]

 Copyright © TCG 2006-2020

28 Context Management

28.1 Introduction

Three of the commands in this clause (TPM2_ContextSave(), TPM2_ContextLoad(), and TPM2_FlushContext()) implement the resource management described in the "Context Management" clause in TPM 2.0 Part 1.

The fourth command in this clause (TPM2_EvictControl()) is used to control the persistence of loadable objects in TPM memory. Background for this command may be found in the "Owner and Platform Evict Objects" clause in TPM 2.0 Part 1.

28.2 TPM2_ContextSave

28.2.1 General Description

This command saves a session context, object context, or sequence object context outside the TPM.

No authorization sessions of any type are allowed with this command and tag is required to be TPM_ST_NO_SESSIONS.

NOTE This preclusion avoids complex issues of dealing with the same session in *handle* and in the session area. While it might be possible to provide specificity, it would add unnecessary complexity to the TPM and, because this capability would provide no application benefit, use of authorization sessions for audit or encryption is prohibited.

The TPM shall encrypt and integrity protect the TPM2B_CONTEXT_SENSITIVE *context* as described in the "Context Protections" clause in TPM 2.0 Part 1.

See the “Context Data” clause in TPM 2.0 Part 2 for a description of the *context* structure in the response.

 Copyright © TCG 2006-2020

28.2.2 Command and Response

Table 194 — TPM2_ContextSave Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ContextSave
TPMI_DH_CONTEXT	saveHandle	handle of the resource to save Auth Index: None

Table 195 — TPM2_ContextSave Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMS_CONTEXT	context	

 Copyright © TCG 2006-2020

28.2.3 Detailed Actions

[[ContextSave]]

28.3 TPM2_ContextLoad

28.3.1 General Description

This command is used to reload a context that has been saved by TPM2_ContextSave().

No authorization sessions of any type are allowed with this command and tag is required to be TPM_ST_NO_SESSIONS (see note in 28.2.1).

The TPM will return TPM_RC_HIERARCHY if the context is associated with a hierarchy that is disabled.

NOTE Contexts for authorization sessions and for sequence objects belong to the NULL hierarchy, which is never disabled.

See the "Context Data" clause in TPM 2.0 Part 2 for a description of the values in the *context* parameter.

If the integrity HMAC of the saved context is not valid, the TPM shall return TPM_RC_INTEGRITY.

The TPM shall perform a check on the decrypted context as described in the "Context Confidentiality Protection" clause of TPM 2.0 Part 1 and enter failure mode if the check fails.

 Copyright © TCG 2006-2020

28.3.2 Command and Response

Table 196 — TPM2_ContextLoad Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ContextLoad
TPMS_CONTEXT	context	the context blob

Table 197 — TPM2_ContextLoad Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_DH_CONTEXT	loadedHandle	the handle assigned to the resource after it has been successfully loaded

 Copyright © TCG 2006-2020

28.3.3 Detailed Actions

[[ContextLoad]]

 Copyright © TCG 2006-2020

28.4 TPM2_FlushContext

28.4.1 General Description

This command causes all context associated with a loaded object, sequence object, or session to be removed from TPM memory.

This command may not be used to remove a persistent object from the TPM. Use TPM2_EvictControl to remove a persistent object.

A session does not have to be loaded in TPM memory to have its context flushed. The saved session context associated with the indicated handle is invalidated. When flushing a session, the upper byte of the handle is ignored.

EXAMPLE A command to flush session handle 0x20000000 will flush session handle 0x03000000.

No sessions of any type are allowed with this command and tag is required to be TPM_ST_NO_SESSIONS (see note in 28.2.1).

If the handle is for a Transient Object and the handle is not associated with a loaded object, then the TPM shall return TPM_RC_HANDLE.

If the handle is for an authorization session and the handle does not reference a loaded or active session, then the TPM shall return TPM_RC_HANDLE.

NOTE *flushHandle* is a parameter and not a handle. If it were in the handle area, the TPM would validate that the context for the referenced entity is in the TPM. When a TPM2_FlushContext references a saved session context, it is not necessary for the context to be in the TPM. When the *flushHandle* is in the parameter area, the TPM does not validate that associated context is actually in the TPM.

 Copyright © TCG 2006-2020

28.4.2 Command and Response

Table 198 — TPM2_FlushContext Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_FlushContext
TPMI_DH_CONTEXT	flushHandle	the handle of the item to flush NOTE This is a use of a handle as a parameter.

Table 199 — TPM2_FlushContext Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

28.4.3 Detailed Actions

[[FlushContext]]

28.5 TPM2_EvictControl

28.5.1 General Description

This command allows certain Transient Objects to be made persistent or a persistent object to be evicted.

NOTE 1 A transient object is one that may be removed from TPM memory using either TPM2_FlushContext or TPM2_Startup(). A persistent object is not removed from TPM memory by TPM2_FlushContext() or TPM2_Startup().

If *objectHandle* is a Transient Object, then this call makes a persistent copy of the object and assigns *persistentHandle* to the persistent version of the object. If *objectHandle* is a persistent object, then the call evicts the persistent object. The call does not affect the transient object.

Before execution of TPM2_EvictControl code below, the TPM verifies that *objectHandle* references an object that is resident on the TPM and that *persistentHandle* is a valid handle for a persistent object.

NOTE 2 This requirement simplifies the unmarshaling code so that it only need check that *persistentHandle* is always a persistent object.

If *objectHandle* references a Transient Object:

a) The TPM shall return TPM_RC_ATTRIBUTES if
   1) it is in the hierarchy of TPM_RH_NULL,
   2) only the public portion of the object is loaded, or

      NOTE 3 This is for NV space efficiency. Loading an object whose private part is empty would unnecessarily consume NV resources.

   3) the *stClear* is SET in the object or in an ancestor key.
b) The TPM shall return TPM_RC_HIERARCHY if the object is not in the proper hierarchy as determined by *auth*.
   1) If *auth* is TPM_RH_PLATFORM, the proper hierarchy is the Platform hierarchy.
   2) If *auth* is TPM_RH_OWNER, the proper hierarchy is either the Storage or the Endorsement hierarchy.
c) The TPM shall return TPM_RC_RANGE if *persistentHandle* is not in the proper range as determined by *auth*.
   1) If *auth* is TPM_RH_OWNER, then *persistentHandle* shall be in the inclusive range of $81\ 00\ 00\ 00_{16}$ to $81\ 7F\ FF\ FF_{16}$.
   2) If *auth* is TPM_RH_PLATFORM, then *persistentHandle* shall be in the inclusive range of $81\ 80\ 00\ 00_{16}$ to $81\ FF\ FF\ FF_{16}$.

   NOTE 4 This separation permits the platform (the platform OEM) a range of indexes that will not interfere with indexes used by the TPM owner (the OS or applications).

d) The TPM shall return TPM_RC_NV_DEFINED if a persistent object exists with the same handle as *persistentHandle*.
e) The TPM shall return TPM_RC_NV_SPACE if insufficient space is available to make the object persistent.
f) The TPM shall return TPM_RC_NV_SPACE if execution of this command will prevent the TPM from being able to hold two transient objects of any kind.

 Copyright © TCG 2006-2020

NOTE 5 This requirement anticipates that a TPM may be implemented such that all TPM memory is non-volatile and not subject to endurance issues. In such case, there is no movement of an object between memory of different types and it is necessary that the TPM ensure that it is always possible for the management software to move objects to/from TPM memory in order to ensure that the objects required for command execution can be context restored.

g) If the TPM returns TPM_RC_SUCCESS, the object referenced by *objectHandle* will not be flushed and both *objectHandle* and *persistentHandle* may be used to access the object.

If *objectHandle* references a persistent object:

a) The TPM shall return TPM_RC_RANGE if *objectHandle* is not in the proper range as determined by *auth*. If *auth* is TPM_RC_OWNER, *objectHandle* shall be in the inclusive range of 81 00 00 00_{16} to 81 7F FF FF_{16}. If *auth* is TPM_RC_PLATFORM, *objectHandle* may be any valid persistent object handle.

b) If objectHandle is not the same value as persistentHandle, return TPM_RC_HANDLE.

c) If the TPM returns TPM_RC_SUCCESS, *objectHandle* will be removed from persistent memory and no longer be accessible.

NOTE 5 The persistent object is not converted to a transient object, as this would prevent the immediate revocation of an object by removing it from persistent memory.

 Copyright © TCG 2006-2020

28.5.2 Command and Response

Table 200 — TPM2_EvictControl Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_EvictControl {NV}
TPMI_RH_PROVISION	@auth	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Handle: 1 Auth Role: USER
TPMI_DH_OBJECT	objectHandle	the handle of a loaded object Auth Index: None
TPMI_DH_PERSISTENT	persistentHandle	if *objectHandle* is a transient object handle, then this is the persistent handle for the object if *objectHandle* is a persistent object handle, then it shall be the same value as *persistentHandle*

Table 201 — TPM2_EvictControl Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

28.5.3 Detailed Actions

[[EvictControl]]

29 Clocks and Timers

29.1 TPM2_ReadClock

29.1.1 General Description

This command reads the current TPMS_TIME_INFO structure that contains the current setting of *Time*, *Clock*, *resetCount*, and *restartCount*.

 Copyright © TCG 2006-2020

29.1.2 Command and Response

Table 202 — TPM2_ReadClock Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ReadClock

Table 203 — TPM2_ReadClock Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMS_TIME_INFO	currentTime	

 Copyright © TCG 2006-2020

29.1.3 Detailed Actions

[[ReadClock]]

29.2 TPM2_ClockSet

29.2.1 General Description

This command is used to advance the value of the TPM's *Clock*. The command will fail if *newTime* is less than the current value of *Clock* or if the new time is greater than FF FF 00 00 00 00 00 00_{16}. If both of these checks succeed, *Clock* is set to *newTime*. If either of these checks fails, the TPM shall return TPM_RC_VALUE and make no change to *Clock*.

NOTE This maximum setting would prevent *Clock* from rolling over to zero for approximately 8,000 years at the real time *Clock* update rate. If the *Clock* update rate was set so that TPM time was passing 33 percent faster than real time, it would still be more than 6,000 years before *Clock* would roll over to zero. Because *Clock* will not roll over in the lifetime of the TPM, there is no need for external software to deal with the possibility that *Clock* may wrap around.

If the value of *Clock* after the update makes the volatile and non-volatile versions of TPMS_CLOCK_INFO.*clock* differ by more than the reported update interval, then the TPM shall update the non-volatile version of TPMS_CLOCK_INFO.*clock* before returning.

This command requires Platform Authorization or Owner Authorization.

 Copyright © TCG 2006-2020

29.2.2 Command and Response

Table 204 — TPM2_ClockSet Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ClockSet {NV}
TPMI_RH_PROVISION	@auth	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Handle: 1 Auth Role: USER
UINT64	newTime	new *Clock* setting in milliseconds

Table 205 — TPM2_ClockSet Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

29.2.3 Detailed Actions

[[ClockSet]]

29.3 TPM2_ClockRateAdjust

29.3.1 General Description

This command adjusts the rate of advance of *Clock* and *Time* to provide a better approximation to real time.

The *rateAdjust* value is relative to the current rate and not the nominal rate of advance.

EXAMPLE 1 If this command had been called three times with *rateAdjust* = TPM_CLOCK_COARSE_SLOWER and once with *rateAdjust* = TPM_CLOCK_COARSE_FASTER, the net effect will be as if the command had been called twice with *rateAdjust* = TPM_CLOCK_COARSE_SLOWER.

The range of adjustment shall be sufficient to allow *Clock* and *Time* to advance at real time but no more. If the requested adjustment would make the rate advance faster or slower than the nominal accuracy of the input frequency, the TPM shall return TPM_RC_VALUE.

EXAMPLE 2 If the frequency tolerance of the TPM's input clock is +/-10 percent, then the TPM will return TPM_RC_VALUE if the adjustment would make *Clock* run more than 10 percent faster or slower than nominal. That is, if the input oscillator were nominally 100 megahertz (MHz), then 1 millisecond (ms) would normally take 100,000 counts. The update *Clock* should be adjustable so that 1 ms is between 90,000 and 110,000 counts.

The interpretation of “fine” and “coarse” adjustments is implementation-specific.

The nominal rate of advance for *Clock* and *Time* shall be accurate to within 15 percent. That is, with no adjustment applied, *Clock* and *Time* shall be advanced at a rate within 15 percent of actual time.

NOTE If the adjustments are incorrect, it will be possible to make the difference between advance of *Clock/Time* and real time to be as much as 1.15^2 or ~1.33.

Changes to the current *Clock* update rate adjustment need not be persisted across TPM power cycles.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

29.3.2 Command and Response

Table 206 — TPM2_ClockRateAdjust Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ClockRateAdjust
TPMI_RH_PROVISION	@auth	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Handle: 1 Auth Role: USER
TPM_CLOCK_ADJUST	rateAdjust	Adjustment to current *Clock* update rate

Table 207 — TPM2_ClockRateAdjust Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

29.3.3 Detailed Actions

[[ClockRateAdjust]]

 Copyright © TCG 2006-2020

30 Capability Commands

30.1 Introduction

The TPM has numerous values that indicate the state, capabilities, and properties of the TPM. These values are needed for proper management of the TPM. The TPM2_GetCapability() command is used to access these values.

TPM2_GetCapability() allows reporting of multiple values in a single call. The values are grouped according to type.

NOTE TPM2_TestParms()is used to determine if a TPM supports a particular combination of algorithm parameters

30.2 TPM2_GetCapability

30.2.1 General Description

This command returns various information regarding the TPM and its current state.

The *capability* parameter determines the category of data returned. The *property* parameter selects the first value of the selected category to be returned. If there is no property that corresponds to the value of *property*, the next higher value is returned, if it exists.

EXAMPLE 1 The list of handles of transient objects currently loaded in the TPM may be read one at a time. On the first read, set the property to TRANSIENT_FIRST and *propertyCount* to one. If a transient object is present, the lowest numbered handle is returned and *moreData* will be YES if transient objects with higher handles are loaded. On the subsequent call, use returned handle value plus 1 in order to access the next higher handle.

The *propertyCount* parameter indicates the number of capabilities in the indicated group that are requested. The TPM will return no more than the number of requested values (*propertyCount*) or until the last property of the requested type has been returned.

NOTE 1 The type of the capability is derived from a combination of *capability* and *property*.

NOTE 2 If the *property* selects an unimplemented property, the next higher implemented property is returned.

When all of the properties of the requested type have been returned, the *moreData* parameter in the response will be set to NO. Otherwise, it will be set to YES.

NOTE 3 The *moreData* parameter will be YES if there are more properties even if the requested number of capabilities has been returned.

The TPM is not required to return more than one value at a time. It is not required to provide the same number of values in response to subsequent requests.

EXAMPLE 2 A TPM may return 4 properties in response to a TPM2_GetCapability(*capability* = TPM_CAP_TPM_PROPERTY, *property* = TPM_PT_MANUFACTURER, *propertyCount* = 8) and for a latter request with the same parameters, the TPM may return as few as one and as many as 8 values.

When the TPM is in Failure mode, a TPM is required to allow use of this command for access of the following capabilities:

 Copyright © TCG 2006-2020

- TPM_PT_MANUFACTURER
- TPM_PT_VENDOR_STRING_1
- TPM_PT_VENDOR_STRING_2 (NOTE 4)
- TPM_PT_VENDOR_STRING_3 (NOTE 4)
- TPM_PT_VENDOR_STRING_4 (NOTE 4)
- TPM_PT_VENDOR_TPM_TYPE
- TPM_PT_FIRMWARE_VERSION_1
- TPM_PT_FIRMWARE_VERSION_2

NOTE 4 If the vendor string does not require one of these values, the property type does not need to exist.

A vendor may optionally allow the TPM to return other values.

If in Failure mode and a capability is requested that is not available in Failure mode, the TPM shall return no value.

EXAMPLE 3 Assume the TPM is in Failure mode and the TPM only supports reporting of the minimum required set of properties (the limited subset of TPML_TAGGED_TPM_PROPERTY values). If a TPM2_GetCapability is received requesting a capability that has a property type value greater than TPM_PT_FIRMWARE_VERSION_2, the TPM may return a zero length list with the moreData parameter set to NO or return the property TPM_PT_FIRMWARE_VERSION_2. If the property type is less than TPM_PT_MANUFACTURER, the TPM will return properties beginning with TPM_PT_MANUFACTURER.

In Failure mode, *tag* is required to be TPM_ST_NO_SESSIONS or the TPM shall return TPM_RC_FAILURE.

The capability categories and the types of the return values are:

capability	***property***	**Return Type**
TPM_CAP_ALGS	TPM_ALG_ID(1)	TPML_ALG_PROPERTY
TPM_CAP_HANDLES	TPM_HANDLE	TPML_HANDLE
TPM_CAP_COMMANDS	TPM_CC	TPML_CCA
TPM_CAP_PP_COMMANDS	TPM_CC	TPML_CC
TPM_CAP_AUDIT_COMMANDS	TPM_CC	TPML_CC
TPM_CAP_PCRS	Reserved	TPML_PCR_SELECTION
TPM_CAP_TPM_PROPERTIES	TPM_PT	TPML_TAGGED_TPM_PROPERTY
TPM_CAP_PCR_PROPERTIES	TPM_PT_PCR	TPML_TAGGED_PCR_PROPERTY
TPM_CAP_ECC_CURVES	TPM_ECC_CURVE(1)	TPML_ECC_CURVE
TPM_CAP_AUTH_POLICIES (3)	TPM_HANDLE(2)	TPML_TAGGED_POLICY
TPM_CAP_ACT(4)	TPM_HANDLE(2)	TPML_ACT_DATA
TPM_CAP_VENDOR_PROPERTY	manufacturer specific	manufacturer-specific values

NOTES:
(1) The TPM_ALG_ID or TPM_ECC_CURVE is cast to a UINT32
(2) The TPM will return TPM_RC_VALUE if the handle does not reference the range for permanent handles.
(3) TPM_CAP_AUTH_POLICIES was added in revision 01.32.
(4) TPM_CAP_ACT was added in revision 01.56.

 Copyright © TCG 2006-2020

- TPM_CAP_ALGS – Returns a list of TPMS_ALG_PROPERTIES. Each entry is an algorithm ID and a set of properties of the algorithm.
- TPM_CAP_HANDLES – Returns a list of all of the handles within the handle range of the *property* parameter. The range of the returned handles is determined by the handle type (the most-significant octet (MSO) of the *property)*. Any of the defined handle types is allowed

EXAMPLE 4 If the MSO of *property* is TPM_HT_NV_INDEX, then the TPM will return a list of NV Index values.

EXAMPLE 5 If the MSO of property is TPM_HT_PCR, then the TPM will return a list of PCR.

- For this capability, use of TPM_HT_LOADED_SESSION and TPM_HT_SAVED_SESSION is allowed. Requesting handles with a handle type of TPM_HT_LOADED_SESSION will return handles for loaded sessions. The returned handle values will have a handle type of either TPM_HT_HMAC_SESSION or TPM_HT_POLICY_SESSION. If saved sessions are requested, all returned values will have the TPM_HT_HMAC_SESSION handle type because the TPM does not track the session type of saved sessions.

NOTE 5 TPM_HT_LOADED_SESSION and TPM_HT_HMAC_SESSION have the same value, as do TPM_HT_SAVED_SESSION and TPM_HT_POLICY_SESSION. It is not possible to request that the TPM return a list of loaded HMAC sessions without including the policy sessions.

- TPM_CAP_COMMANDS – Returns a list of the command attributes for all of the commands implemented in the TPM, starting with the TPM_CC indicated by the *property* parameter. If vendor specific commands are implemented, the vendor-specific command attribute with the lowest *commandIndex*, is returned after the non-vendor-specific (base) command.

NOTE 6 The type of the property parameter is a TPM_CC while the type of the returned list is TPML_CCA.

- TPM_CAP_PP_COMMANDS – Returns a list of all of the commands currently requiring Physical Presence for confirmation of platform authorization. The list will start with the TPM_CC indicated by *property*.
- TPM_CAP_AUDIT_COMMANDS – Returns a list of all of the commands currently set for command audit.
- TPM_CAP_PCRS – Returns the current allocation of PCR in a TPML_PCR_SELECTION. The *property* parameter shall be zero. The TPM will always respond to this command with the full PCR allocation and *moreData* will be NO.

  The TPML_PCR_SELECTION must include a TPMS_PCR_SELECTION for each PCR bank in which there is at least one allocated PCR. The TPML_PCR_SELECTION may return a TPMS_PCR_SELECTION for each implemented PCR bank. The TPML_PCR_SELECTION may return a TPMS_PCR_SELECTION for each implemented hash algorithm.
- TPM_CAP_TPM_PROPERTIES – Returns a list of tagged properties. The tag is a TPM_PT and the property is a 32-bit value. The properties are returned in groups. Each property group is on a 256-value boundary (that is, the boundary occurs when the TPM_PT is evenly divisible by 256). The TPM will only return values in the same group as the *property* parameter in the command.
- TPM_CAP_PCR_PROPERTIES – Returns a list of tagged PCR properties. The tag is a TPM_PT_PCR and the property is a TPMS_PCR_SELECT.

  The input command property is a TPM_PT_PCR (see TPM 2.0 Part 2 for PCR properties to be requested) that specifies the first property to be returned. If propertyCount is greater than 1, the list of properties begins with that property and proceeds in TPM_PT_PCR sequence.

  Each item in the list is a TPMS_PCR_SELECT structure that contains a bitmap of all PCR.

NOTE 7 A PCR index in all banks (all hash algorithms) has the same properties, so the hash algorithm is not specified here.

 Copyright © TCG 2006-2020

- TPM_CAP_TPM_ECC_CURVES – Returns a list of ECC curve identifiers currently available for use in the TPM.
- TPM_CAP_AUTH_POLICIES - Returns a list of tagged policies reporting the authorization policies for the permanent handles.
- TPM_CAP_ACT – Returns a list of TPMS_ACT_DATA, each of which contains the handle for the ACT, the remaining time before it expires, and the ACT attributes.

The *moreData* parameter will have a value of YES if there are more values of the requested type that were not returned.

If no next capability exists, the TPM will return a zero-length list and *moreData* will have a value of NO.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

30.2.2 Command and Response

Table 208 — TPM2_GetCapability Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_GetCapability
TPM_CAP	capability	group selection; determines the format of the response
UINT32	property	further definition of information
UINT32	propertyCount	number of properties of the indicated type to return

Table 209 — TPM2_GetCapability Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_YES_NO	moreData	flag to indicate if there are more values of this type
TPMS_CAPABILITY_DATA	capabilityData	the capability data

 Copyright © TCG 2006-2020

30.2.3 Detailed Actions

[[GetCapability]]

 Copyright © TCG 2006-2020

30.3 TPM2_TestParms

30.3.1 General Description

This command is used to check to see if specific combinations of algorithm parameters are supported.

The TPM will unmarshal the provided TPMT_PUBLIC_PARMS. If the parameters unmarshal correctly, then the TPM will return TPM_RC_SUCCESS, indicating that the parameters are valid for the TPM. The TPM will return the appropriate unmarshaling error if a parameter is not valid.

30.3.2 Command and Response

Table 210 — TPM2_TestParms Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_TestParms
TPMT_PUBLIC_PARMS	parameters	algorithm parameters to be validated

Table 211 — TPM2_TestParms Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	TPM_RC

 Copyright © TCG 2006-2020

30.3.3 Detailed Actions

[[TestParms]]

31 Non-volatile Storage

31.1 Introduction

The NV commands are used to create, update, read, and delete allocations of space in NV memory. Before an Index may be used, it must be defined (TPM2_NV_DefineSpace()).

An Index may be modified if the proper write authorization is provided or read if the proper read authorization is provided. Different controls are available for reading and writing.

An Index may have an Index-specific *authValue* and *authPolicy*. The *authValue* may be used to authorize reading if TPMA_NV_AUTHREAD is SET and writing if TPMA_NV_AUTHWRITE is SET. The *authPolicy* may be used to authorize reading if TPMA_NV_POLICYREAD is SET and writing if TPMA_NV_POLICYWRITE is SET.

For commands that have both *authHandle* and *nvIndex* parameters, *authHandle* can be an NV Index, Platform Authorization, or Owner Authorization. If *authHandle* is an NV Index, it must be the same as *nvIndex* (TPM_RC_NV_AUTHORIZATION).

TPMA_NV_PPREAD and TPMA_NV_PPWRITE indicate if reading or writing of the NV Index may be authorized by *platformAuth* or *platformPolicy*.

TPMA_NV_OWNERREAD and TPMA_NV_OWNERWRITE indicate if reading or writing of the NV Index may be authorized by *ownerAuth* or *ownerPolicy*.

If an operation on an NV index requires authorization, and the authHandle parameter is the handle of an NV Index, then the nvIndex parameter must have the same value or the TPM will return TPM_RC_NV_AUTHORIZATION.

NOTE 1 This check ensures that the authorization that was provided is associated with the NV Index being authorized.

For creating an Index, Owner Authorization may not be used if *shEnable* is CLEAR and Platform Authorization may not be used if *phEnableNV* is CLEAR.

If an Index was defined using Platform Authorization, then that Index is not accessible when *phEnableNV* is CLEAR. If an Index was defined using Owner Authorization, then that Index is not accessible when *shEnable* is CLEAR.

For read access control, any combination of TPMA_NV_PPREAD, TPMA_NV_OWNERREAD, TPMA_NV_AUTHREAD, or TPMA_NV_POLICYREAD is allowed as long as at least one is SET.

For write access control, any combination of TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, TPMA_NV_AUTHWRITE, or TPMA_NV_POLICYWRITE is allowed as long as at least one is SET.

If an Index has been defined and not written, then any operation on the NV Index that requires read authorization will fail (TPM_RC_NV_INITIALIZED). This check may be made before or after other authorization checks but shall be performed before checking the NV Index *authValue*. An authorization failure due to the NV Index not having been written shall not be logged by the dictionary attack logic.

If TPMA_NV_CLEAR_STCLEAR is SET, then the TPMA_NV_WRITTEN will be CLEAR on each TPM2_Startup(TPM_SU_CLEAR). TPMA_NV_CLEAR_STCLEAR shall not be SET if the *nvIndexType* is TPM_NT_COUNTER.

The code in the “Detailed Actions” clause of each command is written to interface with an implementation-dependent library that allows access to NV memory. The actions assume no specific layout of the structure of the NV data.

Only one NV Index may be directly referenced in a command.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

NOTE 2 This means that, if *authHandle* references an NV Index, then *nvIndex* will have the same value. However, this does not limit the number of changes that may occur as side effects. For example, any number of NV Indexes might be relocated as a result of deleting or adding a NV Index.

 Copyright © TCG 2006-2020

31.2 NV Counters

When an Index has the TPM_NT_COUNTER attribute, it behaves as a monotonic counter and may only be updated using TPM2_NV_Increment().

When an NV counter is created, the TPM shall initialize the 8-octet counter value with a number that is greater than any count value for any NV counter on the TPM since the time of TPM manufacture.

An NV counter may be defined with the TPMA_NV_ORDERLY attribute to indicate that the NV Index is expected to be modified at a high frequency and that the data is only required to persist when the TPM goes through an orderly shutdown process. The TPM may update the counter value in RAM and occasionally update the non-volatile version of the counter. An orderly shutdown is one occasion to update the non-volatile count. If the difference between the volatile and non-volatile version of the counter becomes as large as MAX_ORDERLY_COUNT, this shall be another occasion for updating the non-volatile count.

Before an NV counter can be used, the TPM shall validate that the count is not less than a previously reported value. If the TPMA_NV_ORDERLY attribute is not SET, or if the TPM experienced an orderly shutdown, then the count is assumed to be correct. If the TPMA_NV_ORDERLY attribute is SET, and the TPM shutdown was not orderly, then the TPM shall OR MAX_ORDERLY_COUNT to the contents of the non-volatile counter and set that as the current count.

NOTE 1 Because the TPM would have updated the NV Index if the difference between the count values was equal to MAX_ORDERLY_COUNT + 1, the highest value that could have been in the NV Index is MAX_ORDERLY_COUNT so it is safe to restore that value.

NOTE 2 The TPM may implement the RAM portion of the counter such that the effective value of the NV counter is the sum of both the volatile and non-volatile parts. If so, then the TPM may initialize the RAM version of the counter to MAX_ORDERLY_COUNT and no update of NV is necessary.

NOTE 3 When a new NV counter is created, the TPM may search all the counters to determine which has the highest value. In this search, the TPM would use the sum of the non-volatile and RAM portions of the counter. The RAM portion of the counter shall be properly initialized to reflect shutdown process (orderly or not) of the TPM.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.3 TPM2_NV_DefineSpace

31.3.1 General Description

This command defines the attributes of an NV Index and causes the TPM to reserve space to hold the data associated with the NV Index. If a definition already exists at the NV Index, the TPM will return TPM_RC_NV_DEFINED.

The TPM will return TPM_RC_ATTRIBUTES if *nvIndexType* has a reserved value in *publicInfo.*

NOTE 1 It is not required that any of these three attributes be set.

The TPM shall return TPM_RC_ATTRIBUTES if TPMA_NV_WRITTEN, TPMA_NV_READLOCKED, or TPMA_NV_WRITELOCKED is SET.

If *nvIndexType* is TPM_NT_COUNTER, TPM_NT_BITS, TPM_NT_PIN_FAIL, or TPM_NT_PIN_PASS, then *publicInfo→dataSize* shall be set to eight (8) or the TPM shall return TPM_RC_SIZE.

If *nvIndexType* is TPM_NT_EXTEND, then *publicInfo→dataSize* shall match the digest size of the *publicInfo.nameAlg* or the TPM shall return TPM_RC_SIZE.

NOTE 2 TPM_RC_ATTRIBUTES could be returned by a TPM that is based on the reference code of older versions of the specification but the correct response for this error is TPM_RC_SIZE.

If the NV Index is an ordinary Index and *publicInfo→dataSize* is larger than supported by the TPM implementation then the TPM shall return TPM_RC_SIZE.

NOTE 3 The limit for the data size may vary according to the type of the index. For example, if the index has TPMA_NV_ORDERLY SET, then the maximum size of an ordinary NV Index may be less than the size of an ordinary NV Index that has TPMA_NV_ORDERLY CLEAR.

At least one of TPMA_NV_PPREAD, TPMA_NV_OWNERREAD, TPMA_NV_AUTHREAD, or TPMA_NV_POLICYREAD shall be SET or the TPM shall return TPM_RC_ATTRIBUTES.

At least one of TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, TPMA_NV_AUTHWRITE, or TPMA_NV_POLICYWRITE shall be SET or the TPM shall return TPM_RC_ATTRIBUTES.

If TPMA_NV_CLEAR_STCLEAR is SET, then *nvIndexType* shall not be TPM_NT_COUNTER or the TPM shall return TPM_RC_ATTRIBUTES.

If *platformAuth/platformPolicy* is used for authorization, then TPMA_NV_PLATFORMCREATE shall be SET in *publicInfo*. If *ownerAuth/ownerPolicy* is used for authorization, TPMA_NV_PLATFORMCREATE shall be CLEAR in *publicInfo.* If TPMA_NV_PLATFORMCREATE is not set correctly for the authorization, the TPM shall return TPM_RC_ATTRIBUTES.

If TPMA_NV_POLICY_DELETE is SET, then the authorization shall be with Platform Authorization or the TPM shall return TPM_RC_ATTRIBUTES.

If *nvIndexType* is TPM_NT_PIN_FAIL, then TPMA_NV_NO_DA shall be SET. Otherwise, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE 4 The intent of a PIN Fail index is that its DA protection is on a per-index basis, not based on the global DA protection. This avoids conflict over which type of dictionary attack protection is in use.

If *nvIndexType* is TPM_NT_PIN_FAIL or TPM_NT_PIN_PASS, then at least one of TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, or TPMA_NV_POLICYWRITE shall be SET or the TPM shall return TPM_RC_ATTRIBUTES. TPMA_NV_AUTHWRITE shall be CLEAR. Otherwise, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE 5 If TPMA_NV_AUTHWRITE was SET for a PIN Pass index, a user knowing the authorization value could decrease pinCount or increase pinLimit, defeating the purpose of a PIN Pass index. The requirement is also enforced for a PIN Fail index for consistency.

If the implementation does not support TPM2_NV_Increment(), the TPM shall return TPM_RC_ATTRIBUTES if *nvIndexType* is TPM_NT_COUNTER.

If the implementation does not support TPM2_NV_SetBits(), the TPM shall return TPM_RC_ATTRIBUTES if *nvIndexType* is TPM_NT_BITS.

If the implementation does not support TPM2_NV_Extend(), the TPM shall return TPM_RC_ATTRIBUTES if *nvIndexType* is TPM_NT_EXTEND.

If the implementation does not support TPM2_NV_UndefineSpaceSpecial(), the TPM shall return TPM_RC_ATTRIBUTES if TPMA_NV_POLICY_DELETE is SET.

After the successful completion of this command, the NV Index exists but TPMA_NV_WRITTEN will be CLEAR. Any access of the NV data will return TPM_RC_NV_UNINITIALIZED.

In some implementations, an NV Index with the TPM_NT_COUNTER attribute may require special TPM resources that provide higher endurance than regular NV. For those implementations, if this command fails because of lack of resources, the TPM will return TPM_RC_NV_SPACE.

The value of *auth* is saved in the created structure. The size of *auth* is limited to be no larger than the size of the digest produced by the NV Index's *nameAlg* (TPM_RC_SIZE).

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.3.2 Command and Response

Table 212 — TPM2_NV_DefineSpace Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_DefineSpace {NV}
TPMI_RH_PROVISION	@authHandle	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPM2B_AUTH	auth	the authorization value
TPM2B_NV_PUBLIC	publicInfo	the public parameters of the NV area

Table 213 — TPM2_NV_DefineSpace Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

31.3.3 Detailed Actions

[[NV_DefineSpace]]

31.4 TPM2_NV_UndefineSpace

31.4.1 General Description

This command removes an Index from the TPM.

If *nvIndex* is not defined, the TPM shall return TPM_RC_HANDLE.

If *nvIndex* references an Index that has its TPMA_NV_PLATFORMCREATE attribute SET, the TPM shall return TPM_RC_NV_AUTHORIZATION unless Platform Authorization is provided.

If *nvIndex* references an Index that has its TPMA_NV_POLICY_DELETE attribute SET, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE An Index with TPMA_NV_PLATFORMCREATE CLEAR may be deleted with Platform Authorization as long as shEnable is SET. If shEnable is CLEAR, indexes created using Owner Authorization are not accessible even for deletion by the platform.

 Copyright © TCG 2006-2020

31.4.2 Command and Response

Table 214 — TPM2_NV_UndefineSpace Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_UndefineSpace {NV}
TPMI_RH_PROVISION	@authHandle	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index to remove from NV space Auth Index: None

Table 215 — TPM2_NV_UndefineSpace Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.4.3 Detailed Actions

[[NV_UndefineSpace]]

31.5 TPM2_NV_UndefineSpaceSpecial

31.5.1 General Description

This command allows removal of a platform-created NV Index that has TPMA_NV_POLICY_DELETE SET.

This command requires that the policy of the NV Index be satisfied before the NV Index may be deleted. Because administrative role is required, the policy must contain a command that sets the policy command code to TPM_CC_NV_UndefineSpaceSpecial. This indicates that the policy that is being used is a policy that is for this command, and not a policy that would approve another use. That is, authority to use an entity does not grant authority to undefine the entity.

Since the index is deleted, the Empty Buffer is used as the authValue when generating the response HMAC.

If *nvIndex* is not defined, the TPM shall return TPM_RC_HANDLE.

If *nvIndex* references an Index that has its TPMA_NV_PLATFORMCREATE or TPMA_NV_POLICY_DELETE attribute CLEAR, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE An Index with TPMA_NV_PLATFORMCREATE CLEAR may be deleted with TPM2_UndefineSpace()as long as shEnable is SET. If shEnable is CLEAR, indexes created using Owner Authorization are not accessible even for deletion by the platform.

 Copyright © TCG 2006-2020

31.5.2 Command and Response

Table 216 — TPM2_NV_UndefineSpaceSpecial Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_UndefineSpaceSpecial {NV}
TPMI_RH_NV_INDEX	@nvIndex	Index to be deleted Auth Index: 1 Auth Role: ADMIN
TPMI_RH_PLATFORM	@platform	TPM_RH_PLATFORM + {PP} Auth Index: 2 Auth Role: USER

Table 217 — TPM2_NV_UndefineSpaceSpecial Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

31.5.3 Detailed Actions

[[NV_UndefineSpaceSpecial]]

31.6 TPM2_NV_ReadPublic

31.6.1 General Description

This command is used to read the public area and Name of an NV Index. The public area of an Index is not privacy-sensitive and no authorization is required to read this data.

 Copyright © TCG 2006-2020

31.6.2 Command and Response

Table 218 — TPM2_NV_ReadPublic Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_ReadPublic
TPMI_RH_NV_INDEX	nvIndex	the NV Index Auth Index: None

Table 219 — TPM2_NV_ReadPublic Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_NV_PUBLIC	nvPublic	the public area of the NV Index
TPM2B_NAME	nvName	the Name of the *nvIndex*

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.6.3 Detailed Actions

[[NV_ReadPublic]]

 Copyright © TCG 2006-2020

31.7 TPM2_NV_Write

31.7.1 General Description

This command writes a value to an area in NV memory that was previously defined by TPM2_NV_DefineSpace().

Proper authorizations are required for this command as determined by TPMA_NV_PPWRITE; TPMA_NV_OWNERWRITE; TPMA_NV_AUTHWRITE; and, if TPMA_NV_POLICY_WRITE is SET, the *authPolicy* of the NV Index.

If the TPMA_NV_WRITELOCKED attribute of the NV Index is SET, then the TPM shall return TPM_RC_NV_LOCKED.

NOTE 1 If authorization sessions are present, they are checked before checks to see if writes to the NV Index are locked.

If *nvIndexType* is TPM_NT_COUNTER, TPM_NT_BITS or TPM_NT_EXTEND, then the TPM shall return TPM_RC_ATTRIBUTES.

If *offset* and the *size* field of *data* add to a value that is greater than the *dataSize* field of the NV Index referenced by *nvIndex*, the TPM shall return an error (TPM_RC_NV_RANGE). The implementation may return an error (TPM_RC_VALUE) if it performs an additional check and determines that *offset* is greater than the *dataSize* field of the NV Index.

If the TPMA_NV_WRITEALL attribute of the NV Index is SET, then the TPM shall return TPM_RC_NV_RANGE if the size of the *data* parameter of the command is not the same as the *data* field of the NV Index.

If all checks succeed, the TPM will merge the *data.size* octets of *data.buffer* value into the *nvIndex→data* starting at *nvIndex→data[offset].* If the NV memory is implemented with a technology that has endurance limitations, the TPM shall check that the merged data is different from the current contents of the NV Index and only perform a write to NV memory if they differ.

After successful completion of this command, TPMA_NV_WRITTEN for the NV Index will be SET.

NOTE 2 Once SET, TPMA_NV_WRITTEN remains SET until the NV Index is undefined or the NV Index is cleared.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.7.2 Command and Response

Table 220 — TPM2_NV_Write Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_Write {NV}
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index of the area to write Auth Index: None
TPM2B_MAX_NV_BUFFER	data	the data to write
UINT16	offset	the octet offset into the NV Area

Table 221 — TPM2_NV_Write Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

31.7.3 Detailed Actions

[[NV_Write]]

 Copyright © TCG 2006-2020

31.8 TPM2_NV_Increment

31.8.1 General Description

This command is used to increment the value in an NV Index that has the TPM_NT_COUNTER attribute. The data value of the NV Index is incremented by one.

NOTE 1 The NV Index counter is an unsigned value.

If *nvIndexType* is not TPM_NT_COUNTER in the indicated NV Index, the TPM shall return TPM_RC_ATTRIBUTES.

If TPMA_NV_WRITELOCKED is SET, the TPM shall return TPM_RC_NV_LOCKED.

If TPMA_NV_WRITTEN is CLEAR, it will be SET.

If TPMA_NV_ORDERLY is SET, and the difference between the volatile and non-volatile versions of this field is greater than MAX_ORDERLY_COUNT, then the non-volatile version of the counter is updated.

NOTE 2 If a TPM implements TPMA_NV_ORDERLY and an Index is defined with TPMA_NV_ORDERLY and TPM_NT_COUNTER both SET, then in the Event of a non-orderly shutdown, the non-volatile value for the counter Index will be advanced by MAX_ORDERLY_COUNT at the next TPM2_Startup().

NOTE 3 An allowed implementation would keep a counter value in NV and a resettable counter in RAM. The reported value of the NV Index would be the sum of the two values. When the RAM count increments past the maximum allowed value (MAX_ORDERLY_COUNT), the non-volatile version of the count is updated with the sum of the values and the RAM count is reset to zero.

 Copyright © TCG 2006-2020

31.8.2 Command and Response

Table 222 — TPM2_NV_Increment Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_Increment {NV}
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index to increment Auth Index: None

Table 223 — TPM2_NV_Increment Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

31.8.3 Detailed Actions

[[NV_Increment]]

31.9 TPM2_NV_Extend

31.9.1 General Description

This command extends a value to an area in NV memory that was previously defined by TPM2_NV_DefineSpace.

If *nvIndexType* is not TPM_NT_EXTEND, then the TPM shall return TPM_RC_ATTRIBUTES.

Proper write authorizations are required for this command as determined by TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, TPMA_NV_AUTHWRITE, and the *authPolicy* of the NV Index.

After successful completion of this command, TPMA_NV_WRITTEN for the NV Index will be SET.

NOTE 1 Once SET, TPMA_NV_WRITTEN remains SET until the NV Index is undefined, unless the TPMA_NV_CLEAR_STCLEAR attribute is SET and a TPM Reset or TPM Restart occurs.

If the TPMA_NV_WRITELOCKED attribute of the NV Index is SET, then the TPM shall return TPM_RC_NV_LOCKED.

NOTE 2 If authorization sessions are present, they are checked before checks to see if writes to the NV Index are locked.

The *data.buffer* parameter may be larger than the defined size of the NV Index.

The Index will be updated by:

$$nvIndex{\rightarrow}data_{new} \coloneqq \mathbf{H}_{nameAkg}(nvIndex{\rightarrow}data_{old} \mid\mid data.buffer) \tag{41}$$

where

$nvIndex{\rightarrow}data_{new}$	the value of the data field in the NV Index after the command returns
$\mathbf{H}_{nameAkg}()$	the hash algorithm indicated in *nvIndex→nameAlg*
$nvIndex{\rightarrow}data_{old}$	the value of the data field in the NV Index before the command is called
$data.buffer$	the data buffer of the command parameter

NOTE 3 If TPMA_NV_WRITTEN is CLEAR, then $nvIndex{\rightarrow}data_{old}$ is a Zero Digest.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.9.2 Command and Response

Table 224 — TPM2_NV_Extend Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_Extend {NV}
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index to extend Auth Index: None
TPM2B_MAX_NV_BUFFER	data	the data to extend

Table 225 — TPM2_NV_Extend Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

31.9.3 Detailed Actions

[[NV_Extend]]

 Copyright © TCG 2006-2020

31.10 TPM2_NV_SetBits

31.10.1 General Description

This command is used to SET bits in an NV Index that was created as a bit field. Any number of bits from 0 to 64 may be SET. The contents of *bits* are ORed with the current contents of the NV Index.

If TPMA_NV_WRITTEN is not SET, then, for the purposes of this command, the NV Index is considered to contain all zero bits and *data* is ORed with that value.

If TPM_NT_BITS is not SET, then the TPM shall return TPM_RC_ATTRIBUTES.

After successful completion of this command, TPMA_NV_WRITTEN for the NV Index will be SET.

NOTE TPMA_NV_WRITTEN will be SET even if no bits were SET.

 Copyright © TCG 2006-2020

31.10.2 Command and Response

Table 226 — TPM2_NV_SetBits Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_SetBits {NV}
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	NV Index of the area in which the bit is to be set Auth Index: None
UINT64	bits	the data to OR with the current contents

Table 227 — TPM2_NV_SetBits Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.10.3 Detailed Actions

[[NV_SetBits]]

31.11 TPM2_NV_WriteLock

31.11.1 General Description

If the TPMA_NV_WRITEDEFINE or TPMA_NV_WRITE_STCLEAR attributes of an NV location are SET, then this command may be used to inhibit further writes of the NV Index.

Proper write authorization is required for this command as determined by TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, TPMA_NV_AUTHWRITE, and the *authPolicy* of the NV Index.

It is not an error if TPMA_NV_WRITELOCKED for the NV Index is already SET.

If neither TPMA_NV_WRITEDEFINE nor TPMA_NV_WRITE_STCLEAR of the NV Index is SET, then the TPM shall return TPM_RC_ATTRIBUTES.

If the command is properly authorized and TPMA_NV_WRITE_STCLEAR or TPMA_NV_WRITEDEFINE is SET, then the TPM shall SET TPMA_NV_WRITELOCKED for the NV Index. TPMA_NV_WRITELOCKED will be clear on the next TPM2_Startup(TPM_SU_CLEAR) if either TPMA_NV_WRITEDEFINE is CLEAR or TPMA_NV_WRITTEN is CLEAR.

 Copyright © TCG 2006-2020

31.11.2 Command and Response

Table 228 — TPM2_NV_WriteLock Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_WriteLock {NV}
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index of the area to lock Auth Index: None

Table 229 — TPM2_NV_WriteLock Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

31.11.3 Detailed Actions

[[NV_WriteLock]]

 Copyright © TCG 2006-2020

31.12 TPM2_NV_GlobalWriteLock

31.12.1 General Description

The command will SET TPMA_NV_WRITELOCKED for all indexes that have their TPMA_NV_GLOBALLOCK attribute SET.

If an Index has both TPMA_NV_GLOBALLOCK and TPMA_NV_WRITEDEFINE SET, then this command will permanently lock the NV Index for writing unless TPMA_NV_WRITTEN is CLEAR.

NOTE If an Index is defined with TPMA_NV_GLOBALLOCK SET, then the global lock does not apply until the next time this command is executed.

This command requires either platformAuth/platformPolicy or ownerAuth/ownerPolicy.

 Copyright © TCG 2006-2020

31.12.2 Command and Response

Table 230 — TPM2_NV_GlobalWriteLock Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_GlobalWriteLock {NV}
TPMI_RH_PROVISION	@authHandle	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER

Table 231 — TPM2_NV_GlobalWriteLock Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.12.3 Detailed Actions

[[NV_GlobalWriteLock]]

31.13 TPM2_NV_Read

31.13.1 General Description

This command reads a value from an area in NV memory previously defined by TPM2_NV_DefineSpace().

Proper authorizations are required for this command as determined by TPMA_NV_PPREAD, TPMA_NV_OWNERREAD, TPMA_NV_AUTHREAD, and the *authPolicy* of the NV Index.

If TPMA_NV_READLOCKED of the NV Index is SET, then the TPM shall return TPM_RC_NV_LOCKED.

If *offset* and the *size* field of *data* add to a value that is greater than the *dataSize* field of the NV Index referenced by *nvIndex*, the TPM shall return an error (TPM_RC_NV_RANGE). The implementation may return an error (TPM_RC_VALUE) if it performs an additional check and determines that *offset* is greater than the *dataSize* field of the NV Index.

For an NV Index with the TPM_NT_COUNTER or TPM_NT_BITS attribute SET, the TPM may ignore the *offset* parameter and use an offset of 0. Therefore, it is recommended that the caller set the *offset* parameter to 0 for interoperability.

NOTE 1 If authorization sessions are present, they are checked before the read-lock status of the NV Index is checked.

If the NV Index has been defined but the TPMA_NV_WRITTEN attribute is CLEAR, then this command shall return TPM_RC_NV_UNINITIALIZED even if *size* is zero.

The *data* parameter in the response may be encrypted using parameter encryption.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.13.2 Command and Response

Table 232 — TPM2_NV_Read Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_Read
TPMI_RH_NV_AUTH	@authHandle	the handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index to be read Auth Index: None
UINT16	size	number of octets to read
UINT16	offset	octet offset into the NV area This value shall be less than or equal to the size of the *nvIndex* data.

Table 233 — TPM2_NV_Read Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_MAX_NV_BUFFER	data	the data read

 Copyright © TCG 2006-2020

31.13.3 Detailed Actions

[[NV_Read]]

31.14 TPM2_NV_ReadLock

31.14.1 General Description

If TPMA_NV_READ_STCLEAR is SET in an Index, then this command may be used to prevent further reads of the NV Index until the next TPM2_Startup (TPM_SU_CLEAR).

Proper authorizations are required for this command as determined by TPMA_NV_PPREAD, TPMA_NV_OWNERREAD, TPMA_NV_AUTHREAD, and the *authPolicy* of the NV Index.

NOTE Only an entity that may read an Index is allowed to lock the NV Index for read.

If the command is properly authorized and TPMA_NV_READ_STCLEAR of the NV Index is SET, then the TPM shall SET TPMA_NV_READLOCKED for the NV Index. If TPMA_NV_READ_STCLEAR of the NV Index is CLEAR, then the TPM shall return TPM_RC_ATTRIBUTES. TPMA_NV_READLOCKED will be CLEAR by the next TPM2_Startup(TPM_SU_CLEAR).

It is not an error to use this command for an Index that is already locked for reading.

An Index that had not been written may be locked for reading.

 Copyright © TCG 2006-2020

31.14.2 Command and Response

Table 234 — TPM2_NV_ReadLock Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_ReadLock {NV}
TPMI_RH_NV_AUTH	@authHandle	the handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index to be locked Auth Index: None

Table 235 — TPM2_NV_ReadLock Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.14.3 Detailed Actions

[[NV_ReadLock]]

31.15 TPM2_NV_ChangeAuth

31.15.1 General Description

This command allows the authorization secret for an NV Index to be changed.

If successful, the authorization secret (*authValue*) of the NV Index associated with *nvIndex* is changed.

This command requires that a policy session be used for authorization of *nvIndex* so that the ADMIN role may be asserted and that *commandCode* in the policy session context shall be TPM_CC_NV_ChangeAuth. That is, the policy must contain a specific authorization for changing the authorization value of the referenced entity.

NOTE The reason for this restriction is to ensure that the administrative actions on *nvIndex* require explicit approval while other commands may use policy that is not command-dependent.

The size of the *newAuth* value may be no larger than the size of the digest produced by the *nameAlg* of the NV Index.

Since the NV Index authorization is changed before the response HMAC is calculated, the newAuth value is used when generating the response HMAC key if required. See TPM 2.0 Part 4 ComputeResponseHMAC().

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.15.2 Command and Response

Table 236 — TPM2_NV_ChangeAuth Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_ChangeAuth {NV}
TPMI_RH_NV_INDEX	@nvIndex	handle of the entity Auth Index: 1 Auth Role: ADMIN
TPM2B_AUTH	newAuth	new authorization value

Table 237 — TPM2_NV_ChangeAuth Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

31.15.3 Detailed Actions

[[NV_ChangeAuth]]

 Copyright © TCG 2006-2020

31.16 TPM2_NV_Certify

31.16.1 General Description

The purpose of this command is to certify the contents of an NV Index or portion of an NV Index.

If the *sign* attribute is not SET in the key referenced by *signHandle* then the TPM shall return TPM_RC_KEY.

If the NV Index has been defined but the TPMA_NV_WRITTEN attribute is CLEAR, then this command shall return TPM_RC_NV_UNINITIALIZED even if *size* is zero.

If proper authorization for reading the NV Index is provided, the portion of the NV Index selected by *size* and *offset* are included in an attestation block and signed using the key indicated by *signHandle*. The attestation includes *size* and *offset* so that the range of the data can be determined. It also includes the NV index Name.

For an NV Index with the TPM_NT_COUNTER or TPM_NT_BITS attribute SET, the TPM may ignore the *offset* parameter and use an offset of 0. Therefore, it is recommended that the caller set the *offset* parameter to 0 for interoperability.

If *offset* and *size* add to a value that is greater than the *dataSize* field of the NV Index referenced by *nvIndex*, the TPM shall return an error (TPM_RC_NV_RANGE). The implementation may return an error (TPM_RC_VALUE) if it performs an additional check and determines that *offset* is greater than the *dataSize* field of the NV Index, or if *size* is greater than MAX_NV_BUFFER_SIZE.

NOTE 1 See 18.1 for description of how the signing scheme is selected.

NOTE 2 If *signHandle* is TPM_RH_NULL, the TPMS_ATTEST structure is returned and *signature* is a NULL Signature.

If *size* and *offset* are both zero (0), then *certifyInfo* in the response will contain a TPMS_NV_DIGEST_CERTIFY_INFO, otherwise, it will contain a TPMS_NV_CERTIFY_INFO. The digest in the TPMS_NV_DIGEST_CERTIFY_INFO is created using the digest of the selected signing scheme.

NOTE 3 TPMS_NV_DIGEST_CERTIFY_INFO was added in revision 01.53. It permits TPM2_NV_Certify() to certify NV Index contents that are larger than MAX_NV_BUFFER_SIZE.

 Copyright © TCG 2006-2020

31.16.2 Command and Response

Table 238 — TPM2_NV_Certify Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_Certify
TPMI_DH_OBJECT+	@signHandle	handle of the key used to sign the attestation structure Auth Index: 1 Auth Role: USER
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value for the NV Index Auth Index: 2 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	Index for the area to be certified Auth Index: None
TPM2B_DATA	qualifyingData	user-provided qualifying data
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL
UINT16	size	number of octets to certify
UINT16	offset	octet offset into the NV area This value shall be less than or equal to the size of the *nvIndex* data.

Table 239 — TPM2_NV_Certify Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPM2B_ATTEST	certifyInfo	the structure that was signed
TPMT_SIGNATURE	signature	the asymmetric signature over *certifyInfo* using the key referenced by *signHandle*

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

31.16.3 Detailed Actions

[[NV_Certify]]

32 Attached Components

32.1 Introduction

This section contains commands that allow interaction with an Attached Component (AC).

NOTE The Attached Component feature was added in revision 01.40.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

32.2 TPM2_AC_GetCapability

32.2.1 General Description

The purpose of this command is to obtain information about an Attached Component referenced by an AC handle.

The returned list contains 0 or more values starting at the first tagged value that is equal to or greater than *capability*.

The list returned in *capabilitiesData* contains tagged values that indicate the type of the value.

The TPM will return the lesser of a) the available values, b) the number requested in *count,* or c) the number that will fit within the available response buffer. If additional values with higher *capability* numbers are available, *moreData* will be YES.

NOTE TPM2_AC_GetCapability() was added in revision 01.40.

 Copyright © TCG 2006-2020

32.2.2 Command and Response

Table 240 — TPM2_AC_GetCapability Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_AC_GetCapability
TPMI_RH_AC	ac	handle indicating the Attached Component Auth Index: None
TPM_AT	capability	starting info type
UINT32	count	maximum number of values to return

Table 241 — TPM2_AC_GetCapability Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPMI_YES_NO	moreData	flag to indicate whether there are more values
TPML_AC_CAPABILITIES	capabilitiesData	list of capabilities

 Copyright © TCG 2006-2020

32.2.3 Detailed Actions

[[AC_GetCapability]]

32.3 TPM2_AC_Send

32.3.1 General Description

The purpose of this command is to send (copy) a loaded object from the TPM to an Attached Component.

The Object referenced by *sendObject* is required to have *fixedTpm, fixedParent,* and *encryptedDuplication* attributes CLEAR (TPM_RC_ATTRIBUTES). Authorization for *sendObject* is required to be a policy session. The *policySession→commandCode* of the policy session context is required to be TPM_CC_AC_Send (TPM_RC_POLICY_FAIL) to demonstrate that the policy is specific for this command.

Authorization to send to the *ac* is provided by the session associated with *authHandle*.

If an NV Alias is not defined for *ac,* then *authHandle* is required to be either TPM_RH_OWNER or TPM_RH_PLATFORM (TPM_RC_HANDLE).

If an NV Alias is defined for *ac*, then the authorization for *authHandle* is required to be compatible with the write authorization attributes (TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE TPMA_NV_AUTHWRITE, and TPMA_NV_POLICYWRITE) in the NV Alias (TPM_RC_NV_AUTHORIZATION).

NOTE 1 If authorization for *authHandle* is the handle of an NV Index, then it is required to be the NV Alias value for *ac* (TPM_RC_NV_AUTHORIZATION).

If authorization succeeds, the TPM will attempt to send *acDataIn* and relevant portions of *sendObject* to the AC referenced by *ac*.

The TPM will return TPM_RC_SUCCESS if it succeeds in performing all the required authorizations and validations. If problems occur in the process of sending the object from the TPM to the AC, the response code will be TPM_RC_SUCCESS with the AC-dependent error reported in acDataOut.

NOTE 2 TPM2_AC_Send() was added in revision 01.40.

 Copyright © TCG 2006-2020

32.3.2 Command and Response

Table 242 — TPM2_AC_Send Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	Tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_AC_Send
TPMI_DH_OBJECT	@sendObject	handle of the object being sent to ac Auth Index: 1 Auth Role: DUP
TPMI_RH_NV_AUTH	@authHandle	the handle indicating the source of the authorization value Auth Index: 2 Auth Role: USER
TPMI_RH_AC	ac	handle indicating the Attached Component to which the object will be sent Auth Index: None
TPM2B_MAX_BUFFER	acDataIn	Optional non sensitive information related to the object

Table 243 — TPM2_AC_Send Response

Type	Name	Description
TPM_ST	Tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMS_AC_OUTPUT	acDataOut	May include AC specific data or information about an error.

 Copyright © TCG 2006-2020

32.3.3 Detailed Actions

[[AC_Send]]

32.4 TPM2_Policy_AC_SendSelect

32.4.1 General Description

This command allows qualification of the sending (copying) of an Object to an Attached Component (AC). Qualification includes selection of the receiving AC and the method of authentication for the AC, and, in certain circumstances, the Object to be sent may be specified.

If this command is not used in conjunction with TPM2_PolicyAuthorize(), then only the *authHandleName* and *acName* are selected and *includeObject* should be CLEAR.

NOTE 1 In the absence of TPM2_PolicyAuthorize(), a policy session cannot create a *policyDigest* that simultaneously equals the *authPolicy* in an Object and names that Object. This is because the *authPolicy* recorded in an Object is unable to include the Name of the Object as the Name of an Object depends on the Object's *authPolicy*.

NOTE 2 An object's *authPolicy* can incorporate the use of TPM2_PolicyAuthorize(). If the authorizing entity for the TPM2_PolicyAuthorize() command specifies only the *ac* and the *authHandle*, then the resultant *policyDigest* may be applied to the sending of any number of Objects. If the authorizing entity for the TPM2_PolicyAuthorize() specifies also the Name of the Object to be sent, then the resultant *policyDigest* applies only to that specific Object.

If either policySession→cpHash or policySession→nameHash has been previously set, the TPM shall return TPM_RC_CPHASH. Otherwise, policySession→nameHash will be set to:

$$nameHash \coloneqq \mathbf{H}_{policyAlg}(objectName \mathbin{||} authHandleName \mathbin{||} acName) \quad (42)$$

NOTE 3 A policy cannot specify both *cpHash* and *nameHash* because *policySession→nameHash* and *policySession→cpHash* may share the same memory space.

If the command succeeds, *policySession→policyDigest* will be updated according to the setting of the input parameter *includeObject*. If *includeObject* is SET, *policySession→policyDigest* is updated by:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_Policy_AC_SendSelect} \mathbin{||} objectName \mathbin{||} authHandleName \mathbin{||} acName \mathbin{||} includeObject) \quad (43)$$

but if includeObject is CLEAR, *policySession→policyDigest* is updated by:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM_CC_Policy_AC_SendSelect} \mathbin{||} authHandleName \mathbin{||} acName \mathbin{||} includeObject) \quad (44)$$

NOTE 4 *policySession→nameHash* receives the digest of all Names so that the check performed in TPM2_AC_Send() may be the same regardless of which Names are included in *policySession→policyDigest*. This means that, when TPM2_Policy_AC_SendSelect() is executed, it is only valid for a specific triple of *objectName*, *authHandleName*, and *acName*.

If the command succeeds, *policySession→commandCode* is set to TPM_CC_AC_Send.

NOTE 5 The normal use of TPM2_Policy_AC_SendSelect() is before a TPM2_PolicyAuthorize(). An authorized entity would approve a *policyDigest* that allows sending to a specific Attached Component. The authorizing entity may want to limit the authorization so that the approval allows only a specific Object to be sent to the Attached Component. In that case, the authorizing entity would approve the *policyDigest* of *equation* (44).

NOTE 6 TPM2_Policy_AC_SendSelect() was added in revision 01.40.

 Copyright © TCG 2006-2020

32.4.2 Command and Response

Table 244 — TPM2_Policy_AC_SendSelect Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	Tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Policy_AC_SendSelect
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_NAME	objectName	the Name of the Object to be sent
TPM2B_NAME	authHandleName	the Name associated with *authHandle* used in the TPM2_AC_Send() command
TPM2B_NAME	acName	the Name of the Attached Component to which the Object will be sent
TPMI_YES_NO	includeObject	if SET, *objectName* will be included in the value in *policySession→policyDigest*

Table 245 — TPM2_Policy_AC_SendSelect Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

32.4.3 Detailed Actions

[[Policy_AC_SendSelect]]

33 Authenticated Countdown Timer

33.1 Introduction

This section contains commands that allow interaction with an Authenticated Countdown Timer (ACT).

NOTE The Authenticated Countdown Timer was added in revision 01.56.

33.2 TPM2_ACT_SetTimeout

33.2.1 General Description

This command is used to set the time remaining before an Authenticated Countdown Timer (ACT) expires.

This command sets TPMS_ACT_DATA.*timeout* (ACT Timeout) to *startTimeout*. The *startTimeout* value is an integer number of seconds and may be zero. The *startTimeout* parameter may be greater, equal, or less than the current value of ACT Timeout.

When ACT Timeout is non-zero, it will count down, once per second until it reaches zero, at which time the *signaled* attribute of the TPMA_ACT associated with *actHandle* is SET.

When ACT Timeout is zero and the *signaled* attribute is SET, writing a *startTimeout* of $FF\ FF\ FF\ FF_{16}$ will clear *signaled* and stop the counting.

There are four states for ACT Timeout and *startTimeout*. The *signaled* attribute will be set as follows:

1) If ACT Timeout is zero and *startTimeout* is non-zero, then *signaled* will be CLEAR.

2) If ACT Timeout is non-zero and *startTimeout* is non-zero, then *signaled* will be CLEAR.

3) If ACT Timeout is zero and *startTimeout* is zero, then *signaled* will be unchanged.

4) If ACT Timeout is non-zero and *startTimeout* is zero, then *signaled* will be SET.

NOTE 1 The ACT signals on a transition from non-zero to zero. The transition can occur either due to TPM2_ACT_SetTimeout() or a decrement. The effect of *signaled* is platform dependent.

NOTE 2 It may take up to one second until ACT Timeout will be set and *signaled* will be CLEAR or SET by TPM2_ACT_SetTimeout() or TPM2_Startup(STATE). This allows the counting and signaling to take place synchronously with the hardware clock tick.

NOTE 3 TPM2_ACT_SetTimeout() was added in revision 01.56.

Level 00 Revision 01.59 **Copyright © TCG** 2006-2020 November 8, 2019

33.2.2 Command and Response

Table 246 — TPM2_ACT_SetTimeout Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ACT_SetTimeout
TPMI_RH_ACT	@actHandle	Handle of the selected ACT Auth Index: 1 Auth Role: USER
UINT32	startTimeout	the start timeout value for the ACT in seconds

Table 247 — TPM2_ACT_SetTimeout Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

 Copyright © TCG 2006-2020

33.2.3 Detailed Actions

[[ACT_SetTimeout]]

 Copyright © TCG 2006-2020

34 Vendor Specific

34.1 Introduction

This section contains commands that are vendor specific but made public in order to prevent proliferation.

This specification does define TPM2_Vendor_TCG_Test() in order to have at least one command that can be used to ensure the proper operation of the command dispatch code when processing a vendor-specific command.

34.2 TPM2_Vendor_TCG_Test

34.2.1 General Description

This is a placeholder to allow testing of the dispatch code.

 Copyright © TCG 2006-2020

34.2.2 Command and Response

Table 248 — TPM2_Vendor_TCG_Test Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Vendor_TCG_Test
TPM2B_DATA	inputData	dummy data

Table 249 — TPM2_Vendor_TCG_Test Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	TPM_RC_SUCCESS
TPM2B_DATA	outputData	dummy data

 Copyright © TCG 2006-2020

34.2.3 Detailed Actions

[[Vendor_TCG_Test]]

Trusted Platform Module Library
Part 4: Supporting Routines

Family "2.0"

Level 00 Revision 01.59

November 8, 2019

Published

Contact: admin@trustedcomputinggroup.org

TCG Published

Copyright © TCG 2006-2020

Licenses and Notices

Copyright Licenses:

- Trusted Computing Group (TCG) grants to the user of the source code in this specification (the “Source Code”) a worldwide, irrevocable, nonexclusive, royalty free, copyright license to reproduce, create derivative works, distribute, display and perform the Source Code and derivative works thereof, and to grant others the rights granted herein.
- The TCG grants to the user of the other parts of the specification (other than the Source Code) the rights to reproduce, distribute, display, and perform the specification solely for the purpose of developing products based on such documents.

Source Code Distribution Conditions:

- Redistributions of Source Code must retain the above copyright licenses, this list of conditions and the following disclaimers.
- Redistributions in binary form must reproduce the above copyright licenses, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.

Disclaimers:

- THE COPYRIGHT LICENSES SET FORTH ABOVE DO NOT REPRESENT ANY FORM OF LICENSE OR WAIVER, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, WITH RESPECT TO PATENT RIGHTS HELD BY TCG MEMBERS (OR OTHER THIRD PARTIES) THAT MAY BE NECESSARY TO IMPLEMENT THIS SPECIFICATION OR OTHERWISE. Contact TCG Administration (admin@trustedcomputinggroup.org) for information on specification licensing rights available through TCG membership agreements.
- THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO EXPRESS OR IMPLIED WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, ACCURACY, COMPLETENESS, OR NONINFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.
- Without limitation, TCG and its members and licensors disclaim all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification and to the implementation of this specification, and TCG disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this specification or any information herein.

Any marks and brands contained herein are the property of their respective owners.

CONTENTS

1 Scope 1

2 Terms and definitions 1

3 Symbols and abbreviated terms 1

4 Automation 1

4.1 Configuration Parser 1
4.2 Structure Parser 2

4.2.1 Introduction 2
4.2.2 Unmarshaling Code Prototype 2

4.2.2.1 Simple Types and Structures 2
4.2.2.2 Union Types 3
4.2.2.3 Null Types 3
4.2.2.4 Arrays 3

4.2.3 Marshaling Code Function Prototypes 3

4.2.3.1 Simple Types and Structures 3
4.2.3.2 Union Types 4
4.2.3.3 Arrays 4
4.2.3.4 The generated code for an array uses a `count`-limited loop within which it calls the marshaling code for `TYPE`.Table-driven Marshaling 4

4.3 Part 3 Parsing 4
4.4 Function Prototypes 5
4.5 Portability 6

5 Header Files 7

5.1 Introduction 7
5.2 BaseTypes.h 7
5.3 Capabilities.h 8
5.4 CommandAttributeData.h 9
5.5 CommandAttributes.h 23
5.6 CommandDispatchData.h 24
5.7 Commands.h 93
5.8 CompilerDependencies.h 100
5.9 Global.h 102

5.9.1 Description 102
5.9.2 Includes 102
5.9.3 Loaded Object Structures 103

5.9.3.1 Description 103
5.9.3.2 OBJECT_ATTRIBUTES 103
5.9.3.3 OBJECT Structure 104
5.9.3.4 HASH_OBJECT Structure 104
5.9.3.5 ANY_OBJECT 105

5.9.4 AUTH_DUP Types 105
5.9.5 Active Session Context 105

5.9.5.1 Description 105
5.9.5.2 SESSION_ATTRIBUTES 105
5.9.5.3 SESSION Structure 106

5.9.6 PCR 107

5.9.6.1 PCR_SAVE Structure 107
5.9.6.2 PCR_POLICY 108
5.9.6.3 PCR_AUTHVALUE 108

 Copyright © TCG 2006-2020

5.9.7 STARTUP_TYPE 108
5.9.8 NV 108
5.9.8.1 NV_INDEX 108
5.9.8.2 NV_REF 108
5.9.8.3 NV_PIN 109
5.9.9 COMMIT_INDEX_MASK 109
5.9.10 RAM Global Values 109
5.9.10.1 Description 109
5.9.10.2 Crypto Self-Test Values 109
5.9.10.3 g_exclusiveAuditSession 110
5.9.10.4 TPM_SU_DA_USED 111
5.9.10.5 Startup Flags 111
5.9.10.6 g_daUsed 111
5.9.11 Global Macro Definitions 118
5.9.12 From CryptTest.c 120
5.9.12.1 From Object.c 122
5.10 GpMacros.h 125
5.10.1 Introduction 125
5.10.2 For Self-test 125
5.10.3 For Failures 125
5.10.4 Derived from Vendor-specific values 126
5.10.5 Compile-time Checks 126
5.11 InternalRoutines.h 131
5.12 LibSupport.h 133
5.13 MinMax.h 133
5.14 NV.h 134
5.14.1 Index Type Definitions 134
5.14.2 Attribute Macros 134
5.14.3 Orderly RAM Values 135
5.15 TPMB.h 137
5.16 Tpm.h 138
5.17 TpmBuildSwitches.h 139
5.18 TpmError.h 145
5.19 TpmTypes.h 146
5.20 VendorString.h 182
5.21 swap.h 183
5.22 ACT.h 185
6 Main 188
6.1 Introduction 188
6.2 ExecCommand.c 188
6.2.1 Introduction 188
6.2.2 Includes 188
6.2.3 ExecuteCommand() 188
6.3 CommandDispatcher.c 194
6.3.1 Introduction 194
6.3.1.1 Includes and Typedefs 194
6.3.1.2 Marshal/Unmarshal Functions 196
6.3.1.2.1 ParseHandleBuffer() 196
6.3.1.2.2 CommandDispatcher() 198

 Copyright © TCG 2006-2020

6.4 SessionProcess.c 202
6.4.1 Introduction 202
6.4.2 Includes and Data Definitions 202
6.4.3 Authorization Support Functions 202
6.4.3.1 IsDAExempted() 202
6.4.3.2 IncrementLockout() 203
6.4.3.3 IsSessionBindEntity() 204
6.4.3.4 IsPolicySessionRequired() 205
6.4.3.5 IsAuthValueAvailable() 206
6.4.3.6 IsAuthPolicyAvailable() 208
6.4.4 Session Parsing Functions 209
6.4.4.1 ClearCpRpHashes() 209
6.4.4.2 GetCpHashPointer() 210
6.4.4.3 GetRpHashPointer() 211
6.4.4.4 ComputeCpHash() 211
6.4.4.5 GetCpHash() 212
6.4.4.6 CompareTemplateHash() 212
6.4.4.7 CompareNameHash() 213
6.4.4.8 CheckPWAuthSession() 214
6.4.4.9 ComputeCommandHMAC() 214
6.4.4.10 CheckSessionHMAC() 216
6.4.4.11 CheckPolicyAuthSession() 216
6.4.4.12 RetrieveSessionData() 219
6.4.4.13 CheckLockedOut() 222
6.4.4.14 CheckAuthSession() 223
6.4.4.15 CheckCommandAudit() 225
6.4.4.16 ParseSessionBuffer() 226
6.4.4.17 CheckAuthNoSession() 228
6.4.5 Response Session Processing 229
6.4.5.1 Introduction 229
6.4.5.2 ComputeRpHash() 229
6.4.5.3 InitAuditSession() 229
6.4.5.4 UpdateAuditDigest 230
6.4.5.5 Audit() 230
6.4.5.6 CommandAudit() 230
6.4.5.7 UpdateAuditSessionStatus() 231
6.4.5.8 ComputeResponseHMAC() 232
6.4.5.9 UpdateInternalSession() 233
6.4.5.10 BuildSingleResponseAuth() 234
6.4.5.11 UpdateAllNonceTPM() 234
6.4.5.12 BuildResponseSession() 235
6.4.5.13 SessionRemoveAssociationToHandle() 236
7 Command Support Functions 237
7.1 Introduction 237
7.2 Attestation Command Support (Attest_spt.c) 237
7.2.1 Includes 237
7.2.2 Functions 237
7.2.2.1 FillInAttestInfo() 237
7.2.2.2 SignAttestInfo() 238
7.2.2.3 IsSigningObject() 239
7.3 Context Management Command Support (Context_spt.c) 240
7.3.1 Includes 240

7.3.2 Functions 240
7.3.2.1 ComputeContextProtectionKey() 240
7.3.2.2 ComputeContextIntegrity() 241
7.3.2.3 SequenceDataExport() 242
7.3.2.4 SequenceDataImport() 242
7.4 Policy Command Support (Policy_spt.c) 243
7.4.1 Includes 243
7.4.2 Functions 243
7.4.2.1 PolicyParameterChecks() 243
7.4.2.2 PolicyContextUpdate() 244
7.4.2.3 ComputeAuthTimeout() 245
7.4.2.4 PolicyDigestClear() 245
7.5 NV Command Support (NV_spt.c) 248
7.5.1 Includes 248
7.5.2 Functions 248
7.5.2.1 NvReadAccessChecks() 248
7.5.2.2 NvWriteAccessChecks() 249
7.5.2.3 NvClearOrderly() 249
7.5.2.4 NvIsPinPassIndex() 250
7.6 Object Command Support (Object_spt.c) 251
7.6.1 Includes 251
7.6.2 Local Functions 251
7.6.2.1 GetIV2BSize() 251
7.6.2.2 ComputeProtectionKeyParms() 251
7.6.2.3 ComputeOuterIntegrity() 252
7.6.2.4 ComputeInnerIntegrity() 253
7.6.2.5 ProduceInnerIntegrity() 253
7.6.2.6 CheckInnerIntegrity() 254
7.6.3 Public Functions 255
7.6.3.1 AdjustAuthSize() 255
7.6.3.2 AreAttributesForParent() 255
7.6.3.3 CreateChecks() 255
7.6.3.4 SchemeChecks 256
7.6.3.5 PublicAttributesValidation() 260
7.6.3.6 FillInCreationData() 261
7.6.3.7 GetSeedForKDF() 262
7.6.3.8 ProduceOuterWrap() 263
7.6.3.9 UnwrapOuter() 264
7.6.3.10 MarshalSensitive() 265
7.6.3.11 SensitiveToPrivate() 266
7.6.3.12 PrivateToSensitive() 267
7.6.3.13 SensitiveToDuplicate() 268
7.6.3.14 DuplicateToSensitive() 270
7.6.3.15 SecretToCredential() 272
7.6.3.16 CredentialToSecret() 273
7.6.3.17 MemoryRemoveTrailingZeros() 274
7.6.3.18 SetLabelAndContext() 274
7.6.3.19 UnmarshalToPublic() 275
7.6.3.20 ObjectSetExternal() 275
7.7 Encrypt Decrypt Support (EncryptDecrypt_spt.c) 277
7.8 ACT Support (ACT_spt.c) 279

 Copyright © TCG 2006-2020

7.8.1 Introduction 279
7.8.2 Includes 279
7.8.3 Functions 279
7.8.3.1 _ActResume() 279
7.8.3.2 ActStartup() 279
7.8.3.3 _ActSaveState() 280
7.8.3.4 ActGetSignaled() 280
7.8.3.5 ActShutdown() 280
7.8.3.6 ActIsImplemented() 281
7.8.3.7 ActCounterUpdate() 281
7.8.3.8 ActGetCapabilityData() 282
8 Subsystem 284
8.1 CommandAudit.c 284
8.1.1 Introduction 284
8.1.2 Includes 284
8.1.3 Functions 284
8.1.3.1 CommandAuditPreInstall_Init() 284
8.1.3.2 CommandAuditStartup() 284
8.1.3.3 CommandAuditSet() 285
8.1.3.4 CommandAuditClear() 285
8.1.3.5 CommandAuditIsRequired() 286
8.1.3.6 CommandAuditCapGetCCList() 286
8.1.3.7 CommandAuditGetDigest 287
8.2 DA.c 289
8.2.1 Introduction 289
8.2.2 Includes and Data Definitions 289
8.2.3 Functions 289
8.2.3.1 DAPreInstall_Init() 289
8.2.3.2 DAStartup() 289
8.2.3.3 DARegisterFailure() 290
8.2.3.4 DASelfHeal() 291
8.3 Hierarchy.c 293
8.3.1 Introduction 293
8.3.2 Includes 293
8.3.3 Functions 293
8.3.3.1 HierarchyPreInstall() 293
8.3.3.2 HierarchyStartup() 294
8.3.3.3 HierarchyGetProof() 294
8.3.3.4 HierarchyGetPrimarySeed() 295
8.3.3.5 HierarchyIsEnabled() 295
8.4 NvDynamic.c 297
8.4.1 Introduction 297
8.4.2 Includes, Defines and Data Definitions 297
8.4.3 Local Functions 297
8.4.3.1 NvNext() 297
8.4.3.2 NvNextByType() 298
8.4.3.3 NvNextIndex() 298
8.4.3.4 NvNextEvict() 299
8.4.3.5 NvGetEnd() 299
8.4.3.6 NvGetFreeBytes 299
8.4.3.7 NvTestSpace() 299

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

8.4.3.8 NvWriteNvListEnd() 300
8.4.3.9 NvAdd() 301
8.4.3.10 NvDelete() 302

8.4.4 RAM-based NV Index Data Access Functions 303

8.4.4.1 Introduction 303
8.4.4.2 NvRamNext() 303
8.4.4.3 NvRamGetEnd() 303
8.4.4.4 NvRamTestSpaceIndex() 304
8.4.4.5 NvRamGetIndex() 304
8.4.4.6 NvUpdateIndexOrderlyData() 304
8.4.4.7 NvAddRAM() 305
8.4.4.8 NvDeleteRAM() 305
8.4.4.9 NvReadIndex() 306
8.4.4.10 NvReadObject() 306
8.4.4.11 NvFindEvict() 306
8.4.4.12 NvIndexIsDefined() 307
8.4.4.13 NvConditionallyWrite() 307
8.4.4.14 NvReadNvIndexAttributes() 308
8.4.4.15 NvReadRamIndexAttributes() 308
8.4.4.16 NvWriteNvIndexAttributes() 308
8.4.4.17 NvWriteRamIndexAttributes() 308

8.4.5 Externally Accessible Functions 309

8.4.5.1 NvIsPlatformPersistentHandle() 309
8.4.5.2 NvIsOwnerPersistentHandle() 309
8.4.5.3 NvIndexIsAccessible() 309
8.4.5.4 NvGetEvictObject() 310
8.4.5.5 NvIndexCacheInit() 311
8.4.5.6 NvGetIndexData() 311
8.4.5.7 NvHashIndexData() 312
8.4.5.8 NvGetUINT64Data() 312
8.4.5.9 NvWriteIndexAttributes() 313
8.4.5.10 NvWriteIndexAuth() 313
8.4.5.11 NvGetIndexInfo() 314
8.4.5.12 NvWriteIndexData() 314
8.4.5.13 NvWriteUINT64Data() 316
8.4.5.14 NvGetIndexName() 316
8.4.5.15 NvGetNameByIndexHandle() 317
8.4.5.16 NvDefineIndex() 317
8.4.5.17 NvAddEvictObject() 318
8.4.5.18 NvDeleteIndex() 319
8.4.5.19 NvDeleteEvict() 319
8.4.5.20 NvFlushHierarchy() 320
8.4.5.21 NvSetGlobalLock() 321
8.4.5.22 InsertSort() 322
8.4.5.23 NvCapGetPersistent() 322
8.4.5.24 NvCapGetIndex() 323
8.4.5.25 NvCapGetIndexNumber() 324
8.4.5.26 NvCapGetPersistentNumber() 324
8.4.5.27 NvCapGetPersistentAvail() 325
8.4.5.28 NvCapGetCounterNumber() 325
8.4.5.29 NvSetStartupAttributes() 325
8.4.5.30 NvEntityStartup() 326
8.4.5.31 NvCapGetCounterAvail() 327
8.4.5.32 NvFindHandle() 328

8.4.6 NV Max Counter 328

 Copyright © TCG 2006-2020

8.4.6.1 Introduction 328
8.4.6.2 NvReadMaxCount() 328
8.4.6.3 NvUpdateMaxCount() 328
8.4.6.4 NvSetMaxCount() 329
8.4.6.5 NvGetMaxCount() 329

8.5 NvReserved.c 330

8.5.1 Introduction 330
8.5.2 Includes, Defines 330
8.5.3 Functions 330

8.5.3.1 NvInitStatic() 330
8.5.3.2 NvCheckState() 331
8.5.3.3 NvCommit 331
8.5.3.4 NvPowerOn() 331
8.5.3.5 NvManufacture() 332
8.5.3.6 NvRead() 332
8.5.3.7 NvWrite() 332
8.5.3.8 NvUpdatePersistent() 333
8.5.3.9 NvClearPersistent() 333
8.5.3.10 NvReadPersistent() 333

8.6 Object.c 334

8.6.1 Introduction 334
8.6.2 Includes and Data Definitions 334
8.6.3 Functions 334

8.6.3.1 ObjectFlush() 334
8.6.3.2 ObjectSetInUse() 334
8.6.3.3 ObjectStartup() 334
8.6.3.4 ObjectCleanupEvict() 335
8.6.3.5 IsObjectPresent() 335
8.6.3.6 ObjectIsSequence() 335
8.6.3.7 HandleToObject() 336
8.6.3.8 GetQualifiedName() 336
8.6.3.9 ObjectGetHierarchy() 337
8.6.3.10 GetHeriarchy() 337
8.6.3.11 FindEmptyObjectSlot() 338
8.6.3.12 ObjectAllocateSlot() 338
8.6.3.13 ObjectSetLoadedAttributes() 338
8.6.3.14 ObjectLoad() 340
8.6.3.15 AllocateSequenceSlot() 341
8.6.3.16 ObjectCreateHMACSequence() 342
8.6.3.17 ObjectCreateHashSequence() 342
8.6.3.18 ObjectCreateEventSequence() 343
8.6.3.19 ObjectTerminateEvent() 343
8.6.3.20 ObjectContextLoad() 344
8.6.3.21 FlushObject() 345
8.6.3.22 ObjectFlushHierarchy() 345
8.6.3.23 ObjectLoadEvict() 346
8.6.3.24 ObjectComputeName() 346
8.6.3.25 PublicMarshalAndComputeName() 347
8.6.3.26 ComputeQualifiedName() 347
8.6.3.27 ObjectIsStorage() 348
8.6.3.28 ObjectCapGetLoaded() 348
8.6.3.29 ObjectCapGetTransientAvail() 349
8.6.3.30 ObjectGetPublicAttributes() 350

8.7 PCR.c 351

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

8.7.1 Introduction 351
8.7.2 Includes, Defines, and Data Definitions 351
8.7.2.1 PCRBelongsPolicyGroup() 352
8.7.2.2 PCRBelongsTCBGroup() 352
8.7.2.3 PCRPolicyIsAvailable() 353
8.7.2.4 PCRGetAuthValue() 353
8.7.2.5 PCRGetAuthPolicy() 354
8.7.2.6 PCRSimStart() 354
8.7.2.7 GetSavedPcrPointer() 355
8.7.2.8 PcrIsAllocated() 356
8.7.2.9 GetPcrPointer() 356
8.7.2.10 IsPcrSelected() 357
8.7.2.11 FilterPcr() 357
8.7.2.12 PcrDrtm() 358
8.7.2.13 PCR_ClearAuth() 358
8.7.2.14 PCRStartup() 359
8.7.2.15 PCRStateSave() 360
8.7.2.16 PCRIsStateSaved() 361
8.7.2.17 PCRIsResetAllowed() 361
8.7.2.18 PCRChanged() 362
8.7.2.19 PCRIsExtendAllowed() 362
8.7.2.20 PCRExtend() 363
8.7.2.21 PCRComputeCurrentDigest() 363
8.7.2.22 PCRRead() 364
8.7.2.23 PCRAllocate() 365
8.7.2.24 PCRSetValue() 367
8.7.2.25 PCRResetDynamics 368
8.7.2.26 PCRCapGetAllocation() 368
8.7.2.27 PCRSetSelectBit() 369
8.7.2.28 PCRGetProperty() 369
8.7.2.29 PCRCapGetProperties() 371
8.7.2.30 PCRCapGetHandles() 372
8.8 PP.c 373
8.8.1 Introduction 373
8.8.2 Includes 373
8.8.3 Functions 373
8.8.3.1 PhysicalPresencePreInstall_Init() 373
8.8.3.2 PhysicalPresenceCommandSet() 373
8.8.3.3 PhysicalPresenceCommandClear() 374
8.8.3.4 PhysicalPresenceIsRequired() 374
8.8.3.5 PhysicalPresenceCapGetCCList() 374
8.9 Session.c 376
8.9.1 Introduction 376
8.9.2 Includes, Defines, and Local Variables 377
8.9.3 File Scope Function -- ContextIdSetOldest() 377
8.9.4 Startup Function -- SessionStartup() 378
8.9.5 Access Functions 378
8.9.5.1 SessionIsLoaded() 378
8.9.5.2 SessionIsSaved() 379
8.9.5.3 SequenceNumberForSavedContextIsValid() 380
8.9.5.4 SessionPCRValueIsCurrent() 380
8.9.5.5 SessionGet() 380
8.9.6 Utility Functions 381
8.9.6.1 ContextIdSessionCreate() 381

 Copyright © TCG 2006-2020

8.9.6.2 SessionCreate() 382
8.9.6.3 SessionContextSave() 384
8.9.6.4 SessionContextLoad() 385
8.9.6.5 SessionFlush() 387
8.9.6.6 SessionComputeBoundEntity() 387
8.9.6.7 SessionSetStartTime() 388
8.9.6.8 SessionResetPolicyData() 388
8.9.6.9 SessionCapGetLoaded() 389
8.9.6.10 SessionCapGetSaved() 390
8.9.6.11 SessionCapGetLoadedNumber() 391
8.9.6.12 SessionCapGetLoadedAvail() 391
8.9.6.13 SessionCapGetActiveNumber() 391
8.9.6.14 SessionCapGetActiveAvail() 392

8.10 Time.c 393

8.10.1 Introduction 393
8.10.2 Includes 393
8.10.3 Functions 393

8.10.3.1 TimePowerOn() 393
8.10.3.2 TimeNewEpoch() 393
8.10.3.3 TimeStartup() 393
8.10.3.4 TimeClockUpdate() 394
8.10.3.5 TimeUpdate() 394
8.10.3.6 TimeUpdateToCurrent() 395
8.10.3.7 TimeSetAdjustRate() 395
8.10.3.8 TimeGetMarshaled() 396
8.10.3.9 TimeFillInfo 396

9 Support 398

9.1 AlgorithmCap.c 398

9.1.1 Description 398
9.1.2 Includes and Defines 398
9.1.3 AlgorithmCapGetImplemented() 400
9.1.4 AlgorithmGetImplementedVector() 401

9.2 Bits.c 402

9.2.1 Introduction 402
9.2.2 Includes 402
9.2.3 Functions 402

9.2.3.1 TestBit() 402
9.2.3.2 SetBit() 402
9.2.3.3 ClearBit() 402

9.3 CommandCodeAttributes.c 404

9.3.1 Introduction 404
9.3.2 Includes and Defines 404
9.3.3 Command Attribute Functions 404

9.3.3.1 NextImplementedIndex() 404
9.3.3.2 GetClosestCommandIndex() 405
9.3.3.3 CommandCodeToComandIndex() 407
9.3.3.4 GetNextCommandIndex() 408
9.3.3.5 GetCommandCode() 408
9.3.3.6 CommandAuthRole() 409
9.3.3.7 EncryptSize() 409
9.3.3.8 DecryptSize() 410
9.3.3.9 IsSessionAllowed() 410

 Copyright © TCG 2006-2020

9.3.3.10 IsHandleInResponse() ... 410
9.3.3.11 IsWriteOperation() ... 410
9.3.3.12 IsReadOperation() ... 411
9.3.3.13 CommandCapGetCCList() ... 412
9.3.3.14 IsVendorCommand() ... 412

9.4 Entity.c ... 414

9.4.1 Description ... 414
9.4.2 Includes ... 414
9.4.3 Functions ... 414

9.4.3.1 EntityGetLoadStatus() ... 414
9.4.3.2 EntityGetAuthValue() ... 416
9.4.3.3 EntityGetAuthPolicy() ... 418
9.4.3.4 EntityGetName() ... 419
9.4.3.5 EntityGetHierarchy() ... 420

9.5 Global.c ... 422

9.5.1 Description ... 422
9.5.2 Defines and Includes ... 422

9.6 Handle.c ... 423

9.6.1 Description ... 423
9.6.2 Includes ... 423
9.6.3 Functions ... 423

9.6.3.1 HandleGetType() ... 423
9.6.3.2 NextPermanentHandle() ... 423
9.6.3.3 PermanentCapGetHandles() ... 424
9.6.3.4 PermanentHandleGetPolicy() ... 425

9.7 IoBuffers.c ... 426

9.7.1 Includes and Data Definitions ... 426
9.7.2 Buffers and Functions ... 426

9.7.2.1 MemoryIoBufferAllocationReset() ... 426
9.7.2.2 MemoryIoBufferZero() ... 426
9.7.2.3 MemoryGetInBuffer() ... 426
9.7.2.4 MemoryGetOutBuffer() ... 427
9.7.2.5 IsLabelProperlyFormatted() ... 427

9.8 Locality.c ... 428

9.8.1 Includes ... 428
9.8.2 LocalityGetAttributes() ... 428

9.9 Manufacture.c ... 429

9.9.1 Description ... 429
9.9.2 Includes and Data Definitions ... 429
9.9.3 Functions ... 429

9.9.3.1 TPM_Manufacture() ... 429
9.9.3.2 TPM_TearDown() ... 430
9.9.3.3 TpmEndSimulation() ... 431

9.10 Marshal.c ... 432

9.10.1 Introduction ... 432
9.10.2 Unmarshal and Marshal a Value ... 432
9.10.3 Unmarshal and Marshal a Union ... 433
9.10.4 Unmarshal and Marshal a Structure ... 435
9.10.5 Unmarshal and Marshal an Array ... 436

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

9.10.6 TPM2B Handling 438
9.10.7 Table Marshal Headers 438
9.10.7.1 TableMarshal.h 438
9.10.7.2 TableMarshalData.h 442
9.10.7.3 TableMarshalDefines.h 475
9.10.7.4 TableMarshalTypes.h 497
9.10.8 Table Marshal Source 518
9.10.8.1 TableDrivenMarshal.c 518
9.10.8.2 TableMarshalData.c 535
9.11 MathOnByteBuffers.c 557
9.11.1 Introduction 557
9.11.2 Functions 557
9.11.2.1 UnsignedCmpB 557
9.11.2.2 SignedCompareB() 557
9.11.2.3 ModExpB 558
9.11.2.4 DivideB() 559
9.11.2.5 AdjustNumberB() 560
9.11.2.6 ShiftLeft() 560
9.12 Memory.c 562
9.12.1 Description 562
9.12.2 Includes and Data Definitions 562
9.12.3 Functions 562
9.12.3.1 MemoryCopy() 562
9.12.3.2 MemoryEqual() 562
9.12.3.3 MemoryCopy2B() 563
9.12.3.4 MemoryConcat2B() 563
9.12.3.5 MemoryEqual2B() 563
9.12.3.6 MemorySet() 564
9.12.3.7 MemoryPad2B() 564
9.12.3.8 Uint16ToByteArray() 564
9.12.3.9 Uint32ToByteArray() 564
9.12.3.10 Uint64ToByteArray() 565
9.12.3.11 ByteArrayToUint8() 565
9.12.3.12 ByteArrayToUint16() 565
9.12.3.13 ByteArrayToUint32() 565
9.12.3.14 ByteArrayToUint64() 566
9.13 Power.c 567
9.13.1 Description 567
9.13.2 Includes and Data Definitions 567
9.13.3 Functions 567
9.13.3.1 TPMInit() 567
9.13.3.2 TPMRegisterStartup() 567
9.13.3.3 TPMIsStarted() 567
9.14 PropertyCap.c 569
9.14.1 Description 569
9.14.2 Includes 569
9.14.3 Functions 569
9.14.3.1 TPMPropertyIsDefined() 569
9.14.3.2 TPMCapGetProperties() 576
9.15 Response.c 578

 Copyright © TCG 2006-2020

9.15.1 Description 578
9.15.2 Includes and Defines 578
9.15.3 BuildResponseHeader() 578

9.16 ResponseCodeProcessing.c 579

9.16.1 Description 579
9.16.2 Includes and Defines 579
9.16.3 RcSafeAddToResult() 579

9.17 TpmFail.c 580

9.17.1 Includes, Defines, and Types 580
9.17.2 Typedefs 580
9.17.3 Local Functions 581

9.17.3.1 MarshalUint16() 581
9.17.3.2 MarshalUint32() 581
9.17.3.3 Unmarshal32() 581
9.17.3.4 Unmarshal16() 582

9.17.4 Public Functions 582

9.17.4.1 SetForceFailureMode() 582
9.17.4.2 TpmLogFailure() 582
9.17.4.3 TpmFail() 583
9.17.4.4 TpmFailureMode 583
9.17.4.5 UnmarshalFail() 586

10 Cryptographic Functions 587

10.1 Headers 587

10.1.1 BnValues.h 587

10.1.1.1 Introduction 587
10.1.1.2 Defines 587

10.1.2 CryptEcc.h 592

10.1.2.1 Introduction 592
10.1.2.2 Structures 592

10.1.3 CryptHash.h 593

10.1.3.1 Introduction 593
10.1.3.2 Hash-related Structures 593
10.1.3.3 HMAC State Structures 596

10.1.4 CryptRand.h 598

10.1.4.1 Introduction 598
10.1.4.2 DRBG Structures and Defines 598

10.1.5 CryptRsa.h 601
10.1.6 CryptTest.h 602
10.1.7 HashTestData.h 603
10.1.8 KdfTestData.h 605
10.1.9 RsaTestData.h 606
10.1.10 SelfTest.h 612

10.1.10.1 Introduction 612
10.1.10.2 Defines 612

10.1.11 SupportLibraryFunctionPrototypes_fp.h 613

10.1.11.1 Introduction 613
10.1.11.2 SupportLibInit() 614
10.1.11.3 MathLibraryCompatibililtyCheck() 614

10.1.11.4 BnModMult() 614
10.1.11.5 BnMult() 614
10.1.11.6 BnDiv() 614
10.1.11.7 BnMod() 614
10.1.11.8 BnGcd() 614
10.1.11.9 BnModExp() 615
10.1.11.10 BnModInverse() 615
10.1.11.11 BnEccModMult() 615
10.1.11.12 BnEccModMult2() 615
10.1.11.13 BnEccAdd() 615
10.1.11.14 BnCurveInitialize() 615
10.1.11.14.1 BnCurveFree() 615
10.1.12 SymmetricTestData.h 617
10.1.13 SymmetricTest.h 620
10.1.13.1 Introduction 620
10.1.13.2 Symmetric Test Structures 620
10.1.14 EccTestData.h 621
10.1.15 CryptSym.h 624
10.1.15.1 Introduction 624
10.1.15.2 Includes, Defines, and Typedefs 624
10.1.16 OIDs.h 626
10.1.17 PRNG_TestVectors.h 629
10.1.18 TpmAsn1.h 630
10.1.18.1 Introduction 630
10.1.18.2 Includes 630
10.1.18.3 Defined Constants 630
10.1.18.3.1 ASN.1 Universal Types (Class 00b 630
10.1.18.4 Macros 631
10.1.18.4.1 Unmarshaling Macros 631
10.1.18.4.2 Marshaling Macros 631
10.1.18.5 Structures 631
10.1.19 X509.h 631
10.1.19.1 Introduction 631
10.1.19.2 Includes 632
10.1.19.3 Defined Constants 632
10.1.19.3.1 X509 Application-specific types 632
10.1.19.4 Structures 632
10.1.19.5 Global X509 Constants 632
10.1.20 TpmAlgorithmDefines.h 633
10.2 Source 640
10.2.1 AlgorithmTests.c 640
10.2.1.1 Introduction 640
10.2.1.2 Includes and Defines 640
10.2.1.3 Hash Tests 640
10.2.1.3.1 Description 640
10.2.1.3.2 TestHash() 640
10.2.1.4 Symmetric Test Functions 642

 Copyright © TCG 2006-2020

10.2.1.4.1 MakeIv() 642
10.2.1.4.2 TestSymmetricAlgorithm() 642
10.2.1.4.3 AllSymsAreDone() 643
10.2.1.4.4 AllModesAreDone() 643
10.2.1.4.5 TestSymmetric() 643

10.2.1.5 RSA Tests 644

10.2.1.5.1 Introduction 645
10.2.1.5.2 RsaKeyInitialize() 645
10.2.1.5.3 TestRsaEncryptDecrypt() 645
10.2.1.5.4 TestRsaSignAndVerify() 647
10.2.1.5.5 TestRSA() 648

10.2.1.6 ECC Tests 649

10.2.1.6.1 LoadEccParameter() 649
10.2.1.6.2 LoadEccPoint() 649
10.2.1.6.3 TestECDH() 649
10.2.1.6.4 TestEccSignAndVerify() 650
10.2.1.6.5 TestKDFa() 651
10.2.1.6.6 TestEcc() 651
10.2.1.6.7 TestAlgorithm() 652

10.2.2 BnConvert.c 656

10.2.2.1 Introduction 656
10.2.2.2 Includes 656
10.2.2.3 Functions 656

10.2.2.3.1 BnFromBytes() 656
10.2.2.3.2 BnFrom2B() 657
10.2.2.3.3 BnFromHex() 657
10.2.2.3.4 BnToBytes() 658
10.2.2.3.5 BnTo2B() 659
10.2.2.3.6 BnPointFrom2B() 659
10.2.2.3.7 BnPointTo2B() 659

10.2.3 BnMath.c 661

10.2.3.1 Introduction 661
10.2.3.2 Includes 661

10.2.3.2.1 CarryProp() 662
10.2.3.2.2 BnAdd() 662
10.2.3.2.3 BnAddWord() 663
10.2.3.2.4 SubSame() 663
10.2.3.2.5 BorrowProp() 663
10.2.3.2.6 BnSub() 664
10.2.3.2.7 BnSubWord() 664
10.2.3.2.8 BnUnsignedCmp() 665
10.2.3.2.9 BnUnsignedCmpWord() 665
10.2.3.2.10 BnModWord() 666
10.2.3.2.11 Msb() 666
10.2.3.2.12 BnMsb() 666
10.2.3.2.13 BnSizeInBits() 667
10.2.3.2.14 BnSetWord() 667
10.2.3.2.15 BnSetBit() 667
10.2.3.2.16 BnTestBit() 668
10.2.3.2.17 BnMaskBits() 668
10.2.3.2.18 BnShiftRight() 669
10.2.3.2.19 BnGetRandomBits() 669
10.2.3.2.20 BnGenerateRandomInRange() 670

 Copyright © TCG 2006-2020

10.2.4 BnMemory.c ... 671
10.2.4.1 Introduction ... 671
10.2.4.2 Includes ... 671
10.2.4.3 Functions ... 671
10.2.4.3.1 BnSetTop() ... 671
10.2.4.3.2 BnClearTop() ... 671
10.2.4.3.3 BnInitializeWord() ... 672
10.2.4.3.4 BnInit() ... 672
10.2.4.3.5 BnCopy() ... 672
10.2.4.3.6 BnPointCopy() ... 673
10.2.4.3.7 BnInitializePoint() ... 673
10.2.5 CryptCmac.c ... 674
10.2.5.1 Introduction ... 674
10.2.5.2 Includes, Defines, and Typedefs ... 674
10.2.5.3 Functions ... 674
10.2.5.3.1 CryptCmacStart() ... 674
10.2.5.3.2 CryptCmacData() ... 674
10.2.5.3.3 CryptCmacEnd() ... 675
10.2.6 CryptUtil.c ... 677
10.2.6.1 Introduction ... 677
10.2.6.2 Includes ... 677
10.2.6.3 Hash/HMAC Functions ... 677
10.2.6.3.1 CryptHmacSign() ... 677
10.2.6.3.2 CryptHMACVerifySignature() ... 677
10.2.6.3.3 CryptGenerateKeyedHash() ... 678
10.2.6.3.4 CryptIsSchemeAnonymous() ... 679
10.2.6.4 Symmetric Functions ... 679
10.2.6.4.1 ParmDecryptSym() ... 679
10.2.6.4.2 ParmEncryptSym() ... 680
10.2.6.4.3 CryptGenerateKeySymmetric() ... 681
10.2.6.4.4 CryptXORObfuscation() ... 682
10.2.6.5 Initialization and shut down ... 682
10.2.6.5.1 CryptInit() ... 682
10.2.6.5.2 CryptStartup() ... 683
10.2.6.6 Algorithm-Independent Functions ... 684
10.2.6.6.1 Introduction ... 684
10.2.6.6.2 CryptIsAsymAlgorithm() ... 684
10.2.6.6.3 CryptSecretEncrypt() ... 684
10.2.6.6.4 CryptSecretDecrypt() ... 686
10.2.6.6.5 CryptParameterEncryption() ... 689
10.2.6.6.6 CryptParameterDecryption() ... 690
10.2.6.6.7 CryptComputeSymmetricUnique() ... 691
10.2.6.6.8 CryptCreateObject() ... 692
10.2.6.6.9 CryptGetSignHashAlg() ... 694
10.2.6.6.10 CryptIsSplitSign() ... 695
10.2.6.6.11 CryptIsAsymSignScheme() ... 695
10.2.6.6.12 CryptIsAsymDecryptScheme() ... 696
10.2.6.6.13 CryptSelectSignScheme() ... 697
10.2.6.6.14 CryptSign() ... 699
10.2.6.6.15 CryptValidateSignature() ... 700
10.2.6.6.16 CryptGetTestResult ... 701

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.2.6.6.17 CryptValidateKeys() ... 701
10.2.6.6.18 CryptSelectMac() ... 704
10.2.6.6.19 CryptMacIsValidForKey() ... 705
10.2.6.6.20 CryptSmacIsValidAlg() ... 705
10.2.6.6.21 CryptSymModeIsValid() ... 706

10.2.7 CryptSelfTest.c ... 707

10.2.7.1 Introduction ... 707
10.2.7.2 Functions ... 707

10.2.7.2.1 RunSelfTest() ... 707
10.2.7.2.2 CryptSelfTest() ... 707
10.2.7.2.3 CryptIncrementalSelfTest() ... 708
10.2.7.2.4 CryptInitializeToTest() ... 709
10.2.7.2.5 CryptTestAlgorithm() ... 709

10.2.8 CryptEccData.c ... 711
10.2.9 CryptDes.c ... 721

10.2.9.1 Introduction ... 721
10.2.9.2 Includes, Defines, and Typedefs ... 721

10.2.9.2.1 CryptDesIsWeakKey() ... 722
10.2.9.2.2 CryptDesValidateKey() ... 722
10.2.9.2.3 CryptGenerateKeyDes() ... 723

10.2.10 CryptEccKeyExchange.c ... 724

10.2.10.1 Introduction ... 724
10.2.10.2 Functions ... 724

10.2.10.2.1 avf1() ... 724
10.2.10.2.2 C_2_2_MQV() ... 724
10.2.10.2.3 C_2_2_ECDH() ... 726
10.2.10.2.4 CryptEcc2PhaseKeyExchange() ... 726
10.2.10.2.5 ComputeWForSM2() ... 727
10.2.10.2.6 avfSm2() ... 728
10.2.10.2.7 SM2KeyExchange() ... 728

10.2.11 CryptEccMain.c ... 730

10.2.11.1 Includes and Defines ... 730
10.2.11.2 Functions ... 730

10.2.11.2.1 CryptEccInit() ... 730
10.2.11.2.2 CryptEccStartup() ... 730
10.2.11.2.3 ClearPoint2B(generic ... 730
10.2.11.2.4 CryptEccGetParametersByCurveId() ... 731
10.2.11.2.5 CryptEccGetKeySizeForCurve() ... 731
10.2.11.2.6 GetCurveData() ... 731
10.2.11.2.7 CryptEccGetOID() ... 732
10.2.11.2.8 CryptEccGetCurveByIndex() ... 732
10.2.11.2.9 CryptEccGetParameter() ... 732
10.2.11.2.10 CryptCapGetECCCurve() ... 733
10.2.11.2.11 CryptGetCurveSignScheme() ... 734
10.2.11.2.12 CryptGenerateR() ... 734
10.2.11.2.13 CryptCommit() ... 736
10.2.11.2.14 CryptEndCommit() ... 736
10.2.11.2.15 CryptEccGetParameters() ... 736
10.2.11.2.16 BnGetCurvePrime() ... 737
10.2.11.2.17 BnGetCurveOrder() ... 737
10.2.11.2.18 BnIsOnCurve() ... 737
10.2.11.2.19 BnIsValidPrivateEcc() ... 738

 Copyright © TCG 2006-2020

10.2.11.2.20 BnPointMul() 738
10.2.11.2.21 BnEccGetPrivate() 739
10.2.11.2.22 BnEccGenerateKeyPair() 740
10.2.11.2.23 CryptEccNewKeyPair 740
10.2.11.2.24 CryptEccPointMultiply() 741
10.2.11.2.25 CryptEccIsPointOnCurve() 742
10.2.11.2.26 CryptEccGenerateKey() 742

10.2.12 CryptEccSignature.c 744

10.2.12.1 Includes and Defines 744
10.2.12.2 Utility Functions 744

10.2.12.2.1 EcdsaDigest() 744
10.2.12.2.2 BnSchnorrSign() 744

10.2.12.3 Signing Functions 745

10.2.12.3.1 BnSignEcdsa() 745
10.2.12.3.2 BnSignEcdaa() 746
10.2.12.3.3 SchnorrReduce() 748
10.2.12.3.4 SchnorrEcc() 748
10.2.12.3.5 BnHexEqual() 749
10.2.12.3.6 BnSignEcSm2() 750
10.2.12.3.7 CryptEccSign() 751
10.2.12.3.8 BnValidateSignatureEcdsa() 753
10.2.12.3.9 BnValidateSignatureEcSm2() 754
10.2.12.3.10 BnValidateSignatureEcSchnorr() 755
10.2.12.3.11 CryptEccValidateSignature() 756
10.2.12.3.12 CryptEccCommitCompute() 757

10.2.13 CryptHash.c 759

10.2.13.1 Description 759
10.2.13.2 Includes, Defines, and Types 759

10.2.13.2.1 CryptHashStartup() 760

10.2.13.3 Hash Information Access Functions 760

10.2.13.3.1 Introduction 760
10.2.13.3.2 CryptGetHashDef() 760
10.2.13.3.3 CryptHashIsValidAlg() 760
10.2.13.3.4 CryptHashGetAlgByIndex() 761
10.2.13.3.5 CryptHashGetDigestSize() 761
10.2.13.3.6 CryptHashGetBlockSize() 761
10.2.13.3.7 CryptHashGetOid() 762
10.2.13.3.8 CryptHashGetContextAlg() 762

10.2.13.4 State Import and Export 762

10.2.13.4.1 CryptHashCopyState 762
10.2.13.4.2 CryptHashExportState() 763
10.2.13.4.3 CryptHashImportState() 763

10.2.13.5 State Modification Functions 764

10.2.13.5.1 HashEnd() 764
10.2.13.5.2 CryptHashStart() 765
10.2.13.5.3 CryptDigestUpdate() 765
10.2.13.5.4 CryptHashEnd() 766
10.2.13.5.5 CryptHashBlock() 766
10.2.13.5.6 CryptDigestUpdate2B() 766
10.2.13.5.7 CryptHashEnd2B() 767
10.2.13.5.8 CryptDigestUpdateInt() 767

 Copyright © TCG 2006-2020

10.2.13.6 HMAC Functions ... 768
10.2.13.6.1 CryptHmacStart() ... 768
10.2.13.6.2 CryptHmacEnd() ... 769
10.2.13.6.3 CryptHmacStart2B() ... 769
10.2.13.6.4 CryptHmacEnd2B() ... 770
10.2.13.7 Mask and Key Generation Functions ... 770
10.2.13.7.1 CryptMGF1() ... 770
10.2.13.7.2 CryptKDFa() ... 771
10.2.13.7.3 CryptKDFe() ... 773
10.2.14 CryptPrime.c ... 775
10.2.14.1 Introduction ... 775
10.2.14.1.1 IsPrimeInt() ... 775
10.2.14.1.2 BnIsProbablyPrime() ... 776
10.2.14.1.3 MillerRabinRounds() ... 776
10.2.14.1.4 MillerRabin() ... 777
10.2.14.1.5 RsaCheckPrime() ... 778
10.2.14.1.6 AdjustPrimeCandiate() ... 779
10.2.14.1.7 BnGeneratePrimeForRSA() ... 780
10.2.15 CryptPrimeSieve.c ... 781
10.2.15.1 Includes and defines ... 781
10.2.15.1.1 RsaNextPrime() ... 781
10.2.15.1.2 BitsInArry() ... 783
10.2.15.1.3 FindNthSetBit() ... 783
10.2.15.1.4 PrimeSelectWithSieve() ... 786
10.2.16 CryptRand.c ... 790
10.2.16.1 Introduction ... 790
10.2.16.1.1 DfCompute() ... 791
10.2.16.1.2 DfStart() ... 791
10.2.16.1.3 DfUpdate() ... 792
10.2.16.1.4 DfEnd() ... 792
10.2.16.1.5 DfBuffer() ... 793
10.2.16.1.6 DRBG_GetEntropy() ... 793
10.2.16.1.7 IncrementIv() ... 794
10.2.16.1.8 EncryptDRBG() ... 794
10.2.16.1.9 DRBG_Update() ... 795
10.2.16.1.10 DRBG_Reseed() ... 796
10.2.16.1.11 DRBG_SelfTest() ... 797
10.2.16.2 Public Interface ... 798
10.2.16.2.1 Description ... 798
10.2.16.2.2 CryptRandomStir() ... 798
10.2.16.2.3 CryptRandomGenerate() ... 799
10.2.16.2.4 DRBG_InstantiateSeededKdf() ... 799
10.2.16.2.5 DRBG_AdditionalData() ... 800
10.2.16.2.6 DRBG_InstantiateSeeded() ... 800
10.2.16.2.7 CryptRandStartup() ... 801
10.2.16.2.8 DRBG_Generate() ... 802
10.2.16.2.9 DRBG_Instantiate() ... 804
10.2.16.2.10 DRBG_Uninstantiate() ... 805
10.2.17 CryptRsa.c ... 806
10.2.17.1 Introduction ... 806

 Copyright © TCG 2006-2020

10.2.17.2 Includes 806
10.2.17.3 Obligatory Initialization Functions 806

10.2.17.3.1 CryptRsaInit() 806
10.2.17.3.2 CryptRsaStartup() 806

10.2.17.4 Internal Functions 806

10.2.17.4.1 RsaInitializeExponent() 806
10.2.17.4.2 MakePgreaterThanQ() 807
10.2.17.4.3 PackExponent() 807
10.2.17.4.4 UnpackExponent() 808
10.2.17.4.5 ComputePrivateExponent() 808
10.2.17.4.6 RsaPrivateKeyOp() 809
10.2.17.4.7 RSAEP() 809
10.2.17.4.8 RSADP() 810
10.2.17.4.9 OaepEncode() 811
10.2.17.4.10 OaepDecode() 812
10.2.17.4.11 PKCS1v1_5Encode() 813
10.2.17.4.12 RSAES_Decode() 814
10.2.17.4.13 CryptRsaPssSaltSize() 815
10.2.17.4.14 PssEncode() 815
10.2.17.4.15 PssDecode() 816
10.2.17.4.16 MakeDerTag() 818
10.2.17.4.17 RSASSA_Encode() 819
10.2.17.4.18 RSASSA_Decode() 820

10.2.17.5 Externally Accessible Functions 821

10.2.17.5.1 CryptRsaSelectScheme() 821
10.2.17.5.2 CryptRsaLoadPrivateExponent() 821
10.2.17.5.3 CryptRsaEncrypt() 822
10.2.17.5.4 CryptRsaDecrypt() 824
10.2.17.5.5 CryptRsaSign() 825
10.2.17.5.6 CryptRsaValidateSignature() 826
10.2.17.5.7 CryptRsaGenerateKey() 827

10.2.18 CryptSmac.c 830

10.2.18.1 Introduction 830
10.2.18.2 Includes, Defines, and Typedefs 830

10.2.18.2.1 CryptSmacStart() 830
10.2.18.2.2 CryptMacStart() 830
10.2.18.2.3 CryptMacEnd() 831
10.2.18.2.4 CryptMacEnd2B() 831

10.2.19 CryptSym.c 832

10.2.19.1 Introduction 832
10.2.19.2 Includes, Defines, and Typedefs 832
10.2.19.3 Initialization and Data Access Functions 832

10.2.19.3.1 CryptSymInit() 832
10.2.19.3.2 CryptSymStartup() 832
10.2.19.3.3 CryptGetSymmetricBlockSize() 832

10.2.19.4 Symmetric Encryption 833

10.2.19.4.1 CryptSymmetricDecrypt() 836
10.2.19.4.2 CryptSymKeyValidate() 839

10.2.20 PrimeData.c 840
10.2.21 RsaKeyCache.c 847

10.2.21.1 Introduction 847

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.2.21.2 Includes, Types, Locals, and Defines....847
10.2.21.2.1 InitializeKeyCache()....848
10.2.21.2.2 KeyCacheLoaded()....849
10.2.21.2.3 GetCachedRsaKey()....850
10.2.22 Ticket.c....851
10.2.22.1 Introduction....851
10.2.22.2 Includes....851
10.2.22.3 Functions....851
10.2.22.3.1 TicketIsSafe()....851
10.2.22.3.2 TicketComputeVerified()....851
10.2.22.3.3 TicketComputeAuth()....852
10.2.22.3.4 TicketComputeHashCheck()....853
10.2.22.3.5 TicketComputeCreation()....853
10.2.23 TpmAsn1.c....855
10.2.23.1 Includes....855
10.2.23.2 Unmarshaling Functions....855
10.2.23.2.1 ASN1UnmarshalContextInitialize()....855
10.2.23.2.2 ASN1DecodeLength()....855
10.2.23.2.3 ASN1NextTag()....856
10.2.23.2.4 ASN1GetBitStringValue()....857
10.2.23.3 Marshaling Functions....858
10.2.23.3.1 Introduction....858
10.2.23.3.2 ASN1InitialializeMarshalContext()....858
10.2.23.3.3 ASN1StartMarshalContext()....858
10.2.23.3.4 ASN1EndMarshalContext()....859
10.2.23.3.5 ASN1EndEncapsulation()....859
10.2.23.3.6 ASN1PushByte()....860
10.2.23.3.7 ASN1PushBytes()....860
10.2.23.3.8 ASN1PushNull()....860
10.2.23.3.9 ASN1PushLength()....861
10.2.23.3.10 ASN1PushTagAndLength()....861
10.2.23.3.11 ASN1PushTaggedOctetString()....862
10.2.23.3.12 ASN1PushUINT()....862
10.2.23.3.13 ASN1PushInteger....862
10.2.23.3.14 ASN1PushOID()....863
10.2.24 X509_ECC.c....864
10.2.24.1 Includes....864
10.2.24.2 Functions....864
10.2.24.2.1 X509PushPoint()....864
10.2.24.2.2 X509AddSigningAlgorithmECC()....864
10.2.24.2.3 X509AddPublicECC()....865
10.2.25 X509_RSA.c....867
10.2.25.1 Includes....867
10.2.25.2 Functions....867
10.2.25.2.1 X509AddSigningAlgorithmRSA()....867
10.2.25.2.2 X509AddPublicRSA()....869
10.2.26 X509_spt.c....871
10.2.26.1 Includes....871
10.2.26.2 Unmarshaling Functions....871

 Copyright © TCG 2006-2020

10.2.26.2.1 X509FindExtensionByOID() ... 871
10.2.26.2.2 X509GetExtensionBits() ... 872
10.2.26.2.3 X509ProcessExtensions() ... 872
10.2.26.3 Marshaling Functions ... 874
10.2.26.3.1 X509AddSigningAlgorithm() ... 874
10.2.26.3.2 X509AddPublicKey() ... 874
10.2.26.3.3 X509PushAlgorithmIdentifierSequence() ... 875
10.2.27 AC_spt.c ... 876
10.2.27.1 Includes ... 876
10.2.27.1.1 AcToCapabilities() ... 876
10.2.27.1.2 AcIsAccessible() ... 876
10.2.27.1.3 AcCapabilitiesGet() ... 876
10.2.27.1.4 AcSendObject() ... 877
Annex A (informative) Implementation Dependent ... 879
A.1 Introduction ... 879
A.2 TpmProfile.h ... 879
A.3 TpmSizeChecks.c ... 890
A.3.1. Includes, Defines, and Types ... 890
Annex B (informative) Library-Specific ... 894
B.1 Introduction ... 894
B.2 OpenSSL-Specific Files ... 895
B.2.1. Introduction ... 895
B.2.2. Header Files ... 895
B.2.2.1. TpmToOsslHash.h ... 895
B.2.2.1.1. Introduction ... 895
B.2.2.1.2. Links to the OpenSSL HASH code ... 895
B.2.2.2. TpmToOsslMath.h ... 898
B.2.2.2.1. Introduction ... 898
B.2.2.2.2. Macros and Defines ... 898
B.2.2.3. TpmToOsslSym.h ... 900
B.2.2.3.1. Introduction ... 900
B.2.2.3.2. Links to the OpenSSL symmetric algorithms ... 900
B.2.2.3.3. Links to the OpenSSL AES code ... 900
B.2.2.3.4. Links to the OpenSSL DES code ... 901
B.2.2.3.5. Links to the OpenSSL SM4 code ... 901
B.2.2.3.6. Links to the OpenSSL CAMELLIA code ... 901
B.2.3. Source Files ... 902
B.2.3.1. TpmToOsslDesSupport.c ... 902
B.2.3.1.1. Introduction ... 902
B.2.3.1.2. Defines and Includes ... 902
B.2.3.1.3. Functions ... 902
B.2.3.2. TpmToOsslMath.c ... 904
B.2.3.2.1. Introduction ... 904
B.2.3.2.2. Includes and Defines ... 904
B.2.3.2.3. Functions ... 904
B.2.3.2.4. BnEccAdd() ... 914
B.2.3.3. TpmToOsslSupport.c ... 915

 Copyright © TCG 2006-2020

B.2.3.3.1. Introduction 915
B.2.3.3.2. Defines and Includes 915

Annex C (informative) Simulation Environment 917

C.1 Introduction 917
C.2 Cancel.c 917

C.2.1. Description 917
C.2.2. Includes, Typedefs, Structures, and Defines 917
C.2.3. Functions 917

C.2.3.1. _plat__IsCanceled() 917
C.2.3.2. _plat__SetCancel() 917
C.2.3.3. _plat__ClearCancel() 918

C.3 Clock.c 919

C.3.1. Description 919
C.3.2. Includes and Data Definitions 919
C.3.3. Simulator Functions 919

C.3.3.1. Introduction 919
C.3.3.2. _plat__TimerReset() 919
C.3.3.3. _plat__TimerRestart() 919

C.3.4. Functions Used by TPM 920

C.3.4.1. Introduction 920
C.3.4.2. _plat__TimerRead() 920
C.3.4.3. _plat__TimerWasReset() 921
C.3.4.4. _plat__TimerWasStopped() 922
C.3.4.5. _plat__ClockAdjustRate() 922

C.4 Entropy.c 924

C.4.1. Includes and Local Values 924

C.4.1.1. _plat__GetEntropy() 924

C.5 LocalityPlat.c 927

C.5.1. Includes 927
C.5.2. Functions 927

C.5.2.1. _plat__LocalityGet() 927
C.5.2.2. _plat__LocalitySet() 927

C.6 NVMem.c 928

C.6.1. Description 928
C.6.2. Includes and Local 928

C.6.2.1. NvFileCommit() 928
C.6.2.2. NvFileSize() 929
C.6.2.3. _plat__NvErrors() 929
C.6.2.4. _plat__NVEnable() 930
C.6.2.5. _plat__NVDisable() 931
C.6.2.6. _plat__IsNvAvailable() 931
C.6.2.7. _plat__NvMemoryRead() 932
C.6.2.8. _plat__NvIsDifferent() 932
C.6.2.9. _plat__NvMemoryWrite() 932
C.6.2.10. _plat__NvMemoryClear() 933
C.6.2.11. _plat__NvMemoryMove() 933
C.6.2.12. _plat__NvCommit() 933
C.6.2.13. _plat__SetNvAvail() 934
C.6.2.14. _plat__ClearNvAvail() 934

 Copyright © TCG 2006-2020

C.6.2.15. _plat__NVNeedsManufacture() 934
C.7 PowerPlat.c 935
C.7.1. Includes and Function Prototypes 935
C.7.2. Functions 935
C.7.2.1. _plat__Signal_PowerOn() 935
C.7.2.2. _plat__WasPowerLost() 935
C.7.2.3. _plat_Signal_Reset() 935
C.7.2.4. _plat__Signal_PowerOff() 936
C.8 PlatformData.h 937
C.9 PlatformData.c 939
C.9.1. Description 939
C.9.2. Includes 939
C.10 PPPlat.c 940
C.10.1. Description 940
C.10.2. Includes 940
C.10.3. Functions 940
C.10.3.1. _plat__PhysicalPresenceAsserted() 940
C.10.3.2. _plat__Signal_PhysicalPresenceOn() 940
C.10.3.3. _plat__Signal_PhysicalPresenceOff() 940
C.11 RunCommand.c 941
C.11.1. Introduction 941
C.11.2. Includes and locals 941
C.11.2.1. _plat__Fail() 941
C.12 Unique.c 942
C.12.1. Introduction 942
C.12.2. Includes 942
C.13 DebugHelpers.c 943
C.13.1. Description 943
C.13.2. Includes and Local 943
C.13.2.1. DebugFileOpen() 943
C.13.2.2. DebugFileClose() 944
C.13.2.3. DebugDumpBuffer() 944
C.14 Platform.h 945
C.15 PlatformACT.h 946
C.16 PlatformACT.c 949
C.16.1. Includes 949
C.16.2. Functions 949
C.16.2.1. ActSignal() 949
C.16.2.2. ActGetDataPointer() 950
C.16.2.3. _plat__ACT_GetImplemented() 950
C.16.2.4. _plat__ACT_GetRemaining() 951
C.16.2.5. _plat__ACT_GetSignaled() 951
C.16.2.6. _plat__ACT_SetSignaled() 951
C.16.2.7. _plat__ACT_GetPending() 951
C.16.2.8. _plat__ACT_UpdateCounter() 952
C.16.2.9. _plat__ACT_EnableTicks() 952
C.16.2.10. ActDecrement() 952
C.16.2.11. _plat__ACT_Tick() 953

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

C.16.2.12. ActZero() 953
C.16.2.13. _plat__ACT_Initialize() 954
C.17 PlatformClock.h 955
Annex D (informative) Remote Procedure Interface 956
D.1 Introduction 956
D.2 TpmTcpProtocol.h 957
D.2.1. Introduction 957
D.2.2. Typedefs and Defines 957
D.2.3. TPM Commands 957
D.2.4. Enumerations and Structures 957
D.3 TcpServer.c 959
D.3.1. Description 959
D.3.2. Includes, Locals, Defines and Function Prototypes 959
D.3.2.1. PlatformServer() 960
D.3.2.2. PlatformSvcRoutine() 962
D.3.2.3. PlatformSignalService() 963
D.3.2.4. RegularCommandService() 963
D.3.2.5. SimulatorTimeServiceRoutine() 964
D.3.2.6. ActTimeService() 965
D.3.2.7. StartTcpServer() 966
D.3.2.8. ReadBytes() 966
D.3.2.9. WriteBytes() 967
D.3.2.10. WriteUINT32() 967
D.3.2.11. ReadUINT32() 968
D.3.2.12. ReadVarBytes() 968
D.3.2.13. WriteVarBytes() 968
D.3.2.14. TpmServer() 969
D.4 TPMCmdp.c 971
D.4.1. Description 971
D.4.2. Includes and Data Definitions 971
D.4.2.1. Signal_Restart() 972
D.4.2.2. Signal_PowerOff() 972
D.4.2.3. _rpc__ForceFailureMode() 972
D.4.2.4. _rpc__Signal_PhysicalPresenceOn() 972
D.4.2.5. _rpc__Signal_PhysicalPresenceOff() 973
D.4.2.6. _rpc__Signal_Hash_Start() 973
D.4.2.7. _rpc__Signal_Hash_Data() 973
D.4.2.8. _rpc__Signal_HashEnd() 973
D.4.2.9. _rpc__Send_Command() 974
D.4.2.10. _rpc__Signal_CancelOn() 974
D.4.2.11. _rpc__Signal_CancelOff() 974
D.4.2.12. _rpc__Signal_NvOn() 975
D.4.2.13. _rpc__Signal_NvOff() 975
D.4.2.14. _rpc__RsaKeyCacheControl() 975
D.4.2.15. _rpc__ACT_GetSignaled() 976
D.5 TPMCmds.c 977
D.5.1. Description 977
D.5.2. Includes, Defines, Data Definitions, and Function Prototypes 977
D.5.2.1. Usage() 978
D.5.2.2. CmdLineParser_Init() 978
D.5.2.3. CmdLineParser_More() 978
D.5.2.4. CmdLineParser_IsOpt() 979

D.5.2.5. CmdLineParser_IsOptPresent() 979
D.5.2.6. CmdLineParser_IsOptPresent() 979
D.5.2.7. main() 980

Trusted Platform Module Library
Part 4: Supporting Routines

1 Scope

This part contains C code that describes the algorithms and methods used by the command code in TPM 2.0 Part 3. The code in this document augments TPM 2.0 Part 2 and TPM 2.0 Part 3 to provide a complete description of a TPM, including the supporting framework for the code that performs the command actions.

Any TPM 2.0 Part 4 code may be replaced by code that provides similar results when interfacing to the action code in TPM 2.0 Part 3. The behavior of code in this document that is not included in an annex is *normative*, as observed at the interfaces with TPM 2.0 Part 3 code. Code in an annex is provided for completeness, that is, to allow a full implementation of the specification from the provided code.

The code in parts 3 and 4 is written to define the behavior of a compliant TPM. In some cases (e.g., firmware update), it is not possible to provide a compliant implementation. In those cases, any implementation provided by the vendor that meets the general description of the function provided in TPM 2.0 Part 3 would be compliant.

The code in parts 3 and 4 is not written to meet any particular level of conformance nor does this specification require that a TPM meet any particular level of conformance.

2 Terms and definitions

For the purposes of this document, the terms and definitions given in TPM 2.0 Part 1 apply.

3 Symbols and abbreviated terms

For the purposes of this document, the symbols and abbreviated terms given in TPM 2.0 Part 1 apply.

4 Automation

TPM 2.0 Part 2 and 3 are constructed so that they can be processed by an automated parser. For example, TPM 2.0 Part 2 can be processed to generate header file contents such as structures, typedefs, and enums. TPM 2.0 Part 3 can be processed to generate command and response marshaling and unmarshaling code.

The automated processor is not provided by the TCG. It was used to generate the Microsoft Visual Studio TPM simulator files. These files are not specification reference code, but rather design examples.

The automation produces TPM_Types.h, a header representing TPM 2.0 Part 2. It also produces, for each major clause of Part 4, a header of the form _fp.h with the function prototypes.

EXAMPLE The header file for SessionProcess.c is SessionProcess_fp.h.

4.1 Configuration Parser

The TPM configuration is largely defined by TpmProfiles.h. This file may be edited in order to change the algorithms and commands supported by a TPM implementation.

A parser exists to process a Word document that defines the TPM configuration. This parser is used to create TpmProfiles.h.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

4.2 Structure Parser

4.2.1 Introduction

The program that processes the tables in TPM 2.0 Part 2 is called "The TPM 2.0 Part 2 Structure Parser."

NOTE A Perl script was used to parse the tables in TPM 2.0 Part 2 to produce the header files and unmarshaling code in for the reference implementation.

The TPM 2.0 Part 2 Structure Parser takes as input the files produced by the TPM 2.0 Part 2 Configuration Parser and the same TPM 2.0 Part 2 specification that was used as input to the TPM 2.0 Part 2 Configuration Parser. The TPM 2.0 Part 2 Structure Parser will generate all of the C structure constant definitions that are required by the TPM interface. Additionally, the parser will generate unmarshaling code for all structures passed to the TPM, and marshaling code for structures passed from the TPM.

The unmarshaling code produced by the parser uses the prototypes defined below. The unmarshaling code will perform validations of the data to ensure that it is compliant with the limitations on the data imposed by the structure definition and use the response code provided in the table if not.

EXAMPLE: The definition for a TPMI_RH_PROVISION indicates that the primitive data type is a TPM_HANDLE and the only allowed values are TPM_RH_OWNER and TPM_RH_PLATFORM. The definition also indicates that the TPM shall indicate TPM_RC_HANDLE if the input value is not none of these values. The unmarshaling code will validate that the input value has one of those allowed values and return TPM_RC_HANDLE if not.

The sections below describe the function prototypes for the marshaling and unmarshaling code that is automatically generated by the TPM 2.0 Part 2 Structure Parser. These prototypes are described here as the unmarshaling and marshaling of various types occurs in places other than when the command is being parsed or the response is being built. The prototypes and the description of the interface are intended to aid in the comprehension of the code that uses these auto-generated routines.

4.2.2 Unmarshaling Code Prototype

4.2.2.1 Simple Types and Structures

The general form for the unmarshaling code for a simple type or a structure is:

```
TPM_RC TYPE_Unmarshal(TYPE *target, BYTE **buffer, INT32 *size);
```

Where:

TYPE	name of the data type or structure
***target**	location in the TPM memory into which the data from ****buffer** is placed
****buffer**	location in input buffer containing the most significant octet (MSO) of ***target**
***size**	number of octets remaining in ****buffer**

When the data is successfully unmarshaled, the called routine will return TPM_RC_SUCCESS. Otherwise, it will return a Format-One response code (see TPM 2.0 Part 2).

If the data is successfully unmarshaled, ***buffer** is advanced point to the first octet of the next parameter in the input buffer and **size** is reduced by the number of octets removed from the buffer.

When the data type is a simple type, the parser will generate code that will unmarshal the underlying type and then perform checks on the type as indicated by the type definition.

When the data type is a structure, the parser will generate code that unmarshals each of the structure elements in turn and performs any additional parameter checks as indicated by the data type.

 Copyright © TCG 2006-2020

4.2.2.2 Union Types

When a union is defined, an extra parameter is defined for the unmarshaling code. This parameter is the selector for the type. The unmarshaling code for the union will unmarshal the type indicated by the selector.

The function prototype for a union has the form:

```
TPM_RC TYPE_Unmarshal(TYPE *target, BYTE **buffer, INT32 *size, UINT32 selector);
```

where:

TYPE	name of the union type or structure
***target**	location in the TPM memory into which the data from ****buffer** is placed
****buffer**	location in input buffer containing the most significant octet (MSO) of ***target**
***size**	number of octets remaining in ****buffer**
selector	union selector that determines what will be unmarshaled into ***target**

4.2.2.3 Null Types

In some cases, the structure definition allows an optional "null" value. The "null" value allows the use of the same C type for the entity even though it does not always have the same members.

For example, the TPMI_ALG_HASH data type is used in many places. In some cases, TPM_ALG_NULL is permitted and in some cases it is not. If two different data types had to be defined, the interfaces and code would become more complex because of the number of cast operations that would be necessary. Rather than encumber the code, the "null" value is defined and the unmarshaling code is given a flag to indicate if this instance of the type accepts the "null" parameter or not. When the data type has a "null" value, the function prototype is

```
TPM_RC TYPE_Unmarshal(TYPE *target, BYTE **buffer, INT32 *size, BOOL flag);
```

The parser detects when the type allows a "null" value and will always include **flag** in any call to unmarshal that type. **flag** TRUE indicates that null is accepted.

4.2.2.4 Arrays

Any data type may be included in an array. The function prototype use to unmarshal an array for a **TYPE** is

```
TPM_RC TYPE_Array_Unmarshal(TYPE *target, BYTE **buffer, INT32 *size,INT32 count);
```

The generated code for an array uses a **count**-limited loop within which it calls the unmarshaling code for **TYPE**.

4.2.3 Marshaling Code Function Prototypes

4.2.3.1 Simple Types and Structures

The general form for the marshaling code for a simple type or a structure is:

```
UINT16 TYPE_Marshal(TYPE *source, BYTE **buffer, INT32 *size);
```

Where:

 Copyright © TCG 2006-2020

TYPE	name of the data type or structure
***source**	location in the TPM memory containing the value that is to be marshaled in to the designated buffer
****buffer**	location in the output buffer where the first octet of the **TYPE** is to be placed
***size**	number of octets remaining in ****buffer**.

If **buffer** is a NULL pointer, then no data is marshaled, but the routine will compute and return the size of the memory required to marshal the indicated type. ***size** is not changed.

If **buffer** is not a NULL pointer, data is marshaled, ***buffer** is advanced to point to the first octet of the next location in the output buffer, and the called routine will return the number of octets marshaled into ****buffer**. This occurs even if **size** is a NULL pointer. If **size** is a not NULL pointer ***size** is reduced by the number of octets placed in the buffer.

When the data type is a simple type, the parser will generate code that will marshal the underlying type. The presumption is that the TPM internal structures are consistent and correct so the marshaling code does not validate that the data placed in the buffer has a permissible value. The presumption is also that the **size** is sufficient for the source being marshaled.

When the data type is a structure, the parser will generate code that marshals each of the structure elements in turn.

4.2.3.2 Union Types

An extra parameter is defined for the marshaling function of a union. This parameter is the selector for the type. The marshaling code for the union will marshal the type indicated by the selector.

The function prototype for a union has the form:

```
UINT16 TYPE_Marshal(TYPE *source, BYTE **buffer, INT32 *size, UINT32 selector);
```

The parameters have a similar meaning as those in 4.2.2.2 but the data movement is from **source** to **buffer**.

4.2.3.3 Arrays

Any type may be included in an array. The function prototype use to unmarshal an array is:

```
UINT16 TYPE_Array_Marshal(TYPE *source, BYTE **buffer, INT32 *size, INT32 count);
```

4.2.3.4 The generated code for an array uses a count-limited loop within which it calls the marshaling code for TYPE.Table-driven Marshaling

The most recent versions of the TPM code includes the option to use table-driven marshaling rather that the procedural marshaling described in previous clauses in 4.2.2. The structure and processing of this code is complex and is provided in the code.

4.3 Part 3 Parsing

The Command / Response tables in Part 3 of this specification are processed by scripts to produce the command-specific data structures used by functions in this TPM 2.0 Part 4. They are:

- **CommandAttributeData.h** -- This file contains the command attributes reported by TPM2_GetCapability.

 Copyright © TCG 2006-2020

- **CommandAttributes.h** – This file contains the definition of command attributes that are extracted by the parsing code. The file mainly exists to ensure that the parsing code and the function code are using the same attributes.
- **CommandDispatchData.h** – This file contains the data definitions for the table driven version of the command dispatcher.

Part 3 parsing also produces special function prototype files as described in 4.4.

4.4 Function Prototypes

For functions that have entry definitions not defined by Part 3 tables. a script is used to extracts function prototypes from the code. For each .c file that is not in Part 3, a file with the same name is created with a suffix of _fp.h. For example, the function prototypes for Create.c will be placed in a file called Create_fp.h. The _fp.h is added because some files have two types of associated headers: the one containing the function prototypes for the file and another containing definitions that are specific to that file.

In some cases, a function will be replaced by a macro. The macro is defined in the .c file and extracted by the function prototype processor. A special comment tag ("//%") is used to indicate that the line is to be included in the function prototype file. If the "//%" tag occurs at the start of the line, it is deleted. If it occurs later in the line, it is preserved. Removing the "//%/ at the start of the line allows the macro to be placed in the .c file with the tag as a prefix, and then show up in the _fp.h file as the actual macro. This allows the code that includes that function prototype code to use the appropriate macro.

For files that that contain the command actions, a special _fp.h file is created from the tables in Part 3. These files contain:

- the definition of the input and output structure of the function;
- definition of command-specific return code modifiers (parameter identifiers); and
- the function prototype for the command action function.

Create_fp.h (shown below) is prototypical of the command _fp.h files.

```
1   #if CC_Create // Command must be enabled
2   #ifndef _Create_FP_H_
3   #define _Create_FP_H_
```

Input structure definition

```
4   typedef struct {
5       TPMI_DH_OBJECT              parentHandle;
6       TPM2B_SENSITIVE_CREATE      inSensitive;
7       TPM2B_PUBLIC                inPublic;
8       TPM2B_DATA                  outsideInfo;
9       TPML_PCR_SELECTION          creationPCR;
10  } Create_In;
```

Output structure definition

```
11  typedef struct {
12      TPM2B_PRIVATE               outPrivate;
13      TPM2B_PUBLIC                outPublic;
14      TPM2B_CREATION_DATA         creationData;
15      TPM2B_DIGEST                creationHash;
16      TPMT_TK_CREATION            creationTicket;
17  } Create_Out;
```

Response code modifiers

```
18  #define RC_Create_parentHandle   (TPM_RC_H + TPM_RC_1)
```

```
19  #define RC_Create_inSensitive    (TPM_RC_P + TPM_RC_1)
20  #define RC_Create_inPublic       (TPM_RC_P + TPM_RC_2)
21  #define RC_Create_outsideInfo    (TPM_RC_P + TPM_RC_3)
22  #define RC_Create_creationPCR    (TPM_RC_P + TPM_RC_4)
```

Function prototype

```
23  TPM_RC
24  TPM2_Create(
25      Create_In                   *in,
26      Create_Out                  *out
27  );
28  #endif  // _Create_FP_H_
29  #endif  // CC_Create
```

4.5 Portability

Where reasonable, the code is written to be portable. There are a few known cases where the code is not portable. Specifically, the handling of bit fields will not always be portable. The bit fields are marshaled and unmarshaled as a simple element of the underlying type. For example, a TPMA_SESSION is defined as a bit field in an octet (BYTE). When sent on the interface a TPMA_SESSION will occupy one octet. When unmarshaled, it is unmarshaled as a UINT8. The ramifications of this are that a TPMA_SESSION will occupy the 0th octet of the structure in which it is placed regardless of the size of the structure.

Many compilers will pad a bit field to some "natural" size for the processor, often 4 octets, meaning that `sizeof(TPMA_SESSION)` would return 4 rather than 1 (the canonical size of a TPMA_SESSION).

For a little endian machine, padding of bit fields should have little consequence since the 0th octet always contains the 0th bit of the structure no matter how large the structure. However, for a big endian machine, the 0th bit will be in the highest numbered octet. When unmarshaling a TPMA_SESSION, the current unmarshaling code will place the input octet at the 0th octet of the TPMA_SESSION. Since the 0th octet is most significant octet, this has the effect of shifting all the session attribute bits left by 24 places.

As a consequence, someone implementing on a big endian machine should do one of two things:

a) allocate all structures as packed to a byte boundary (this may not be possible if the processor does not handle unaligned accesses); or

b) modify the code that manipulates bit fields that are not defined as being the alignment size of the system.

For many RISC processors, option #2 would be the only choice. This is may not be a terribly daunting task since only two attribute structures are not 32-bits (TPMA_SESSION and TPMA_LOCALITY).

 Copyright © TCG 2006-2020

5 Header Files

5.1 Introduction

The files in this section are used to define values that are used in multiple parts of the specification and are not confined to a single module.

5.2 BaseTypes.h

```
1   #ifndef _BASE_TYPES_H_
2   #define _BASE_TYPES_H_
```

NULL definition

```
3   #ifndef NULL
4   #define NULL                (0)
5   #endif
6   typedef uint8_t             UINT8;
7   typedef uint8_t             BYTE;
8   typedef int8_t              INT8;
9   typedef int                 BOOL;
10  typedef uint16_t            UINT16;
11  typedef int16_t             INT16;
12  typedef uint32_t            UINT32;
13  typedef int32_t             INT32;
14  typedef uint64_t            UINT64;
15  typedef int64_t             INT64;
16  #endif // _BASE_TYPES_H_
```

5.3 Capabilities.h

This file contains defines for the number of capability values that will fit into the largest data buffer.

These defines are used in various function in the "support" and the "subsystem" code groups. A module that supports a type that is returned by a capability will have a function that returns the capabilities of the type.

EXAMPLE PCR.c contains PCRCapGetHandles() and PCRCapGetProperties().

```
1  #ifndef     _CAPABILITIES_H
2  #define     _CAPABILITIES_H
3  #define     MAX_CAP_DATA          (MAX_CAP_BUFFER - sizeof(TPM_CAP)-sizeof(UINT32))
4  #define     MAX_CAP_ALGS          (MAX_CAP_DATA / sizeof(TPMS_ALG_PROPERTY))
5  #define     MAX_CAP_HANDLES       (MAX_CAP_DATA / sizeof(TPM_HANDLE))
6  #define     MAX_CAP_CC            (MAX_CAP_DATA / sizeof(TPM_CC))
7  #define     MAX_TPM_PROPERTIES    (MAX_CAP_DATA / sizeof(TPMS_TAGGED_PROPERTY))
8  #define     MAX_PCR_PROPERTIES    (MAX_CAP_DATA / sizeof(TPMS_TAGGED_PCR_SELECT))
9  #define     MAX_ECC_CURVES        (MAX_CAP_DATA / sizeof(TPM_ECC_CURVE))
10 #define     MAX_TAGGED_POLICIES   (MAX_CAP_DATA / sizeof(TPMS_TAGGED_POLICY))
11 #define     MAX_ACT_DATA          (MAX_CAP_DATA / sizeof(TPMS_ACT_DATA))
12 #define     MAX_AC_CAPABILITIES   (MAX_CAP_DATA / sizeof(TPMS_AC_OUTPUT))
13 #endif
```

 Copyright © TCG 2006-2020

5.4 CommandAttributeData.h

This file should only be included by CommandCodeAttibutes.c

```
1  #ifdef _COMMAND_CODE_ATTRIBUTES_
2  #include "CommandAttributes.h"
3  #if COMPRESSED_LISTS
4  #   define      PAD_LIST    0
5  #else
6  #   define      PAD_LIST    1
7  #endif
```

This is the command code attribute array for GetCapability(). Both this array and *s_commandAttributes* provides command code attributes, but tuned for different purpose

```
8  const TPMA_CC    s_ccAttr [] = {
9  #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
10         TPMA_CC_INITIALIZER(0x011F, 0, 1, 0, 0, 2, 0, 0, 0),
11 #endif
12 #if (PAD_LIST || CC_EvictControl)
13         TPMA_CC_INITIALIZER(0x0120, 0, 1, 0, 0, 2, 0, 0, 0),
14 #endif
15 #if (PAD_LIST || CC_HierarchyControl)
16         TPMA_CC_INITIALIZER(0x0121, 0, 1, 1, 0, 1, 0, 0, 0),
17 #endif
18 #if (PAD_LIST || CC_NV_UndefineSpace)
19         TPMA_CC_INITIALIZER(0x0122, 0, 1, 0, 0, 2, 0, 0, 0),
20 #endif
21 #if (PAD_LIST )
22         TPMA_CC_INITIALIZER(0x0123, 0, 0, 0, 0, 0, 0, 0, 0),
23 #endif
24 #if (PAD_LIST || CC_ChangeEPS)
25         TPMA_CC_INITIALIZER(0x0124, 0, 1, 1, 0, 1, 0, 0, 0),
26 #endif
27 #if (PAD_LIST || CC_ChangePPS)
28         TPMA_CC_INITIALIZER(0x0125, 0, 1, 1, 0, 1, 0, 0, 0),
29 #endif
30 #if (PAD_LIST || CC_Clear)
31         TPMA_CC_INITIALIZER(0x0126, 0, 1, 1, 0, 1, 0, 0, 0),
32 #endif
33 #if (PAD_LIST || CC_ClearControl)
34         TPMA_CC_INITIALIZER(0x0127, 0, 1, 0, 0, 1, 0, 0, 0),
35 #endif
36 #if (PAD_LIST || CC_ClockSet)
37         TPMA_CC_INITIALIZER(0x0128, 0, 1, 0, 0, 1, 0, 0, 0),
38 #endif
39 #if (PAD_LIST || CC_HierarchyChangeAuth)
40         TPMA_CC_INITIALIZER(0x0129, 0, 1, 0, 0, 1, 0, 0, 0),
41 #endif
42 #if (PAD_LIST || CC_NV_DefineSpace)
43         TPMA_CC_INITIALIZER(0x012A, 0, 1, 0, 0, 1, 0, 0, 0),
44 #endif
45 #if (PAD_LIST || CC_PCR_Allocate)
46         TPMA_CC_INITIALIZER(0x012B, 0, 1, 0, 0, 1, 0, 0, 0),
47 #endif
48 #if (PAD_LIST || CC_PCR_SetAuthPolicy)
49         TPMA_CC_INITIALIZER(0x012C, 0, 1, 0, 0, 1, 0, 0, 0),
50 #endif
51 #if (PAD_LIST || CC_PP_Commands)
52         TPMA_CC_INITIALIZER(0x012D, 0, 1, 0, 0, 1, 0, 0, 0),
53 #endif
54 #if (PAD_LIST || CC_SetPrimaryPolicy)
55         TPMA_CC_INITIALIZER(0x012E, 0, 1, 0, 0, 1, 0, 0, 0),
56 #endif
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
57  #if (PAD_LIST || CC_FieldUpgradeStart)
58          TPMA_CC_INITIALIZER(0x012F, 0, 0, 0, 0, 2, 0, 0, 0),
59  #endif
60  #if (PAD_LIST || CC_ClockRateAdjust)
61          TPMA_CC_INITIALIZER(0x0130, 0, 0, 0, 0, 1, 0, 0, 0),
62  #endif
63  #if (PAD_LIST || CC_CreatePrimary)
64          TPMA_CC_INITIALIZER(0x0131, 0, 0, 0, 0, 1, 1, 0, 0),
65  #endif
66  #if (PAD_LIST || CC_NV_GlobalWriteLock)
67          TPMA_CC_INITIALIZER(0x0132, 0, 1, 0, 0, 1, 0, 0, 0),
68  #endif
69  #if (PAD_LIST || CC_GetCommandAuditDigest)
70          TPMA_CC_INITIALIZER(0x0133, 0, 1, 0, 0, 2, 0, 0, 0),
71  #endif
72  #if (PAD_LIST || CC_NV_Increment)
73          TPMA_CC_INITIALIZER(0x0134, 0, 1, 0, 0, 2, 0, 0, 0),
74  #endif
75  #if (PAD_LIST || CC_NV_SetBits)
76          TPMA_CC_INITIALIZER(0x0135, 0, 1, 0, 0, 2, 0, 0, 0),
77  #endif
78  #if (PAD_LIST || CC_NV_Extend)
79          TPMA_CC_INITIALIZER(0x0136, 0, 1, 0, 0, 2, 0, 0, 0),
80  #endif
81  #if (PAD_LIST || CC_NV_Write)
82          TPMA_CC_INITIALIZER(0x0137, 0, 1, 0, 0, 2, 0, 0, 0),
83  #endif
84  #if (PAD_LIST || CC_NV_WriteLock)
85          TPMA_CC_INITIALIZER(0x0138, 0, 1, 0, 0, 2, 0, 0, 0),
86  #endif
87  #if (PAD_LIST || CC_DictionaryAttackLockReset)
88          TPMA_CC_INITIALIZER(0x0139, 0, 1, 0, 0, 1, 0, 0, 0),
89  #endif
90  #if (PAD_LIST || CC_DictionaryAttackParameters)
91          TPMA_CC_INITIALIZER(0x013A, 0, 1, 0, 0, 1, 0, 0, 0),
92  #endif
93  #if (PAD_LIST || CC_NV_ChangeAuth)
94          TPMA_CC_INITIALIZER(0x013B, 0, 1, 0, 0, 1, 0, 0, 0),
95  #endif
96  #if (PAD_LIST || CC_PCR_Event)
97          TPMA_CC_INITIALIZER(0x013C, 0, 1, 0, 0, 1, 0, 0, 0),
98  #endif
99  #if (PAD_LIST || CC_PCR_Reset)
100         TPMA_CC_INITIALIZER(0x013D, 0, 1, 0, 0, 1, 0, 0, 0),
101 #endif
102 #if (PAD_LIST || CC_SequenceComplete)
103         TPMA_CC_INITIALIZER(0x013E, 0, 0, 0, 1, 1, 0, 0, 0),
104 #endif
105 #if (PAD_LIST || CC_SetAlgorithmSet)
106         TPMA_CC_INITIALIZER(0x013F, 0, 1, 0, 0, 1, 0, 0, 0),
107 #endif
108 #if (PAD_LIST || CC_SetCommandCodeAuditStatus)
109         TPMA_CC_INITIALIZER(0x0140, 0, 1, 0, 0, 1, 0, 0, 0),
110 #endif
111 #if (PAD_LIST || CC_FieldUpgradeData)
112         TPMA_CC_INITIALIZER(0x0141, 0, 1, 0, 0, 0, 0, 0, 0),
113 #endif
114 #if (PAD_LIST || CC_IncrementalSelfTest)
115         TPMA_CC_INITIALIZER(0x0142, 0, 1, 0, 0, 0, 0, 0, 0),
116 #endif
117 #if (PAD_LIST || CC_SelfTest)
118         TPMA_CC_INITIALIZER(0x0143, 0, 1, 0, 0, 0, 0, 0, 0),
119 #endif
120 #if (PAD_LIST || CC_Startup)
121         TPMA_CC_INITIALIZER(0x0144, 0, 1, 0, 0, 0, 0, 0, 0),
122 #endif
```

 Copyright © TCG 2006-2020

```
123  #if (PAD_LIST || CC_Shutdown)
124          TPMA_CC_INITIALIZER(0x0145, 0, 1, 0, 0, 0, 0, 0, 0),
125  #endif
126  #if (PAD_LIST || CC_StirRandom)
127          TPMA_CC_INITIALIZER(0x0146, 0, 1, 0, 0, 0, 0, 0, 0),
128  #endif
129  #if (PAD_LIST || CC_ActivateCredential)
130          TPMA_CC_INITIALIZER(0x0147, 0, 0, 0, 0, 2, 0, 0, 0),
131  #endif
132  #if (PAD_LIST || CC_Certify)
133          TPMA_CC_INITIALIZER(0x0148, 0, 0, 0, 0, 2, 0, 0, 0),
134  #endif
135  #if (PAD_LIST || CC_PolicyNV)
136          TPMA_CC_INITIALIZER(0x0149, 0, 0, 0, 0, 3, 0, 0, 0),
137  #endif
138  #if (PAD_LIST || CC_CertifyCreation)
139          TPMA_CC_INITIALIZER(0x014A, 0, 0, 0, 0, 2, 0, 0, 0),
140  #endif
141  #if (PAD_LIST || CC_Duplicate)
142          TPMA_CC_INITIALIZER(0x014B, 0, 0, 0, 0, 2, 0, 0, 0),
143  #endif
144  #if (PAD_LIST || CC_GetTime)
145          TPMA_CC_INITIALIZER(0x014C, 0, 0, 0, 0, 2, 0, 0, 0),
146  #endif
147  #if (PAD_LIST || CC_GetSessionAuditDigest)
148          TPMA_CC_INITIALIZER(0x014D, 0, 0, 0, 0, 3, 0, 0, 0),
149  #endif
150  #if (PAD_LIST || CC_NV_Read)
151          TPMA_CC_INITIALIZER(0x014E, 0, 0, 0, 0, 2, 0, 0, 0),
152  #endif
153  #if (PAD_LIST || CC_NV_ReadLock)
154          TPMA_CC_INITIALIZER(0x014F, 0, 1, 0, 0, 2, 0, 0, 0),
155  #endif
156  #if (PAD_LIST || CC_ObjectChangeAuth)
157          TPMA_CC_INITIALIZER(0x0150, 0, 0, 0, 0, 2, 0, 0, 0),
158  #endif
159  #if (PAD_LIST || CC_PolicySecret)
160          TPMA_CC_INITIALIZER(0x0151, 0, 0, 0, 0, 2, 0, 0, 0),
161  #endif
162  #if (PAD_LIST || CC_Rewrap)
163          TPMA_CC_INITIALIZER(0x0152, 0, 0, 0, 0, 2, 0, 0, 0),
164  #endif
165  #if (PAD_LIST || CC_Create)
166          TPMA_CC_INITIALIZER(0x0153, 0, 0, 0, 0, 1, 0, 0, 0),
167  #endif
168  #if (PAD_LIST || CC_ECDH_ZGen)
169          TPMA_CC_INITIALIZER(0x0154, 0, 0, 0, 0, 1, 0, 0, 0),
170  #endif
171  #if (PAD_LIST || (CC_HMAC || CC_MAC))
172          TPMA_CC_INITIALIZER(0x0155, 0, 0, 0, 0, 1, 0, 0, 0),
173  #endif
174  #if (PAD_LIST || CC_Import)
175          TPMA_CC_INITIALIZER(0x0156, 0, 0, 0, 0, 1, 0, 0, 0),
176  #endif
177  #if (PAD_LIST || CC_Load)
178          TPMA_CC_INITIALIZER(0x0157, 0, 0, 0, 0, 1, 1, 0, 0),
179  #endif
180  #if (PAD_LIST || CC_Quote)
181          TPMA_CC_INITIALIZER(0x0158, 0, 0, 0, 0, 1, 0, 0, 0),
182  #endif
183  #if (PAD_LIST || CC_RSA_Decrypt)
184          TPMA_CC_INITIALIZER(0x0159, 0, 0, 0, 0, 1, 0, 0, 0),
185  #endif
186  #if (PAD_LIST )
187          TPMA_CC_INITIALIZER(0x015A, 0, 0, 0, 0, 0, 0, 0, 0),
188  #endif
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
189 #if (PAD_LIST || (CC_HMAC_Start || CC_MAC_Start))
190         TPMA_CC_INITIALIZER(0x015B, 0, 0, 0, 0, 1, 1, 0, 0),
191 #endif
192 #if (PAD_LIST || CC_SequenceUpdate)
193         TPMA_CC_INITIALIZER(0x015C, 0, 0, 0, 0, 1, 0, 0, 0),
194 #endif
195 #if (PAD_LIST || CC_Sign)
196         TPMA_CC_INITIALIZER(0x015D, 0, 0, 0, 0, 1, 0, 0, 0),
197 #endif
198 #if (PAD_LIST || CC_Unseal)
199         TPMA_CC_INITIALIZER(0x015E, 0, 0, 0, 0, 1, 0, 0, 0),
200 #endif
201 #if (PAD_LIST )
202         TPMA_CC_INITIALIZER(0x015F, 0, 0, 0, 0, 0, 0, 0, 0),
203 #endif
204 #if (PAD_LIST || CC_PolicySigned)
205         TPMA_CC_INITIALIZER(0x0160, 0, 0, 0, 0, 2, 0, 0, 0),
206 #endif
207 #if (PAD_LIST || CC_ContextLoad)
208         TPMA_CC_INITIALIZER(0x0161, 0, 0, 0, 0, 0, 1, 0, 0),
209 #endif
210 #if (PAD_LIST || CC_ContextSave)
211         TPMA_CC_INITIALIZER(0x0162, 0, 0, 0, 0, 1, 0, 0, 0),
212 #endif
213 #if (PAD_LIST || CC_ECDH_KeyGen)
214         TPMA_CC_INITIALIZER(0x0163, 0, 0, 0, 0, 1, 0, 0, 0),
215 #endif
216 #if (PAD_LIST || CC_EncryptDecrypt)
217         TPMA_CC_INITIALIZER(0x0164, 0, 0, 0, 0, 1, 0, 0, 0),
218 #endif
219 #if (PAD_LIST || CC_FlushContext)
220         TPMA_CC_INITIALIZER(0x0165, 0, 0, 0, 0, 0, 0, 0, 0),
221 #endif
222 #if (PAD_LIST )
223         TPMA_CC_INITIALIZER(0x0166, 0, 0, 0, 0, 0, 0, 0, 0),
224 #endif
225 #if (PAD_LIST || CC_LoadExternal)
226         TPMA_CC_INITIALIZER(0x0167, 0, 0, 0, 0, 0, 1, 0, 0),
227 #endif
228 #if (PAD_LIST || CC_MakeCredential)
229         TPMA_CC_INITIALIZER(0x0168, 0, 0, 0, 0, 1, 0, 0, 0),
230 #endif
231 #if (PAD_LIST || CC_NV_ReadPublic)
232         TPMA_CC_INITIALIZER(0x0169, 0, 0, 0, 0, 1, 0, 0, 0),
233 #endif
234 #if (PAD_LIST || CC_PolicyAuthorize)
235         TPMA_CC_INITIALIZER(0x016A, 0, 0, 0, 0, 1, 0, 0, 0),
236 #endif
237 #if (PAD_LIST || CC_PolicyAuthValue)
238         TPMA_CC_INITIALIZER(0x016B, 0, 0, 0, 0, 1, 0, 0, 0),
239 #endif
240 #if (PAD_LIST || CC_PolicyCommandCode)
241         TPMA_CC_INITIALIZER(0x016C, 0, 0, 0, 0, 1, 0, 0, 0),
242 #endif
243 #if (PAD_LIST || CC_PolicyCounterTimer)
244         TPMA_CC_INITIALIZER(0x016D, 0, 0, 0, 0, 1, 0, 0, 0),
245 #endif
246 #if (PAD_LIST || CC_PolicyCpHash)
247         TPMA_CC_INITIALIZER(0x016E, 0, 0, 0, 0, 1, 0, 0, 0),
248 #endif
249 #if (PAD_LIST || CC_PolicyLocality)
250         TPMA_CC_INITIALIZER(0x016F, 0, 0, 0, 0, 1, 0, 0, 0),
251 #endif
252 #if (PAD_LIST || CC_PolicyNameHash)
253         TPMA_CC_INITIALIZER(0x0170, 0, 0, 0, 0, 1, 0, 0, 0),
254 #endif
```

```
255 #if (PAD_LIST || CC_PolicyOR)
256         TPMA_CC_INITIALIZER(0x0171, 0, 0, 0, 0, 1, 0, 0, 0),
257 #endif
258 #if (PAD_LIST || CC_PolicyTicket)
259         TPMA_CC_INITIALIZER(0x0172, 0, 0, 0, 0, 1, 0, 0, 0),
260 #endif
261 #if (PAD_LIST || CC_ReadPublic)
262         TPMA_CC_INITIALIZER(0x0173, 0, 0, 0, 0, 1, 0, 0, 0),
263 #endif
264 #if (PAD_LIST || CC_RSA_Encrypt)
265         TPMA_CC_INITIALIZER(0x0174, 0, 0, 0, 0, 1, 0, 0, 0),
266 #endif
267 #if (PAD_LIST )
268         TPMA_CC_INITIALIZER(0x0175, 0, 0, 0, 0, 0, 0, 0, 0),
269 #endif
270 #if (PAD_LIST || CC_StartAuthSession)
271         TPMA_CC_INITIALIZER(0x0176, 0, 0, 0, 0, 2, 1, 0, 0),
272 #endif
273 #if (PAD_LIST || CC_VerifySignature)
274         TPMA_CC_INITIALIZER(0x0177, 0, 0, 0, 0, 1, 0, 0, 0),
275 #endif
276 #if (PAD_LIST || CC_ECC_Parameters)
277         TPMA_CC_INITIALIZER(0x0178, 0, 0, 0, 0, 0, 0, 0, 0),
278 #endif
279 #if (PAD_LIST || CC_FirmwareRead)
280         TPMA_CC_INITIALIZER(0x0179, 0, 0, 0, 0, 0, 0, 0, 0),
281 #endif
282 #if (PAD_LIST || CC_GetCapability)
283         TPMA_CC_INITIALIZER(0x017A, 0, 0, 0, 0, 0, 0, 0, 0),
284 #endif
285 #if (PAD_LIST || CC_GetRandom)
286         TPMA_CC_INITIALIZER(0x017B, 0, 0, 0, 0, 0, 0, 0, 0),
287 #endif
288 #if (PAD_LIST || CC_GetTestResult)
289         TPMA_CC_INITIALIZER(0x017C, 0, 0, 0, 0, 0, 0, 0, 0),
290 #endif
291 #if (PAD_LIST || CC_Hash)
292         TPMA_CC_INITIALIZER(0x017D, 0, 0, 0, 0, 0, 0, 0, 0),
293 #endif
294 #if (PAD_LIST || CC_PCR_Read)
295         TPMA_CC_INITIALIZER(0x017E, 0, 0, 0, 0, 0, 0, 0, 0),
296 #endif
297 #if (PAD_LIST || CC_PolicyPCR)
298         TPMA_CC_INITIALIZER(0x017F, 0, 0, 0, 0, 1, 0, 0, 0),
299 #endif
300 #if (PAD_LIST || CC_PolicyRestart)
301         TPMA_CC_INITIALIZER(0x0180, 0, 0, 0, 0, 1, 0, 0, 0),
302 #endif
303 #if (PAD_LIST || CC_ReadClock)
304         TPMA_CC_INITIALIZER(0x0181, 0, 0, 0, 0, 0, 0, 0, 0),
305 #endif
306 #if (PAD_LIST || CC_PCR_Extend)
307         TPMA_CC_INITIALIZER(0x0182, 0, 1, 0, 0, 1, 0, 0, 0),
308 #endif
309 #if (PAD_LIST || CC_PCR_SetAuthValue)
310         TPMA_CC_INITIALIZER(0x0183, 0, 0, 0, 0, 1, 0, 0, 0),
311 #endif
312 #if (PAD_LIST || CC_NV_Certify)
313         TPMA_CC_INITIALIZER(0x0184, 0, 0, 0, 0, 3, 0, 0, 0),
314 #endif
315 #if (PAD_LIST || CC_EventSequenceComplete)
316         TPMA_CC_INITIALIZER(0x0185, 0, 1, 0, 1, 2, 0, 0, 0),
317 #endif
318 #if (PAD_LIST || CC_HashSequenceStart)
319         TPMA_CC_INITIALIZER(0x0186, 0, 0, 0, 0, 0, 1, 0, 0),
320 #endif
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
321  #if (PAD_LIST || CC_PolicyPhysicalPresence)
322          TPMA_CC_INITIALIZER(0x0187, 0, 0, 0, 0, 1, 0, 0, 0),
323  #endif
324  #if (PAD_LIST || CC_PolicyDuplicationSelect)
325          TPMA_CC_INITIALIZER(0x0188, 0, 0, 0, 0, 1, 0, 0, 0),
326  #endif
327  #if (PAD_LIST || CC_PolicyGetDigest)
328          TPMA_CC_INITIALIZER(0x0189, 0, 0, 0, 0, 1, 0, 0, 0),
329  #endif
330  #if (PAD_LIST || CC_TestParms)
331          TPMA_CC_INITIALIZER(0x018A, 0, 0, 0, 0, 0, 0, 0, 0),
332  #endif
333  #if (PAD_LIST || CC_Commit)
334          TPMA_CC_INITIALIZER(0x018B, 0, 0, 0, 0, 1, 0, 0, 0),
335  #endif
336  #if (PAD_LIST || CC_PolicyPassword)
337          TPMA_CC_INITIALIZER(0x018C, 0, 0, 0, 0, 1, 0, 0, 0),
338  #endif
339  #if (PAD_LIST || CC_ZGen_2Phase)
340          TPMA_CC_INITIALIZER(0x018D, 0, 0, 0, 0, 1, 0, 0, 0),
341  #endif
342  #if (PAD_LIST || CC_EC_Ephemeral)
343          TPMA_CC_INITIALIZER(0x018E, 0, 0, 0, 0, 0, 0, 0, 0),
344  #endif
345  #if (PAD_LIST || CC_PolicyNvWritten)
346          TPMA_CC_INITIALIZER(0x018F, 0, 0, 0, 0, 1, 0, 0, 0),
347  #endif
348  #if (PAD_LIST || CC_PolicyTemplate)
349          TPMA_CC_INITIALIZER(0x0190, 0, 0, 0, 0, 1, 0, 0, 0),
350  #endif
351  #if (PAD_LIST || CC_CreateLoaded)
352          TPMA_CC_INITIALIZER(0x0191, 0, 0, 0, 0, 1, 1, 0, 0),
353  #endif
354  #if (PAD_LIST || CC_PolicyAuthorizeNV)
355          TPMA_CC_INITIALIZER(0x0192, 0, 0, 0, 0, 3, 0, 0, 0),
356  #endif
357  #if (PAD_LIST || CC_EncryptDecrypt2)
358          TPMA_CC_INITIALIZER(0x0193, 0, 0, 0, 0, 1, 0, 0, 0),
359  #endif
360  #if (PAD_LIST || CC_AC_GetCapability)
361          TPMA_CC_INITIALIZER(0x0194, 0, 0, 0, 0, 1, 0, 0, 0),
362  #endif
363  #if (PAD_LIST || CC_AC_Send)
364          TPMA_CC_INITIALIZER(0x0195, 0, 0, 0, 0, 3, 0, 0, 0),
365  #endif
366  #if (PAD_LIST || CC_Policy_AC_SendSelect)
367          TPMA_CC_INITIALIZER(0x0196, 0, 0, 0, 0, 1, 0, 0, 0),
368  #endif
369  #if (PAD_LIST || CC_CertifyX509)
370          TPMA_CC_INITIALIZER(0x0197, 0, 0, 0, 0, 2, 0, 0, 0),
371  #endif
372  #if (PAD_LIST || CC_ACT_SetTimeout)
373          TPMA_CC_INITIALIZER(0x0198, 0, 0, 0, 0, 1, 0, 0, 0),
374  #endif
375  #if (PAD_LIST || CC_Vendor_TCG_Test)
376          TPMA_CC_INITIALIZER(0x0000, 0, 0, 0, 0, 0, 0, 1, 0),
377  #endif
378          TPMA_ZERO_INITIALIZER()
379  };
```

This is the command code attribute structure.

```
380  const COMMAND_ATTRIBUTES    s_commandAttributes [] = {
381  #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
382          (COMMAND_ATTRIBUTES)(CC_NV_UndefineSpaceSpecial     *  // 0x011F
```

 Copyright © TCG 2006-2020

```
383                 (IS_IMPLEMENTED+HANDLE_1_ADMIN+HANDLE_2_USER+PP_COMMAND)),
384     #endif
385     #if (PAD_LIST || CC_EvictControl)
386             (COMMAND_ATTRIBUTES)(CC_EvictControl               *  // 0x0120
387                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
388     #endif
389     #if (PAD_LIST || CC_HierarchyControl)
390             (COMMAND_ATTRIBUTES)(CC_HierarchyControl           *  // 0x0121
391                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
392     #endif
393     #if (PAD_LIST || CC_NV_UndefineSpace)
394             (COMMAND_ATTRIBUTES)(CC_NV_UndefineSpace           *  // 0x0122
395                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
396     #endif
397     #if (PAD_LIST )
398             (COMMAND_ATTRIBUTES)(0),                              // 0x0123
399     #endif
400     #if (PAD_LIST || CC_ChangeEPS)
401             (COMMAND_ATTRIBUTES)(CC_ChangeEPS                  *  // 0x0124
402                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
403     #endif
404     #if (PAD_LIST || CC_ChangePPS)
405             (COMMAND_ATTRIBUTES)(CC_ChangePPS                  *  // 0x0125
406                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
407     #endif
408     #if (PAD_LIST || CC_Clear)
409             (COMMAND_ATTRIBUTES)(CC_Clear                      *  // 0x0126
410                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
411     #endif
412     #if (PAD_LIST || CC_ClearControl)
413             (COMMAND_ATTRIBUTES)(CC_ClearControl               *  // 0x0127
414                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
415     #endif
416     #if (PAD_LIST || CC_ClockSet)
417             (COMMAND_ATTRIBUTES)(CC_ClockSet                   *  // 0x0128
418                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
419     #endif
420     #if (PAD_LIST || CC_HierarchyChangeAuth)
421             (COMMAND_ATTRIBUTES)(CC_HierarchyChangeAuth        *  // 0x0129
422                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
423     #endif
424     #if (PAD_LIST || CC_NV_DefineSpace)
425             (COMMAND_ATTRIBUTES)(CC_NV_DefineSpace             *  // 0x012A
426                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
427     #endif
428     #if (PAD_LIST || CC_PCR_Allocate)
429             (COMMAND_ATTRIBUTES)(CC_PCR_Allocate               *  // 0x012B
430                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
431     #endif
432     #if (PAD_LIST || CC_PCR_SetAuthPolicy)
433             (COMMAND_ATTRIBUTES)(CC_PCR_SetAuthPolicy          *  // 0x012C
434                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
435     #endif
436     #if (PAD_LIST || CC_PP_Commands)
437             (COMMAND_ATTRIBUTES)(CC_PP_Commands                *  // 0x012D
438                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_REQUIRED)),
439     #endif
440     #if (PAD_LIST || CC_SetPrimaryPolicy)
441             (COMMAND_ATTRIBUTES)(CC_SetPrimaryPolicy           *  // 0x012E
442                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
443     #endif
444     #if (PAD_LIST || CC_FieldUpgradeStart)
445             (COMMAND_ATTRIBUTES)(CC_FieldUpgradeStart          *  // 0x012F
446                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+PP_COMMAND)),
447     #endif
448     #if (PAD_LIST || CC_ClockRateAdjust)
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
449         (COMMAND_ATTRIBUTES)(CC_ClockRateAdjust            *  // 0x0130
450             (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
451 #endif
452 #if (PAD_LIST || CC_CreatePrimary)
453         (COMMAND_ATTRIBUTES)(CC_CreatePrimary              *  // 0x0131
454             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND+ENCRYPT_2+R_HANDLE)),
455 #endif
456 #if (PAD_LIST || CC_NV_GlobalWriteLock)
457         (COMMAND_ATTRIBUTES)(CC_NV_GlobalWriteLock         *  // 0x0132
458             (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
459 #endif
460 #if (PAD_LIST || CC_GetCommandAuditDigest)
461         (COMMAND_ATTRIBUTES)(CC_GetCommandAuditDigest      *  // 0x0133
462             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
463 #endif
464 #if (PAD_LIST || CC_NV_Increment)
465         (COMMAND_ATTRIBUTES)(CC_NV_Increment               *  // 0x0134
466             (IS_IMPLEMENTED+HANDLE_1_USER)),
467 #endif
468 #if (PAD_LIST || CC_NV_SetBits)
469         (COMMAND_ATTRIBUTES)(CC_NV_SetBits                 *  // 0x0135
470             (IS_IMPLEMENTED+HANDLE_1_USER)),
471 #endif
472 #if (PAD_LIST || CC_NV_Extend)
473         (COMMAND_ATTRIBUTES)(CC_NV_Extend                  *  // 0x0136
474             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
475 #endif
476 #if (PAD_LIST || CC_NV_Write)
477         (COMMAND_ATTRIBUTES)(CC_NV_Write                   *  // 0x0137
478             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
479 #endif
480 #if (PAD_LIST || CC_NV_WriteLock)
481         (COMMAND_ATTRIBUTES)(CC_NV_WriteLock               *  // 0x0138
482             (IS_IMPLEMENTED+HANDLE_1_USER)),
483 #endif
484 #if (PAD_LIST || CC_DictionaryAttackLockReset)
485         (COMMAND_ATTRIBUTES)(CC_DictionaryAttackLockReset  *  // 0x0139
486             (IS_IMPLEMENTED+HANDLE_1_USER)),
487 #endif
488 #if (PAD_LIST || CC_DictionaryAttackParameters)
489         (COMMAND_ATTRIBUTES)(CC_DictionaryAttackParameters *  // 0x013A
490             (IS_IMPLEMENTED+HANDLE_1_USER)),
491 #endif
492 #if (PAD_LIST || CC_NV_ChangeAuth)
493         (COMMAND_ATTRIBUTES)(CC_NV_ChangeAuth              *  // 0x013B
494             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN)),
495 #endif
496 #if (PAD_LIST || CC_PCR_Event)
497         (COMMAND_ATTRIBUTES)(CC_PCR_Event                  *  // 0x013C
498             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
499 #endif
500 #if (PAD_LIST || CC_PCR_Reset)
501         (COMMAND_ATTRIBUTES)(CC_PCR_Reset                  *  // 0x013D
502             (IS_IMPLEMENTED+HANDLE_1_USER)),
503 #endif
504 #if (PAD_LIST || CC_SequenceComplete)
505         (COMMAND_ATTRIBUTES)(CC_SequenceComplete           *  // 0x013E
506             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
507 #endif
508 #if (PAD_LIST || CC_SetAlgorithmSet)
509         (COMMAND_ATTRIBUTES)(CC_SetAlgorithmSet            *  // 0x013F
510             (IS_IMPLEMENTED+HANDLE_1_USER)),
511 #endif
512 #if (PAD_LIST || CC_SetCommandCodeAuditStatus)
513         (COMMAND_ATTRIBUTES)(CC_SetCommandCodeAuditStatus  *  // 0x0140
514             (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
```

 Copyright © TCG 2006-2020

```
515 #endif
516 #if (PAD_LIST || CC_FieldUpgradeData)
517         (COMMAND_ATTRIBUTES)(CC_FieldUpgradeData          *  // 0x0141
518             (IS_IMPLEMENTED+DECRYPT_2)),
519 #endif
520 #if (PAD_LIST || CC_IncrementalSelfTest)
521         (COMMAND_ATTRIBUTES)(CC_IncrementalSelfTest       *  // 0x0142
522             (IS_IMPLEMENTED)),
523 #endif
524 #if (PAD_LIST || CC_SelfTest)
525         (COMMAND_ATTRIBUTES)(CC_SelfTest                  *  // 0x0143
526             (IS_IMPLEMENTED)),
527 #endif
528 #if (PAD_LIST || CC_Startup)
529         (COMMAND_ATTRIBUTES)(CC_Startup                   *  // 0x0144
530             (IS_IMPLEMENTED+NO_SESSIONS)),
531 #endif
532 #if (PAD_LIST || CC_Shutdown)
533         (COMMAND_ATTRIBUTES)(CC_Shutdown                  *  // 0x0145
534             (IS_IMPLEMENTED)),
535 #endif
536 #if (PAD_LIST || CC_StirRandom)
537         (COMMAND_ATTRIBUTES)(CC_StirRandom                *  // 0x0146
538             (IS_IMPLEMENTED+DECRYPT_2)),
539 #endif
540 #if (PAD_LIST || CC_ActivateCredential)
541         (COMMAND_ATTRIBUTES)(CC_ActivateCredential        *  // 0x0147
542             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
543 #endif
544 #if (PAD_LIST || CC_Certify)
545         (COMMAND_ATTRIBUTES)(CC_Certify                   *  // 0x0148
546             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
547 #endif
548 #if (PAD_LIST || CC_PolicyNV)
549         (COMMAND_ATTRIBUTES)(CC_PolicyNV                  *  // 0x0149
550             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ALLOW_TRIAL)),
551 #endif
552 #if (PAD_LIST || CC_CertifyCreation)
553         (COMMAND_ATTRIBUTES)(CC_CertifyCreation           *  // 0x014A
554             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
555 #endif
556 #if (PAD_LIST || CC_Duplicate)
557         (COMMAND_ATTRIBUTES)(CC_Duplicate                 *  // 0x014B
558             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_DUP+ENCRYPT_2)),
559 #endif
560 #if (PAD_LIST || CC_GetTime)
561         (COMMAND_ATTRIBUTES)(CC_GetTime                   *  // 0x014C
562             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
563 #endif
564 #if (PAD_LIST || CC_GetSessionAuditDigest)
565         (COMMAND_ATTRIBUTES)(CC_GetSessionAuditDigest     *  // 0x014D
566             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
567 #endif
568 #if (PAD_LIST || CC_NV_Read)
569         (COMMAND_ATTRIBUTES)(CC_NV_Read                   *  // 0x014E
570             (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
571 #endif
572 #if (PAD_LIST || CC_NV_ReadLock)
573         (COMMAND_ATTRIBUTES)(CC_NV_ReadLock               *  // 0x014F
574             (IS_IMPLEMENTED+HANDLE_1_USER)),
575 #endif
576 #if (PAD_LIST || CC_ObjectChangeAuth)
577         (COMMAND_ATTRIBUTES)(CC_ObjectChangeAuth          *  // 0x0150
578             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+ENCRYPT_2)),
579 #endif
580 #if (PAD_LIST || CC_PolicySecret)
```

 Copyright © TCG 2006-2020

```
581         (COMMAND_ATTRIBUTES)(CC_PolicySecret                *  // 0x0151
582             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ALLOW_TRIAL+ENCRYPT_2)),
583  #endif
584  #if (PAD_LIST || CC_Rewrap)
585         (COMMAND_ATTRIBUTES)(CC_Rewrap                      *  // 0x0152
586             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
587  #endif
588  #if (PAD_LIST || CC_Create)
589         (COMMAND_ATTRIBUTES)(CC_Create                      *  // 0x0153
590             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
591  #endif
592  #if (PAD_LIST || CC_ECDH_ZGen)
593         (COMMAND_ATTRIBUTES)(CC_ECDH_ZGen                   *  // 0x0154
594             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
595  #endif
596  #if (PAD_LIST || (CC_HMAC || CC_MAC))
597         (COMMAND_ATTRIBUTES)((CC_HMAC || CC_MAC)            *  // 0x0155
598             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
599  #endif
600  #if (PAD_LIST || CC_Import)
601         (COMMAND_ATTRIBUTES)(CC_Import                      *  // 0x0156
602             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
603  #endif
604  #if (PAD_LIST || CC_Load)
605         (COMMAND_ATTRIBUTES)(CC_Load                        *  // 0x0157
606             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2+R_HANDLE)),
607  #endif
608  #if (PAD_LIST || CC_Quote)
609         (COMMAND_ATTRIBUTES)(CC_Quote                       *  // 0x0158
610             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
611  #endif
612  #if (PAD_LIST || CC_RSA_Decrypt)
613         (COMMAND_ATTRIBUTES)(CC_RSA_Decrypt                 *  // 0x0159
614             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
615  #endif
616  #if (PAD_LIST )
617         (COMMAND_ATTRIBUTES)(0),                               // 0x015A
618  #endif
619  #if (PAD_LIST || (CC_HMAC_Start || CC_MAC_Start))
620         (COMMAND_ATTRIBUTES)((CC_HMAC_Start || CC_MAC_Start) *  // 0x015B
621             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+R_HANDLE)),
622  #endif
623  #if (PAD_LIST || CC_SequenceUpdate)
624         (COMMAND_ATTRIBUTES)(CC_SequenceUpdate              *  // 0x015C
625             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
626  #endif
627  #if (PAD_LIST || CC_Sign)
628         (COMMAND_ATTRIBUTES)(CC_Sign                        *  // 0x015D
629             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
630  #endif
631  #if (PAD_LIST || CC_Unseal)
632         (COMMAND_ATTRIBUTES)(CC_Unseal                      *  // 0x015E
633             (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
634  #endif
635  #if (PAD_LIST )
636         (COMMAND_ATTRIBUTES)(0),                               // 0x015F
637  #endif
638  #if (PAD_LIST || CC_PolicySigned)
639         (COMMAND_ATTRIBUTES)(CC_PolicySigned                *  // 0x0160
640             (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL+ENCRYPT_2)),
641  #endif
642  #if (PAD_LIST || CC_ContextLoad)
643         (COMMAND_ATTRIBUTES)(CC_ContextLoad                 *  // 0x0161
644             (IS_IMPLEMENTED+NO_SESSIONS+R_HANDLE)),
645  #endif
646  #if (PAD_LIST || CC_ContextSave)
```

 Copyright © TCG 2006-2020

```
647             (COMMAND_ATTRIBUTES)(CC_ContextSave                  *  // 0x0162
648                 (IS_IMPLEMENTED+NO_SESSIONS)),
649     #endif
650     #if (PAD_LIST || CC_ECDH_KeyGen)
651             (COMMAND_ATTRIBUTES)(CC_ECDH_KeyGen                  *  // 0x0163
652                 (IS_IMPLEMENTED+ENCRYPT_2)),
653     #endif
654     #if (PAD_LIST || CC_EncryptDecrypt)
655             (COMMAND_ATTRIBUTES)(CC_EncryptDecrypt               *  // 0x0164
656                 (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
657     #endif
658     #if (PAD_LIST || CC_FlushContext)
659             (COMMAND_ATTRIBUTES)(CC_FlushContext                 *  // 0x0165
660                 (IS_IMPLEMENTED+NO_SESSIONS)),
661     #endif
662     #if (PAD_LIST )
663             (COMMAND_ATTRIBUTES)(0),                                // 0x0166
664     #endif
665     #if (PAD_LIST || CC_LoadExternal)
666             (COMMAND_ATTRIBUTES)(CC_LoadExternal                 *  // 0x0167
667                 (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),
668     #endif
669     #if (PAD_LIST || CC_MakeCredential)
670             (COMMAND_ATTRIBUTES)(CC_MakeCredential               *  // 0x0168
671                 (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
672     #endif
673     #if (PAD_LIST || CC_NV_ReadPublic)
674             (COMMAND_ATTRIBUTES)(CC_NV_ReadPublic                *  // 0x0169
675                 (IS_IMPLEMENTED+ENCRYPT_2)),
676     #endif
677     #if (PAD_LIST || CC_PolicyAuthorize)
678             (COMMAND_ATTRIBUTES)(CC_PolicyAuthorize              *  // 0x016A
679                 (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
680     #endif
681     #if (PAD_LIST || CC_PolicyAuthValue)
682             (COMMAND_ATTRIBUTES)(CC_PolicyAuthValue              *  // 0x016B
683                 (IS_IMPLEMENTED+ALLOW_TRIAL)),
684     #endif
685     #if (PAD_LIST || CC_PolicyCommandCode)
686             (COMMAND_ATTRIBUTES)(CC_PolicyCommandCode            *  // 0x016C
687                 (IS_IMPLEMENTED+ALLOW_TRIAL)),
688     #endif
689     #if (PAD_LIST || CC_PolicyCounterTimer)
690             (COMMAND_ATTRIBUTES)(CC_PolicyCounterTimer           *  // 0x016D
691                 (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
692     #endif
693     #if (PAD_LIST || CC_PolicyCpHash)
694             (COMMAND_ATTRIBUTES)(CC_PolicyCpHash                 *  // 0x016E
695                 (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
696     #endif
697     #if (PAD_LIST || CC_PolicyLocality)
698             (COMMAND_ATTRIBUTES)(CC_PolicyLocality               *  // 0x016F
699                 (IS_IMPLEMENTED+ALLOW_TRIAL)),
700     #endif
701     #if (PAD_LIST || CC_PolicyNameHash)
702             (COMMAND_ATTRIBUTES)(CC_PolicyNameHash               *  // 0x0170
703                 (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
704     #endif
705     #if (PAD_LIST || CC_PolicyOR)
706             (COMMAND_ATTRIBUTES)(CC_PolicyOR                     *  // 0x0171
707                 (IS_IMPLEMENTED+ALLOW_TRIAL)),
708     #endif
709     #if (PAD_LIST || CC_PolicyTicket)
710             (COMMAND_ATTRIBUTES)(CC_PolicyTicket                 *  // 0x0172
711                 (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
712     #endif
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
713  #if (PAD_LIST || CC_ReadPublic)
714          (COMMAND_ATTRIBUTES)(CC_ReadPublic                    *  // 0x0173
715              (IS_IMPLEMENTED+ENCRYPT_2)),
716  #endif
717  #if (PAD_LIST || CC_RSA_Encrypt)
718          (COMMAND_ATTRIBUTES)(CC_RSA_Encrypt                   *  // 0x0174
719              (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
720  #endif
721  #if (PAD_LIST )
722          (COMMAND_ATTRIBUTES)(0),                                 // 0x0175
723  #endif
724  #if (PAD_LIST || CC_StartAuthSession)
725          (COMMAND_ATTRIBUTES)(CC_StartAuthSession              *  // 0x0176
726              (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),
727  #endif
728  #if (PAD_LIST || CC_VerifySignature)
729          (COMMAND_ATTRIBUTES)(CC_VerifySignature               *  // 0x0177
730              (IS_IMPLEMENTED+DECRYPT_2)),
731  #endif
732  #if (PAD_LIST || CC_ECC_Parameters)
733          (COMMAND_ATTRIBUTES)(CC_ECC_Parameters                *  // 0x0178
734              (IS_IMPLEMENTED)),
735  #endif
736  #if (PAD_LIST || CC_FirmwareRead)
737          (COMMAND_ATTRIBUTES)(CC_FirmwareRead                  *  // 0x0179
738              (IS_IMPLEMENTED+ENCRYPT_2)),
739  #endif
740  #if (PAD_LIST || CC_GetCapability)
741          (COMMAND_ATTRIBUTES)(CC_GetCapability                 *  // 0x017A
742              (IS_IMPLEMENTED)),
743  #endif
744  #if (PAD_LIST || CC_GetRandom)
745          (COMMAND_ATTRIBUTES)(CC_GetRandom                     *  // 0x017B
746              (IS_IMPLEMENTED+ENCRYPT_2)),
747  #endif
748  #if (PAD_LIST || CC_GetTestResult)
749          (COMMAND_ATTRIBUTES)(CC_GetTestResult                 *  // 0x017C
750              (IS_IMPLEMENTED+ENCRYPT_2)),
751  #endif
752  #if (PAD_LIST || CC_Hash)
753          (COMMAND_ATTRIBUTES)(CC_Hash                          *  // 0x017D
754              (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
755  #endif
756  #if (PAD_LIST || CC_PCR_Read)
757          (COMMAND_ATTRIBUTES)(CC_PCR_Read                      *  // 0x017E
758              (IS_IMPLEMENTED)),
759  #endif
760  #if (PAD_LIST || CC_PolicyPCR)
761          (COMMAND_ATTRIBUTES)(CC_PolicyPCR                     *  // 0x017F
762              (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
763  #endif
764  #if (PAD_LIST || CC_PolicyRestart)
765          (COMMAND_ATTRIBUTES)(CC_PolicyRestart                 *  // 0x0180
766              (IS_IMPLEMENTED+ALLOW_TRIAL)),
767  #endif
768  #if (PAD_LIST || CC_ReadClock)
769          (COMMAND_ATTRIBUTES)(CC_ReadClock                     *  // 0x0181
770              (IS_IMPLEMENTED)),
771  #endif
772  #if (PAD_LIST || CC_PCR_Extend)
773          (COMMAND_ATTRIBUTES)(CC_PCR_Extend                    *  // 0x0182
774              (IS_IMPLEMENTED+HANDLE_1_USER)),
775  #endif
776  #if (PAD_LIST || CC_PCR_SetAuthValue)
777          (COMMAND_ATTRIBUTES)(CC_PCR_SetAuthValue              *  // 0x0183
778              (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
779 #endif
780 #if (PAD_LIST || CC_NV_Certify)
781         (COMMAND_ATTRIBUTES)(CC_NV_Certify                    *  // 0x0184
782             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
783 #endif
784 #if (PAD_LIST || CC_EventSequenceComplete)
785         (COMMAND_ATTRIBUTES)(CC_EventSequenceComplete         *  // 0x0185
786             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER)),
787 #endif
788 #if (PAD_LIST || CC_HashSequenceStart)
789         (COMMAND_ATTRIBUTES)(CC_HashSequenceStart             *  // 0x0186
790             (IS_IMPLEMENTED+DECRYPT_2+R_HANDLE)),
791 #endif
792 #if (PAD_LIST || CC_PolicyPhysicalPresence)
793         (COMMAND_ATTRIBUTES)(CC_PolicyPhysicalPresence        *  // 0x0187
794             (IS_IMPLEMENTED+ALLOW_TRIAL)),
795 #endif
796 #if (PAD_LIST || CC_PolicyDuplicationSelect)
797         (COMMAND_ATTRIBUTES)(CC_PolicyDuplicationSelect       *  // 0x0188
798             (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
799 #endif
800 #if (PAD_LIST || CC_PolicyGetDigest)
801         (COMMAND_ATTRIBUTES)(CC_PolicyGetDigest               *  // 0x0189
802             (IS_IMPLEMENTED+ALLOW_TRIAL+ENCRYPT_2)),
803 #endif
804 #if (PAD_LIST || CC_TestParms)
805         (COMMAND_ATTRIBUTES)(CC_TestParms                     *  // 0x018A
806             (IS_IMPLEMENTED)),
807 #endif
808 #if (PAD_LIST || CC_Commit)
809         (COMMAND_ATTRIBUTES)(CC_Commit                        *  // 0x018B
810             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
811 #endif
812 #if (PAD_LIST || CC_PolicyPassword)
813         (COMMAND_ATTRIBUTES)(CC_PolicyPassword                *  // 0x018C
814             (IS_IMPLEMENTED+ALLOW_TRIAL)),
815 #endif
816 #if (PAD_LIST || CC_ZGen_2Phase)
817         (COMMAND_ATTRIBUTES)(CC_ZGen_2Phase                   *  // 0x018D
818             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
819 #endif
820 #if (PAD_LIST || CC_EC_Ephemeral)
821         (COMMAND_ATTRIBUTES)(CC_EC_Ephemeral                  *  // 0x018E
822             (IS_IMPLEMENTED+ENCRYPT_2)),
823 #endif
824 #if (PAD_LIST || CC_PolicyNvWritten)
825         (COMMAND_ATTRIBUTES)(CC_PolicyNvWritten               *  // 0x018F
826             (IS_IMPLEMENTED+ALLOW_TRIAL)),
827 #endif
828 #if (PAD_LIST || CC_PolicyTemplate)
829         (COMMAND_ATTRIBUTES)(CC_PolicyTemplate                *  // 0x0190
830             (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
831 #endif
832 #if (PAD_LIST || CC_CreateLoaded)
833         (COMMAND_ATTRIBUTES)(CC_CreateLoaded                  *  // 0x0191
834             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND+ENCRYPT_2+R_HANDLE)),
835 #endif
836 #if (PAD_LIST || CC_PolicyAuthorizeNV)
837         (COMMAND_ATTRIBUTES)(CC_PolicyAuthorizeNV             *  // 0x0192
838             (IS_IMPLEMENTED+HANDLE_1_USER+ALLOW_TRIAL)),
839 #endif
840 #if (PAD_LIST || CC_EncryptDecrypt2)
841         (COMMAND_ATTRIBUTES)(CC_EncryptDecrypt2               *  // 0x0193
842             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
843 #endif
844 #if (PAD_LIST || CC_AC_GetCapability)
```


 Copyright © TCG 2006-2020

```
845         (COMMAND_ATTRIBUTES)(CC_AC_GetCapability           *  // 0x0194
846             (IS_IMPLEMENTED)),
847 #endif
848 #if (PAD_LIST || CC_AC_Send)
849         (COMMAND_ATTRIBUTES)(CC_AC_Send                    *  // 0x0195
850             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_DUP+HANDLE_2_USER)),
851 #endif
852 #if (PAD_LIST || CC_Policy_AC_SendSelect)
853         (COMMAND_ATTRIBUTES)(CC_Policy_AC_SendSelect       *  // 0x0196
854             (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
855 #endif
856 #if (PAD_LIST || CC_CertifyX509)
857         (COMMAND_ATTRIBUTES)(CC_CertifyX509                *  // 0x0197
858             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
859 #endif
860 #if (PAD_LIST || CC_ACT_SetTimeout)
861         (COMMAND_ATTRIBUTES)(CC_ACT_SetTimeout             *  // 0x0198
862             (IS_IMPLEMENTED+HANDLE_1_USER)),
863 #endif
864 #if (PAD_LIST || CC_Vendor_TCG_Test)
865         (COMMAND_ATTRIBUTES)(CC_Vendor_TCG_Test            *  // 0x0000
866             (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
867 #endif
868         0
869 };
870 #endif  // _COMMAND_CODE_ATTRIBUTES_
```

 Copyright © TCG 2006-2020

5.5 CommandAttributes.h

The attributes defined in this file are produced by the parser that creates the structure definitions from Part 3. The attributes are defined in that parser and should track the attributes being tested in CommandCodeAttributes.c. Generally, when an attribute is added to this list, new code will be needed in CommandCodeAttributes.c to test it.

```
 1  #ifndef COMMAND_ATTRIBUTES_H
 2  #define COMMAND_ATTRIBUTES_H
 3  typedef UINT16              COMMAND_ATTRIBUTES;
 4  #define NOT_IMPLEMENTED     (COMMAND_ATTRIBUTES)(0)
 5  #define ENCRYPT_2           ((COMMAND_ATTRIBUTES)1 << 0)
 6  #define ENCRYPT_4           ((COMMAND_ATTRIBUTES)1 << 1)
 7  #define DECRYPT_2           ((COMMAND_ATTRIBUTES)1 << 2)
 8  #define DECRYPT_4           ((COMMAND_ATTRIBUTES)1 << 3)
 9  #define HANDLE_1_USER       ((COMMAND_ATTRIBUTES)1 << 4)
10  #define HANDLE_1_ADMIN      ((COMMAND_ATTRIBUTES)1 << 5)
11  #define HANDLE_1_DUP        ((COMMAND_ATTRIBUTES)1 << 6)
12  #define HANDLE_2_USER       ((COMMAND_ATTRIBUTES)1 << 7)
13  #define PP_COMMAND          ((COMMAND_ATTRIBUTES)1 << 8)
14  #define IS_IMPLEMENTED      ((COMMAND_ATTRIBUTES)1 << 9)
15  #define NO_SESSIONS         ((COMMAND_ATTRIBUTES)1 << 10)
16  #define NV_COMMAND          ((COMMAND_ATTRIBUTES)1 << 11)
17  #define PP_REQUIRED         ((COMMAND_ATTRIBUTES)1 << 12)
18  #define R_HANDLE            ((COMMAND_ATTRIBUTES)1 << 13)
19  #define ALLOW_TRIAL         ((COMMAND_ATTRIBUTES)1 << 14)
20  #endif // COMMAND_ATTRIBUTES_H
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

5.6 CommandDispatchData.h

This file should only be included by CommandCodeAttibutes.c

```
1   #ifdef _COMMAND_TABLE_DISPATCH_
```

Define the stop value

```
2   #define END_OF_LIST     0xff
3   #define ADD_FLAG        0x80
```

These macros provide some variability in how the data is encoded. They also make the lines a little shorter. ;-)

```
4   #if TABLE_DRIVEN_MARSHAL
5   #   define UNMARSHAL_DISPATCH(name)    (marshalIndex_t)name##_MARSHAL_REF
6   #   define MARSHAL_DISPATCH(name)      (marshalIndex_t)name##_MARSHAL_REF
7   #   define _UNMARSHAL_T_    marshalIndex_t
8   #   define _MARSHAL_T_      marshalIndex_t
9   #
10  #else
11  #   define UNMARSHAL_DISPATCH(name)    (UNMARSHAL_t)name##_Unmarshal
12  #   define MARSHAL_DISPATCH(name)      (MARSHAL_t)name##_Marshal
13  #   define _UNMARSHAL_T_    UNMARSHAL_t
14  #   define _MARSHAL_T_      MARSHAL_t
15  #endif
```

The UnmarshalArray() contains the dispatch functions for the unmarshaling code. The defines in this array are used to make it easier to cross reference the unmarshaling values in the types array of each command

```
16  const _UNMARSHAL_T_ UnmarshalArray[] = {
17  #define TPMI_DH_CONTEXT_H_UNMARSHAL             0
18              UNMARSHAL_DISPATCH(TPMI_DH_CONTEXT),
19  #define TPMI_RH_AC_H_UNMARSHAL                  (TPMI_DH_CONTEXT_H_UNMARSHAL + 1)
20              UNMARSHAL_DISPATCH(TPMI_RH_AC),
21  #define TPMI_RH_ACT_H_UNMARSHAL                 (TPMI_RH_AC_H_UNMARSHAL + 1)
22              UNMARSHAL_DISPATCH(TPMI_RH_ACT),
23  #define TPMI_RH_CLEAR_H_UNMARSHAL               (TPMI_RH_ACT_H_UNMARSHAL + 1)
24              UNMARSHAL_DISPATCH(TPMI_RH_CLEAR),
25  #define TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL      (TPMI_RH_CLEAR_H_UNMARSHAL + 1)
26              UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY_AUTH),
27  #define TPMI_RH_HIERARCHY_POLICY_H_UNMARSHAL    \
28              (TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL + 1)
29              UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY_POLICY),
30  #define TPMI_RH_LOCKOUT_H_UNMARSHAL             \
31              (TPMI_RH_HIERARCHY_POLICY_H_UNMARSHAL + 1)
32              UNMARSHAL_DISPATCH(TPMI_RH_LOCKOUT),
33  #define TPMI_RH_NV_AUTH_H_UNMARSHAL             (TPMI_RH_LOCKOUT_H_UNMARSHAL + 1)
34              UNMARSHAL_DISPATCH(TPMI_RH_NV_AUTH),
35  #define TPMI_RH_NV_INDEX_H_UNMARSHAL            (TPMI_RH_NV_AUTH_H_UNMARSHAL + 1)
36              UNMARSHAL_DISPATCH(TPMI_RH_NV_INDEX),
37  #define TPMI_RH_PLATFORM_H_UNMARSHAL            (TPMI_RH_NV_INDEX_H_UNMARSHAL + 1)
38              UNMARSHAL_DISPATCH(TPMI_RH_PLATFORM),
39  #define TPMI_RH_PROVISION_H_UNMARSHAL           (TPMI_RH_PLATFORM_H_UNMARSHAL + 1)
40              UNMARSHAL_DISPATCH(TPMI_RH_PROVISION),
41  #define TPMI_SH_HMAC_H_UNMARSHAL                (TPMI_RH_PROVISION_H_UNMARSHAL + 1)
42              UNMARSHAL_DISPATCH(TPMI_SH_HMAC),
43  #define TPMI_SH_POLICY_H_UNMARSHAL              (TPMI_SH_HMAC_H_UNMARSHAL + 1)
44              UNMARSHAL_DISPATCH(TPMI_SH_POLICY),
45  // HANDLE_FIRST_FLAG_TYPE is the first handle that needs a flag when called.
46  #define HANDLE_FIRST_FLAG_TYPE                  (TPMI_SH_POLICY_H_UNMARSHAL + 1)
```

 Copyright © TCG 2006-2020

```
47   #define TPMI_DH_ENTITY_H_UNMARSHAL              (TPMI_SH_POLICY_H_UNMARSHAL + 1)
48              UNMARSHAL_DISPATCH(TPMI_DH_ENTITY),
49   #define TPMI_DH_OBJECT_H_UNMARSHAL              (TPMI_DH_ENTITY_H_UNMARSHAL + 1)
50              UNMARSHAL_DISPATCH(TPMI_DH_OBJECT),
51   #define TPMI_DH_PARENT_H_UNMARSHAL              (TPMI_DH_OBJECT_H_UNMARSHAL + 1)
52              UNMARSHAL_DISPATCH(TPMI_DH_PARENT),
53   #define TPMI_DH_PCR_H_UNMARSHAL                 (TPMI_DH_PARENT_H_UNMARSHAL + 1)
54              UNMARSHAL_DISPATCH(TPMI_DH_PCR),
55   #define TPMI_RH_ENDORSEMENT_H_UNMARSHAL         (TPMI_DH_PCR_H_UNMARSHAL + 1)
56              UNMARSHAL_DISPATCH(TPMI_RH_ENDORSEMENT),
57   #define TPMI_RH_HIERARCHY_H_UNMARSHAL           \
58              (TPMI_RH_ENDORSEMENT_H_UNMARSHAL + 1)
59              UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY),
60   // PARAMETER_FIRST_TYPE marks the end of the handle list.
61   #define PARAMETER_FIRST_TYPE                    (TPMI_RH_HIERARCHY_H_UNMARSHAL + 1)
62   #define TPM2B_DATA_P_UNMARSHAL                  (TPMI_RH_HIERARCHY_H_UNMARSHAL + 1)
63              UNMARSHAL_DISPATCH(TPM2B_DATA),
64   #define TPM2B_DIGEST_P_UNMARSHAL                (TPM2B_DATA_P_UNMARSHAL + 1)
65              UNMARSHAL_DISPATCH(TPM2B_DIGEST),
66   #define TPM2B_ECC_PARAMETER_P_UNMARSHAL         (TPM2B_DIGEST_P_UNMARSHAL + 1)
67              UNMARSHAL_DISPATCH(TPM2B_ECC_PARAMETER),
68   #define TPM2B_ECC_POINT_P_UNMARSHAL             \
69              (TPM2B_ECC_PARAMETER_P_UNMARSHAL + 1)
70              UNMARSHAL_DISPATCH(TPM2B_ECC_POINT),
71   #define TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL      (TPM2B_ECC_POINT_P_UNMARSHAL + 1)
72              UNMARSHAL_DISPATCH(TPM2B_ENCRYPTED_SECRET),
73   #define TPM2B_EVENT_P_UNMARSHAL                 \
74              (TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL + 1)
75              UNMARSHAL_DISPATCH(TPM2B_EVENT),
76   #define TPM2B_ID_OBJECT_P_UNMARSHAL             (TPM2B_EVENT_P_UNMARSHAL + 1)
77              UNMARSHAL_DISPATCH(TPM2B_ID_OBJECT),
78   #define TPM2B_IV_P_UNMARSHAL                    (TPM2B_ID_OBJECT_P_UNMARSHAL + 1)
79              UNMARSHAL_DISPATCH(TPM2B_IV),
80   #define TPM2B_MAX_BUFFER_P_UNMARSHAL            (TPM2B_IV_P_UNMARSHAL + 1)
81              UNMARSHAL_DISPATCH(TPM2B_MAX_BUFFER),
82   #define TPM2B_MAX_NV_BUFFER_P_UNMARSHAL         (TPM2B_MAX_BUFFER_P_UNMARSHAL + 1)
83              UNMARSHAL_DISPATCH(TPM2B_MAX_NV_BUFFER),
84   #define TPM2B_NAME_P_UNMARSHAL                  \
85              (TPM2B_MAX_NV_BUFFER_P_UNMARSHAL + 1)
86              UNMARSHAL_DISPATCH(TPM2B_NAME),
87   #define TPM2B_NV_PUBLIC_P_UNMARSHAL             (TPM2B_NAME_P_UNMARSHAL + 1)
88              UNMARSHAL_DISPATCH(TPM2B_NV_PUBLIC),
89   #define TPM2B_PRIVATE_P_UNMARSHAL               (TPM2B_NV_PUBLIC_P_UNMARSHAL + 1)
90              UNMARSHAL_DISPATCH(TPM2B_PRIVATE),
91   #define TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL        (TPM2B_PRIVATE_P_UNMARSHAL + 1)
92              UNMARSHAL_DISPATCH(TPM2B_PUBLIC_KEY_RSA),
93   #define TPM2B_SENSITIVE_P_UNMARSHAL             \
94              (TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL + 1)
95              UNMARSHAL_DISPATCH(TPM2B_SENSITIVE),
96   #define TPM2B_SENSITIVE_CREATE_P_UNMARSHAL      (TPM2B_SENSITIVE_P_UNMARSHAL + 1)
97              UNMARSHAL_DISPATCH(TPM2B_SENSITIVE_CREATE),
98   #define TPM2B_SENSITIVE_DATA_P_UNMARSHAL        \
99              (TPM2B_SENSITIVE_CREATE_P_UNMARSHAL + 1)
100             UNMARSHAL_DISPATCH(TPM2B_SENSITIVE_DATA),
101  #define TPM2B_TEMPLATE_P_UNMARSHAL              \
102             (TPM2B_SENSITIVE_DATA_P_UNMARSHAL + 1)
103             UNMARSHAL_DISPATCH(TPM2B_TEMPLATE),
104  #define TPM2B_TIMEOUT_P_UNMARSHAL               (TPM2B_TEMPLATE_P_UNMARSHAL + 1)
105             UNMARSHAL_DISPATCH(TPM2B_TIMEOUT),
106  #define TPMI_DH_CONTEXT_P_UNMARSHAL             (TPM2B_TIMEOUT_P_UNMARSHAL + 1)
107             UNMARSHAL_DISPATCH(TPMI_DH_CONTEXT),
108  #define TPMI_DH_PERSISTENT_P_UNMARSHAL          (TPMI_DH_CONTEXT_P_UNMARSHAL + 1)
109             UNMARSHAL_DISPATCH(TPMI_DH_PERSISTENT),
110  #define TPMI_ECC_CURVE_P_UNMARSHAL              (TPMI_DH_PERSISTENT_P_UNMARSHAL + 1)
111             UNMARSHAL_DISPATCH(TPMI_ECC_CURVE),
112  #define TPMI_YES_NO_P_UNMARSHAL                 (TPMI_ECC_CURVE_P_UNMARSHAL + 1)
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
113                 UNMARSHAL_DISPATCH(TPMI_YES_NO),
114     #define TPML_ALG_P_UNMARSHAL                    (TPMI_YES_NO_P_UNMARSHAL + 1)
115                 UNMARSHAL_DISPATCH(TPML_ALG),
116     #define TPML_CC_P_UNMARSHAL                     (TPML_ALG_P_UNMARSHAL + 1)
117                 UNMARSHAL_DISPATCH(TPML_CC),
118     #define TPML_DIGEST_P_UNMARSHAL                 (TPML_CC_P_UNMARSHAL + 1)
119                 UNMARSHAL_DISPATCH(TPML_DIGEST),
120     #define TPML_DIGEST_VALUES_P_UNMARSHAL          (TPML_DIGEST_P_UNMARSHAL + 1)
121                 UNMARSHAL_DISPATCH(TPML_DIGEST_VALUES),
122     #define TPML_PCR_SELECTION_P_UNMARSHAL          (TPML_DIGEST_VALUES_P_UNMARSHAL + 1)
123                 UNMARSHAL_DISPATCH(TPML_PCR_SELECTION),
124     #define TPMS_CONTEXT_P_UNMARSHAL                (TPML_PCR_SELECTION_P_UNMARSHAL + 1)
125                 UNMARSHAL_DISPATCH(TPMS_CONTEXT),
126     #define TPMT_PUBLIC_PARMS_P_UNMARSHAL           (TPMS_CONTEXT_P_UNMARSHAL + 1)
127                 UNMARSHAL_DISPATCH(TPMT_PUBLIC_PARMS),
128     #define TPMT_TK_AUTH_P_UNMARSHAL                (TPMT_PUBLIC_PARMS_P_UNMARSHAL + 1)
129                 UNMARSHAL_DISPATCH(TPMT_TK_AUTH),
130     #define TPMT_TK_CREATION_P_UNMARSHAL            (TPMT_TK_AUTH_P_UNMARSHAL + 1)
131                 UNMARSHAL_DISPATCH(TPMT_TK_CREATION),
132     #define TPMT_TK_HASHCHECK_P_UNMARSHAL           (TPMT_TK_CREATION_P_UNMARSHAL + 1)
133                 UNMARSHAL_DISPATCH(TPMT_TK_HASHCHECK),
134     #define TPMT_TK_VERIFIED_P_UNMARSHAL            (TPMT_TK_HASHCHECK_P_UNMARSHAL + 1)
135                 UNMARSHAL_DISPATCH(TPMT_TK_VERIFIED),
136     #define TPM_AT_P_UNMARSHAL                      (TPMT_TK_VERIFIED_P_UNMARSHAL + 1)
137                 UNMARSHAL_DISPATCH(TPM_AT),
138     #define TPM_CAP_P_UNMARSHAL                     (TPM_AT_P_UNMARSHAL + 1)
139                 UNMARSHAL_DISPATCH(TPM_CAP),
140     #define TPM_CLOCK_ADJUST_P_UNMARSHAL            (TPM_CAP_P_UNMARSHAL + 1)
141                 UNMARSHAL_DISPATCH(TPM_CLOCK_ADJUST),
142     #define TPM_EO_P_UNMARSHAL                      (TPM_CLOCK_ADJUST_P_UNMARSHAL + 1)
143                 UNMARSHAL_DISPATCH(TPM_EO),
144     #define TPM_SE_P_UNMARSHAL                      (TPM_EO_P_UNMARSHAL + 1)
145                 UNMARSHAL_DISPATCH(TPM_SE),
146     #define TPM_SU_P_UNMARSHAL                      (TPM_SE_P_UNMARSHAL + 1)
147                 UNMARSHAL_DISPATCH(TPM_SU),
148     #define UINT16_P_UNMARSHAL                      (TPM_SU_P_UNMARSHAL + 1)
149                 UNMARSHAL_DISPATCH(UINT16),
150     #define UINT32_P_UNMARSHAL                      (UINT16_P_UNMARSHAL + 1)
151                 UNMARSHAL_DISPATCH(UINT32),
152     #define UINT64_P_UNMARSHAL                      (UINT32_P_UNMARSHAL + 1)
153                 UNMARSHAL_DISPATCH(UINT64),
154     #define UINT8_P_UNMARSHAL                       (UINT64_P_UNMARSHAL + 1)
155                 UNMARSHAL_DISPATCH(UINT8),
156     // PARAMETER_FIRST_FLAG_TYPE is the first parameter to need a flag.
157     #define PARAMETER_FIRST_FLAG_TYPE               (UINT8_P_UNMARSHAL + 1)
158     #define TPM2B_PUBLIC_P_UNMARSHAL                (UINT8_P_UNMARSHAL + 1)
159                 UNMARSHAL_DISPATCH(TPM2B_PUBLIC),
160     #define TPMI_ALG_CIPHER_MODE_P_UNMARSHAL        (TPM2B_PUBLIC_P_UNMARSHAL + 1)
161                 UNMARSHAL_DISPATCH(TPMI_ALG_CIPHER_MODE),
162     #define TPMI_ALG_HASH_P_UNMARSHAL               \
163                 (TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + 1)
164                 UNMARSHAL_DISPATCH(TPMI_ALG_HASH),
165     #define TPMI_ALG_MAC_SCHEME_P_UNMARSHAL         (TPMI_ALG_HASH_P_UNMARSHAL + 1)
166                 UNMARSHAL_DISPATCH(TPMI_ALG_MAC_SCHEME),
167     #define TPMI_DH_PCR_P_UNMARSHAL                 \
168                 (TPMI_ALG_MAC_SCHEME_P_UNMARSHAL + 1)
169                 UNMARSHAL_DISPATCH(TPMI_DH_PCR),
170     #define TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL       (TPMI_DH_PCR_P_UNMARSHAL + 1)
171                 UNMARSHAL_DISPATCH(TPMI_ECC_KEY_EXCHANGE),
172     #define TPMI_RH_ENABLES_P_UNMARSHAL             \
173                 (TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL + 1)
174                 UNMARSHAL_DISPATCH(TPMI_RH_ENABLES),
175     #define TPMI_RH_HIERARCHY_P_UNMARSHAL           (TPMI_RH_ENABLES_P_UNMARSHAL + 1)
176                 UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY),
177     #define TPMT_RSA_DECRYPT_P_UNMARSHAL            (TPMI_RH_HIERARCHY_P_UNMARSHAL + 1)
178                 UNMARSHAL_DISPATCH(TPMT_RSA_DECRYPT),
```

```
179  #define TPMT_SIGNATURE_P_UNMARSHAL              (TPMT_RSA_DECRYPT_P_UNMARSHAL + 1)
180            UNMARSHAL_DISPATCH(TPMT_SIGNATURE),
181  #define TPMT_SIG_SCHEME_P_UNMARSHAL             (TPMT_SIGNATURE_P_UNMARSHAL + 1)
182            UNMARSHAL_DISPATCH(TPMT_SIG_SCHEME),
183  #define TPMT_SYM_DEF_P_UNMARSHAL                (TPMT_SIG_SCHEME_P_UNMARSHAL + 1)
184            UNMARSHAL_DISPATCH(TPMT_SYM_DEF),
185  #define TPMT_SYM_DEF_OBJECT_P_UNMARSHAL         (TPMT_SYM_DEF_P_UNMARSHAL + 1)
186            UNMARSHAL_DISPATCH(TPMT_SYM_DEF_OBJECT)
187  // PARAMETER_LAST_TYPE is the end of the command parameter list.
188  #define PARAMETER_LAST_TYPE                     (TPMT_SYM_DEF_OBJECT_P_UNMARSHAL)
189  };
```

The MarshalArray() contains the dispatch functions for the marshaling code. The defines in this array are used to make it easier to cross reference the marshaling values in the types array of each command

```
190  const _MARSHAL_T_ MarshalArray[] = {
191
192  #define UINT32_H_MARSHAL                        0
193             MARSHAL_DISPATCH(UINT32),
194  // RESPONSE_PARAMETER_FIRST_TYPE marks the end of the response handles.
195  #define RESPONSE_PARAMETER_FIRST_TYPE           (UINT32_H_MARSHAL + 1)
196  #define TPM2B_ATTEST_P_MARSHAL                  (UINT32_H_MARSHAL + 1)
197            MARSHAL_DISPATCH(TPM2B_ATTEST),
198  #define TPM2B_CREATION_DATA_P_MARSHAL           (TPM2B_ATTEST_P_MARSHAL + 1)
199            MARSHAL_DISPATCH(TPM2B_CREATION_DATA),
200  #define TPM2B_DATA_P_MARSHAL                    (TPM2B_CREATION_DATA_P_MARSHAL + 1)
201            MARSHAL_DISPATCH(TPM2B_DATA),
202  #define TPM2B_DIGEST_P_MARSHAL                  (TPM2B_DATA_P_MARSHAL + 1)
203            MARSHAL_DISPATCH(TPM2B_DIGEST),
204  #define TPM2B_ECC_POINT_P_MARSHAL               (TPM2B_DIGEST_P_MARSHAL + 1)
205            MARSHAL_DISPATCH(TPM2B_ECC_POINT),
206  #define TPM2B_ENCRYPTED_SECRET_P_MARSHAL        (TPM2B_ECC_POINT_P_MARSHAL + 1)
207            MARSHAL_DISPATCH(TPM2B_ENCRYPTED_SECRET),
208  #define TPM2B_ID_OBJECT_P_MARSHAL               \
209            (TPM2B_ENCRYPTED_SECRET_P_MARSHAL + 1)
210            MARSHAL_DISPATCH(TPM2B_ID_OBJECT),
211  #define TPM2B_IV_P_MARSHAL                      (TPM2B_ID_OBJECT_P_MARSHAL + 1)
212            MARSHAL_DISPATCH(TPM2B_IV),
213  #define TPM2B_MAX_BUFFER_P_MARSHAL              (TPM2B_IV_P_MARSHAL + 1)
214            MARSHAL_DISPATCH(TPM2B_MAX_BUFFER),
215  #define TPM2B_MAX_NV_BUFFER_P_MARSHAL           (TPM2B_MAX_BUFFER_P_MARSHAL + 1)
216            MARSHAL_DISPATCH(TPM2B_MAX_NV_BUFFER),
217  #define TPM2B_NAME_P_MARSHAL                    (TPM2B_MAX_NV_BUFFER_P_MARSHAL + 1)
218            MARSHAL_DISPATCH(TPM2B_NAME),
219  #define TPM2B_NV_PUBLIC_P_MARSHAL               (TPM2B_NAME_P_MARSHAL + 1)
220            MARSHAL_DISPATCH(TPM2B_NV_PUBLIC),
221  #define TPM2B_PRIVATE_P_MARSHAL                 (TPM2B_NV_PUBLIC_P_MARSHAL + 1)
222            MARSHAL_DISPATCH(TPM2B_PRIVATE),
223  #define TPM2B_PUBLIC_P_MARSHAL                  (TPM2B_PRIVATE_P_MARSHAL + 1)
224            MARSHAL_DISPATCH(TPM2B_PUBLIC),
225  #define TPM2B_PUBLIC_KEY_RSA_P_MARSHAL          (TPM2B_PUBLIC_P_MARSHAL + 1)
226            MARSHAL_DISPATCH(TPM2B_PUBLIC_KEY_RSA),
227  #define TPM2B_SENSITIVE_DATA_P_MARSHAL          (TPM2B_PUBLIC_KEY_RSA_P_MARSHAL + 1)
228            MARSHAL_DISPATCH(TPM2B_SENSITIVE_DATA),
229  #define TPM2B_TIMEOUT_P_MARSHAL                 (TPM2B_SENSITIVE_DATA_P_MARSHAL + 1)
230            MARSHAL_DISPATCH(TPM2B_TIMEOUT),
231  #define UINT8_P_MARSHAL                         (TPM2B_TIMEOUT_P_MARSHAL + 1)
232            MARSHAL_DISPATCH(UINT8),
233  #define TPML_AC_CAPABILITIES_P_MARSHAL          (UINT8_P_MARSHAL + 1)
234            MARSHAL_DISPATCH(TPML_AC_CAPABILITIES),
235  #define TPML_ALG_P_MARSHAL                      (TPML_AC_CAPABILITIES_P_MARSHAL + 1)
236            MARSHAL_DISPATCH(TPML_ALG),
237  #define TPML_DIGEST_P_MARSHAL                   (TPML_ALG_P_MARSHAL + 1)
238            MARSHAL_DISPATCH(TPML_DIGEST),
239  #define TPML_DIGEST_VALUES_P_MARSHAL            (TPML_DIGEST_P_MARSHAL + 1)
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
240             MARSHAL_DISPATCH(TPML_DIGEST_VALUES),
241     #define TPML_PCR_SELECTION_P_MARSHAL            (TPML_DIGEST_VALUES_P_MARSHAL + 1)
242             MARSHAL_DISPATCH(TPML_PCR_SELECTION),
243     #define TPMS_AC_OUTPUT_P_MARSHAL                (TPML_PCR_SELECTION_P_MARSHAL + 1)
244             MARSHAL_DISPATCH(TPMS_AC_OUTPUT),
245     #define TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL     (TPMS_AC_OUTPUT_P_MARSHAL + 1)
246             MARSHAL_DISPATCH(TPMS_ALGORITHM_DETAIL_ECC),
247     #define TPMS_CAPABILITY_DATA_P_MARSHAL          \
248              (TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL + 1)
249             MARSHAL_DISPATCH(TPMS_CAPABILITY_DATA),
250     #define TPMS_CONTEXT_P_MARSHAL                  (TPMS_CAPABILITY_DATA_P_MARSHAL + 1)
251             MARSHAL_DISPATCH(TPMS_CONTEXT),
252     #define TPMS_TIME_INFO_P_MARSHAL                (TPMS_CONTEXT_P_MARSHAL + 1)
253             MARSHAL_DISPATCH(TPMS_TIME_INFO),
254     #define TPMT_HA_P_MARSHAL                       (TPMS_TIME_INFO_P_MARSHAL + 1)
255             MARSHAL_DISPATCH(TPMT_HA),
256     #define TPMT_SIGNATURE_P_MARSHAL                (TPMT_HA_P_MARSHAL + 1)
257             MARSHAL_DISPATCH(TPMT_SIGNATURE),
258     #define TPMT_TK_AUTH_P_MARSHAL                  (TPMT_SIGNATURE_P_MARSHAL + 1)
259             MARSHAL_DISPATCH(TPMT_TK_AUTH),
260     #define TPMT_TK_CREATION_P_MARSHAL              (TPMT_TK_AUTH_P_MARSHAL + 1)
261             MARSHAL_DISPATCH(TPMT_TK_CREATION),
262     #define TPMT_TK_HASHCHECK_P_MARSHAL             (TPMT_TK_CREATION_P_MARSHAL + 1)
263             MARSHAL_DISPATCH(TPMT_TK_HASHCHECK),
264     #define TPMT_TK_VERIFIED_P_MARSHAL              (TPMT_TK_HASHCHECK_P_MARSHAL + 1)
265             MARSHAL_DISPATCH(TPMT_TK_VERIFIED),
266     #define UINT32_P_MARSHAL                        (TPMT_TK_VERIFIED_P_MARSHAL + 1)
267             MARSHAL_DISPATCH(UINT32),
268     #define UINT16_P_MARSHAL                        (UINT32_P_MARSHAL + 1)
269             MARSHAL_DISPATCH(UINT16)
270     // RESPONSE_PARAMETER_LAST_TYPE is the end of the response parameter list.
271     #define RESPONSE_PARAMETER_LAST_TYPE            (UINT16_P_MARSHAL)
272     };
```

This list of aliases allows the types in the _COMMAND_DESCRIPTOR_T to match the types in the command/response templates of part 3.

```
273     #define INT32_P_UNMARSHAL                       UINT32_P_UNMARSHAL
274     #define TPM2B_AUTH_P_UNMARSHAL                  TPM2B_DIGEST_P_UNMARSHAL
275     #define TPM2B_NONCE_P_UNMARSHAL                 TPM2B_DIGEST_P_UNMARSHAL
276     #define TPM2B_OPERAND_P_UNMARSHAL               TPM2B_DIGEST_P_UNMARSHAL
277     #define TPMA_LOCALITY_P_UNMARSHAL               UINT8_P_UNMARSHAL
278     #define TPM_CC_P_UNMARSHAL                      UINT32_P_UNMARSHAL
279     #define TPMI_DH_CONTEXT_H_MARSHAL               UINT32_H_MARSHAL
280     #define TPMI_DH_OBJECT_H_MARSHAL                UINT32_H_MARSHAL
281     #define TPMI_SH_AUTH_SESSION_H_MARSHAL          UINT32_H_MARSHAL
282     #define TPM_HANDLE_H_MARSHAL                    UINT32_H_MARSHAL
283     #define TPM2B_NONCE_P_MARSHAL                   TPM2B_DIGEST_P_MARSHAL
284     #define TPMI_YES_NO_P_MARSHAL                   UINT8_P_MARSHAL
285     #define TPM_RC_P_MARSHAL                        UINT32_P_MARSHAL
286     #if CC_Startup
287     #include "Startup_fp.h"
288     typedef TPM_RC  (Startup_Entry)(
289         Startup_In                  *in
290     );
291     typedef const struct {
292         Startup_Entry           *entry;
293         UINT16                  inSize;
294         UINT16                  outSize;
295         UINT16                  offsetOfTypes;
296         BYTE                    types[3];
297     } Startup_COMMAND_DESCRIPTOR_t;
298     Startup_COMMAND_DESCRIPTOR_t _StartupData = {
299         /* entry         */     &TPM2_Startup,
300         /* inSize        */     (UINT16)(sizeof(Startup_In)),
```

```
301         /* outSize       */      0,
302         /* offsetOfTypes */      offsetof(Startup_COMMAND_DESCRIPTOR_t, types),
303         /* offsets       */      // No parameter offsets;
304         /* types         */      {TPM_SU_P_UNMARSHAL,
305                                   END_OF_LIST,
306                                   END_OF_LIST}
307     };
308     #define _StartupDataAddress (&_StartupData)
309     #else
310     #define _StartupDataAddress 0
311     #endif // CC_Startup
312     #if CC_Shutdown
313     #include "Shutdown_fp.h"
314     typedef TPM_RC  (Shutdown_Entry)(
315         Shutdown_In                 *in
316     );
317     typedef const struct {
318         Shutdown_Entry          *entry;
319         UINT16                  inSize;
320         UINT16                  outSize;
321         UINT16                  offsetOfTypes;
322         BYTE                    types[3];
323     } Shutdown_COMMAND_DESCRIPTOR_t;
324     Shutdown_COMMAND_DESCRIPTOR_t _ShutdownData = {
325         /* entry         */      &TPM2_Shutdown,
326         /* inSize        */      (UINT16)(sizeof(Shutdown_In)),
327         /* outSize       */      0,
328         /* offsetOfTypes */      offsetof(Shutdown_COMMAND_DESCRIPTOR_t, types),
329         /* offsets       */      // No parameter offsets;
330         /* types         */      {TPM_SU_P_UNMARSHAL,
331                                   END_OF_LIST,
332                                   END_OF_LIST}
333     };
334     #define _ShutdownDataAddress (&_ShutdownData)
335     #else
336     #define _ShutdownDataAddress 0
337     #endif // CC_Shutdown
338     #if CC_SelfTest
339     #include "SelfTest_fp.h"
340     typedef TPM_RC  (SelfTest_Entry)(
341         SelfTest_In                 *in
342     );
343     typedef const struct {
344         SelfTest_Entry          *entry;
345         UINT16                  inSize;
346         UINT16                  outSize;
347         UINT16                  offsetOfTypes;
348         BYTE                    types[3];
349     } SelfTest_COMMAND_DESCRIPTOR_t;
350     SelfTest_COMMAND_DESCRIPTOR_t _SelfTestData = {
351         /* entry         */      &TPM2_SelfTest,
352         /* inSize        */      (UINT16)(sizeof(SelfTest_In)),
353         /* outSize       */      0,
354         /* offsetOfTypes */      offsetof(SelfTest_COMMAND_DESCRIPTOR_t, types),
355         /* offsets       */      // No parameter offsets;
356         /* types         */      {TPMI_YES_NO_P_UNMARSHAL,
357                                   END_OF_LIST,
358                                   END_OF_LIST}
359     };
360     #define _SelfTestDataAddress (&_SelfTestData)
361     #else
362     #define _SelfTestDataAddress 0
363     #endif // CC_SelfTest
364     #if CC_IncrementalSelfTest
365     #include "IncrementalSelfTest_fp.h"
366     typedef TPM_RC  (IncrementalSelfTest_Entry)(
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
367         IncrementalSelfTest_In          *in,
368         IncrementalSelfTest_Out         *out
369     );
370     typedef const struct {
371         IncrementalSelfTest_Entry   *entry;
372         UINT16                      inSize;
373         UINT16                      outSize;
374         UINT16                      offsetOfTypes;
375         BYTE                        types[4];
376     } IncrementalSelfTest_COMMAND_DESCRIPTOR_t;
377     IncrementalSelfTest_COMMAND_DESCRIPTOR_t _IncrementalSelfTestData = {
378         /* entry         */         &TPM2_IncrementalSelfTest,
379         /* inSize        */         (UINT16)(sizeof(IncrementalSelfTest_In)),
380         /* outSize       */         (UINT16)(sizeof(IncrementalSelfTest_Out)),
381         /* offsetOfTypes */         offsetof(IncrementalSelfTest_COMMAND_DESCRIPTOR_t,
    types),
382         /* offsets       */         // No parameter offsets;
383         /* types         */         {TPML_ALG_P_UNMARSHAL,
384                                      END_OF_LIST,
385                                      TPML_ALG_P_MARSHAL,
386                                      END_OF_LIST}
387     };
388     #define _IncrementalSelfTestDataAddress (&_IncrementalSelfTestData)
389     #else
390     #define _IncrementalSelfTestDataAddress 0
391     #endif // CC_IncrementalSelfTest
392     #if CC_GetTestResult
393     #include "GetTestResult_fp.h"
394     typedef TPM_RC  (GetTestResult_Entry)(
395         GetTestResult_Out           *out
396     );
397     typedef const struct {
398         GetTestResult_Entry     *entry;
399         UINT16                  inSize;
400         UINT16                  outSize;
401         UINT16                  offsetOfTypes;
402         UINT16                  paramOffsets[1];
403         BYTE                    types[4];
404     } GetTestResult_COMMAND_DESCRIPTOR_t;
405     GetTestResult_COMMAND_DESCRIPTOR_t _GetTestResultData = {
406         /* entry         */     &TPM2_GetTestResult,
407         /* inSize        */     0,
408         /* outSize       */     (UINT16)(sizeof(GetTestResult_Out)),
409         /* offsetOfTypes */     offsetof(GetTestResult_COMMAND_DESCRIPTOR_t, types),
410         /* offsets       */     {(UINT16)(offsetof(GetTestResult_Out, testResult))},
411         /* types         */     {END_OF_LIST,
412                                  TPM2B_MAX_BUFFER_P_MARSHAL,
413                                  TPM_RC_P_MARSHAL,
414                                  END_OF_LIST}
415     };
416     #define _GetTestResultDataAddress (&_GetTestResultData)
417     #else
418     #define _GetTestResultDataAddress 0
419     #endif // CC_GetTestResult
420     #if CC_StartAuthSession
421     #include "StartAuthSession_fp.h"
422     typedef TPM_RC  (StartAuthSession_Entry)(
423         StartAuthSession_In         *in,
424         StartAuthSession_Out        *out
425     );
426     typedef const struct {
427         StartAuthSession_Entry  *entry;
428         UINT16                  inSize;
429         UINT16                  outSize;
430         UINT16                  offsetOfTypes;
431         UINT16                  paramOffsets[7];
```

```
432     BYTE                    types[11];
433 } StartAuthSession_COMMAND_DESCRIPTOR_t;
434 StartAuthSession_COMMAND_DESCRIPTOR_t _StartAuthSessionData = {
435     /* entry         */     &TPM2_StartAuthSession,
436     /* inSize        */     (UINT16)(sizeof(StartAuthSession_In)),
437     /* outSize       */     (UINT16)(sizeof(StartAuthSession_Out)),
438     /* offsetOfTypes */     offsetof(StartAuthSession_COMMAND_DESCRIPTOR_t, types),
439     /* offsets       */     {(UINT16)(offsetof(StartAuthSession_In, bind)),
440                              (UINT16)(offsetof(StartAuthSession_In, nonceCaller)),
441                              (UINT16)(offsetof(StartAuthSession_In, encryptedSalt)),
442                              (UINT16)(offsetof(StartAuthSession_In, sessionType)),
443                              (UINT16)(offsetof(StartAuthSession_In, symmetric)),
444                              (UINT16)(offsetof(StartAuthSession_In, authHash)),
445                              (UINT16)(offsetof(StartAuthSession_Out, nonceTPM))},
446     /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
447                              TPMI_DH_ENTITY_H_UNMARSHAL + ADD_FLAG,
448                              TPM2B_NONCE_P_UNMARSHAL,
449                              TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
450                              TPM_SE_P_UNMARSHAL,
451                              TPMT_SYM_DEF_P_UNMARSHAL + ADD_FLAG,
452                              TPMI_ALG_HASH_P_UNMARSHAL,
453                              END_OF_LIST,
454                              TPMI_SH_AUTH_SESSION_H_MARSHAL,
455                              TPM2B_NONCE_P_MARSHAL,
456                              END_OF_LIST}
457 };
458 #define _StartAuthSessionDataAddress (&_StartAuthSessionData)
459 #else
460 #define _StartAuthSessionDataAddress 0
461 #endif // CC_StartAuthSession
462 #if CC_PolicyRestart
463 #include "PolicyRestart_fp.h"
464 typedef TPM_RC  (PolicyRestart_Entry)(
465     PolicyRestart_In            *in
466 );
467 typedef const struct {
468     PolicyRestart_Entry     *entry;
469     UINT16                  inSize;
470     UINT16                  outSize;
471     UINT16                  offsetOfTypes;
472     BYTE                    types[3];
473 } PolicyRestart_COMMAND_DESCRIPTOR_t;
474 PolicyRestart_COMMAND_DESCRIPTOR_t _PolicyRestartData = {
475     /* entry         */     &TPM2_PolicyRestart,
476     /* inSize        */     (UINT16)(sizeof(PolicyRestart_In)),
477     /* outSize       */     0,
478     /* offsetOfTypes */     offsetof(PolicyRestart_COMMAND_DESCRIPTOR_t, types),
479     /* offsets       */     // No parameter offsets;
480     /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
481                              END_OF_LIST,
482                              END_OF_LIST}
483 };
484 #define _PolicyRestartDataAddress (&_PolicyRestartData)
485 #else
486 #define _PolicyRestartDataAddress 0
487 #endif // CC_PolicyRestart
488 #if CC_Create
489 #include "Create_fp.h"
490 typedef TPM_RC  (Create_Entry)(
491     Create_In                   *in,
492     Create_Out                  *out
493 );
494 typedef const struct {
495     Create_Entry            *entry;
496     UINT16                  inSize;
497     UINT16                  outSize;
```


 Copyright © TCG 2006-2020

```
498         UINT16                  offsetOfTypes;
499         UINT16                  paramOffsets[8];
500         BYTE                    types[12];
501     } Create_COMMAND_DESCRIPTOR_t;
502     Create_COMMAND_DESCRIPTOR_t _CreateData = {
503         /* entry         */     &TPM2_Create,
504         /* inSize        */     (UINT16)(sizeof(Create_In)),
505         /* outSize       */     (UINT16)(sizeof(Create_Out)),
506         /* offsetOfTypes */     offsetof(Create_COMMAND_DESCRIPTOR_t, types),
507         /* offsets       */     {(UINT16)(offsetof(Create_In, inSensitive)),
508                                  (UINT16)(offsetof(Create_In, inPublic)),
509                                  (UINT16)(offsetof(Create_In, outsideInfo)),
510                                  (UINT16)(offsetof(Create_In, creationPCR)),
511                                  (UINT16)(offsetof(Create_Out, outPublic)),
512                                  (UINT16)(offsetof(Create_Out, creationData)),
513                                  (UINT16)(offsetof(Create_Out, creationHash)),
514                                  (UINT16)(offsetof(Create_Out, creationTicket))},
515         /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
516                                  TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
517                                  TPM2B_PUBLIC_P_UNMARSHAL,
518                                  TPM2B_DATA_P_UNMARSHAL,
519                                  TPML_PCR_SELECTION_P_UNMARSHAL,
520                                  END_OF_LIST,
521                                  TPM2B_PRIVATE_P_MARSHAL,
522                                  TPM2B_PUBLIC_P_MARSHAL,
523                                  TPM2B_CREATION_DATA_P_MARSHAL,
524                                  TPM2B_DIGEST_P_MARSHAL,
525                                  TPMT_TK_CREATION_P_MARSHAL,
526                                  END_OF_LIST}
527     };
528     #define _CreateDataAddress (&_CreateData)
529     #else
530     #define _CreateDataAddress 0
531     #endif // CC_Create
532     #if CC_Load
533     #include "Load_fp.h"
534     typedef TPM_RC  (Load_Entry)(
535         Load_In                     *in,
536         Load_Out                    *out
537     );
538     typedef const struct {
539         Load_Entry              *entry;
540         UINT16                  inSize;
541         UINT16                  outSize;
542         UINT16                  offsetOfTypes;
543         UINT16                  paramOffsets[3];
544         BYTE                    types[7];
545     } Load_COMMAND_DESCRIPTOR_t;
546     Load_COMMAND_DESCRIPTOR_t _LoadData = {
547         /* entry         */     &TPM2_Load,
548         /* inSize        */     (UINT16)(sizeof(Load_In)),
549         /* outSize       */     (UINT16)(sizeof(Load_Out)),
550         /* offsetOfTypes */     offsetof(Load_COMMAND_DESCRIPTOR_t, types),
551         /* offsets       */     {(UINT16)(offsetof(Load_In, inPrivate)),
552                                  (UINT16)(offsetof(Load_In, inPublic)),
553                                  (UINT16)(offsetof(Load_Out, name))},
554         /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
555                                  TPM2B_PRIVATE_P_UNMARSHAL,
556                                  TPM2B_PUBLIC_P_UNMARSHAL,
557                                  END_OF_LIST,
558                                  TPM_HANDLE_H_MARSHAL,
559                                  TPM2B_NAME_P_MARSHAL,
560                                  END_OF_LIST}
561     };
562     #define _LoadDataAddress (&_LoadData)
563     #else
```

 Copyright © TCG 2006-2020

```
564 #define _LoadDataAddress 0
565 #endif // CC_Load
566 #if CC_LoadExternal
567 #include "LoadExternal_fp.h"
568 typedef TPM_RC  (LoadExternal_Entry)(
569     LoadExternal_In              *in,
570     LoadExternal_Out             *out
571 );
572 typedef const struct {
573     LoadExternal_Entry      *entry;
574     UINT16                  inSize;
575     UINT16                  outSize;
576     UINT16                  offsetOfTypes;
577     UINT16                  paramOffsets[3];
578     BYTE                    types[7];
579 } LoadExternal_COMMAND_DESCRIPTOR_t;
580 LoadExternal_COMMAND_DESCRIPTOR_t _LoadExternalData = {
581     /* entry         */     &TPM2_LoadExternal,
582     /* inSize        */     (UINT16)(sizeof(LoadExternal_In)),
583     /* outSize       */     (UINT16)(sizeof(LoadExternal_Out)),
584     /* offsetOfTypes */     offsetof(LoadExternal_COMMAND_DESCRIPTOR_t, types),
585     /* offsets       */     {(UINT16)(offsetof(LoadExternal_In, inPublic)),
586                              (UINT16)(offsetof(LoadExternal_In, hierarchy)),
587                              (UINT16)(offsetof(LoadExternal_Out, name))},
588     /* types         */     {TPM2B_SENSITIVE_P_UNMARSHAL,
589                              TPM2B_PUBLIC_P_UNMARSHAL + ADD_FLAG,
590                              TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
591                              END_OF_LIST,
592                              TPM_HANDLE_H_MARSHAL,
593                              TPM2B_NAME_P_MARSHAL,
594                              END_OF_LIST}
595 };
596 #define _LoadExternalDataAddress (&_LoadExternalData)
597 #else
598 #define _LoadExternalDataAddress 0
599 #endif // CC_LoadExternal
600 #if CC_ReadPublic
601 #include "ReadPublic_fp.h"
602 typedef TPM_RC  (ReadPublic_Entry)(
603     ReadPublic_In                *in,
604     ReadPublic_Out               *out
605 );
606 typedef const struct {
607     ReadPublic_Entry        *entry;
608     UINT16                  inSize;
609     UINT16                  outSize;
610     UINT16                  offsetOfTypes;
611     UINT16                  paramOffsets[2];
612     BYTE                    types[6];
613 } ReadPublic_COMMAND_DESCRIPTOR_t;
614 ReadPublic_COMMAND_DESCRIPTOR_t _ReadPublicData = {
615     /* entry         */     &TPM2_ReadPublic,
616     /* inSize        */     (UINT16)(sizeof(ReadPublic_In)),
617     /* outSize       */     (UINT16)(sizeof(ReadPublic_Out)),
618     /* offsetOfTypes */     offsetof(ReadPublic_COMMAND_DESCRIPTOR_t, types),
619     /* offsets       */     {(UINT16)(offsetof(ReadPublic_Out, name)),
620                              (UINT16)(offsetof(ReadPublic_Out, qualifiedName))},
621     /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
622                              END_OF_LIST,
623                              TPM2B_PUBLIC_P_MARSHAL,
624                              TPM2B_NAME_P_MARSHAL,
625                              TPM2B_NAME_P_MARSHAL,
626                              END_OF_LIST}
627 };
628 #define _ReadPublicDataAddress (&_ReadPublicData)
629 #else
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
630 #define _ReadPublicDataAddress 0
631 #endif // CC_ReadPublic
632 #if CC_ActivateCredential
633 #include "ActivateCredential_fp.h"
634 typedef TPM_RC  (ActivateCredential_Entry)(
635     ActivateCredential_In            *in,
636     ActivateCredential_Out           *out
637 );
638 typedef const struct {
639     ActivateCredential_Entry    *entry;
640     UINT16                      inSize;
641     UINT16                      outSize;
642     UINT16                      offsetOfTypes;
643     UINT16                      paramOffsets[3];
644     BYTE                        types[7];
645 } ActivateCredential_COMMAND_DESCRIPTOR_t;
646 ActivateCredential_COMMAND_DESCRIPTOR_t _ActivateCredentialData = {
647     /* entry         */          &TPM2_ActivateCredential,
648     /* inSize        */          (UINT16)(sizeof(ActivateCredential_In)),
649     /* outSize       */          (UINT16)(sizeof(ActivateCredential_Out)),
650     /* offsetOfTypes */          offsetof(ActivateCredential_COMMAND_DESCRIPTOR_t,
types),
651     /* offsets       */          {(UINT16)(offsetof(ActivateCredential_In, keyHandle)),
652                                   (UINT16)(offsetof(ActivateCredential_In,
credentialBlob)),
653                                   (UINT16)(offsetof(ActivateCredential_In, secret))},
654     /* types         */          {TPMI_DH_OBJECT_H_UNMARSHAL,
655                                   TPMI_DH_OBJECT_H_UNMARSHAL,
656                                   TPM2B_ID_OBJECT_P_UNMARSHAL,
657                                   TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
658                                   END_OF_LIST,
659                                   TPM2B_DIGEST_P_MARSHAL,
660                                   END_OF_LIST}
661 };
662 #define _ActivateCredentialDataAddress (&_ActivateCredentialData)
663 #else
664 #define _ActivateCredentialDataAddress 0
665 #endif // CC_ActivateCredential
666 #if CC_MakeCredential
667 #include "MakeCredential_fp.h"
668 typedef TPM_RC  (MakeCredential_Entry)(
669     MakeCredential_In            *in,
670     MakeCredential_Out           *out
671 );
672 typedef const struct {
673     MakeCredential_Entry    *entry;
674     UINT16                  inSize;
675     UINT16                  outSize;
676     UINT16                  offsetOfTypes;
677     UINT16                  paramOffsets[3];
678     BYTE                    types[7];
679 } MakeCredential_COMMAND_DESCRIPTOR_t;
680 MakeCredential_COMMAND_DESCRIPTOR_t _MakeCredentialData = {
681     /* entry         */      &TPM2_MakeCredential,
682     /* inSize        */      (UINT16)(sizeof(MakeCredential_In)),
683     /* outSize       */      (UINT16)(sizeof(MakeCredential_Out)),
684     /* offsetOfTypes */      offsetof(MakeCredential_COMMAND_DESCRIPTOR_t, types),
685     /* offsets       */      {(UINT16)(offsetof(MakeCredential_In, credential)),
686                               (UINT16)(offsetof(MakeCredential_In, objectName)),
687                               (UINT16)(offsetof(MakeCredential_Out, secret))},
688     /* types         */      {TPMI_DH_OBJECT_H_UNMARSHAL,
689                               TPM2B_DIGEST_P_UNMARSHAL,
690                               TPM2B_NAME_P_UNMARSHAL,
691                               END_OF_LIST,
692                               TPM2B_ID_OBJECT_P_MARSHAL,
693                               TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
```

```
694                                 END_OF_LIST}
695  };
696  #define _MakeCredentialDataAddress (&_MakeCredentialData)
697  #else
698  #define _MakeCredentialDataAddress 0
699  #endif // CC_MakeCredential
700  #if CC_Unseal
701  #include "Unseal_fp.h"
702  typedef TPM_RC  (Unseal_Entry)(
703      Unseal_In                    *in,
704      Unseal_Out                   *out
705  );
706  typedef const struct {
707      Unseal_Entry             *entry;
708      UINT16                   inSize;
709      UINT16                   outSize;
710      UINT16                   offsetOfTypes;
711      BYTE                     types[4];
712  } Unseal_COMMAND_DESCRIPTOR_t;
713  Unseal_COMMAND_DESCRIPTOR_t _UnsealData = {
714      /* entry          */     &TPM2_Unseal,
715      /* inSize         */     (UINT16)(sizeof(Unseal_In)),
716      /* outSize        */     (UINT16)(sizeof(Unseal_Out)),
717      /* offsetOfTypes */      offsetof(Unseal_COMMAND_DESCRIPTOR_t, types),
718      /* offsets        */     // No parameter offsets;
719      /* types          */     {TPMI_DH_OBJECT_H_UNMARSHAL,
720                               END_OF_LIST,
721                               TPM2B_SENSITIVE_DATA_P_MARSHAL,
722                               END_OF_LIST}
723  };
724  #define _UnsealDataAddress (&_UnsealData)
725  #else
726  #define _UnsealDataAddress 0
727  #endif // CC_Unseal
728  #if CC_ObjectChangeAuth
729  #include "ObjectChangeAuth_fp.h"
730  typedef TPM_RC  (ObjectChangeAuth_Entry)(
731      ObjectChangeAuth_In          *in,
732      ObjectChangeAuth_Out         *out
733  );
734  typedef const struct {
735      ObjectChangeAuth_Entry   *entry;
736      UINT16                   inSize;
737      UINT16                   outSize;
738      UINT16                   offsetOfTypes;
739      UINT16                   paramOffsets[2];
740      BYTE                     types[6];
741  } ObjectChangeAuth_COMMAND_DESCRIPTOR_t;
742  ObjectChangeAuth_COMMAND_DESCRIPTOR_t _ObjectChangeAuthData = {
743      /* entry          */     &TPM2_ObjectChangeAuth,
744      /* inSize         */     (UINT16)(sizeof(ObjectChangeAuth_In)),
745      /* outSize        */     (UINT16)(sizeof(ObjectChangeAuth_Out)),
746      /* offsetOfTypes */      offsetof(ObjectChangeAuth_COMMAND_DESCRIPTOR_t, types),
747      /* offsets        */     {(UINT16)(offsetof(ObjectChangeAuth_In, parentHandle)),
748                               (UINT16)(offsetof(ObjectChangeAuth_In, newAuth))},
749      /* types          */     {TPMI_DH_OBJECT_H_UNMARSHAL,
750                               TPMI_DH_OBJECT_H_UNMARSHAL,
751                               TPM2B_AUTH_P_UNMARSHAL,
752                               END_OF_LIST,
753                               TPM2B_PRIVATE_P_MARSHAL,
754                               END_OF_LIST}
755  };
756  #define _ObjectChangeAuthDataAddress (&_ObjectChangeAuthData)
757  #else
758  #define _ObjectChangeAuthDataAddress 0
759  #endif // CC_ObjectChangeAuth
```

```
760 #if CC_CreateLoaded
761 #include "CreateLoaded_fp.h"
762 typedef TPM_RC  (CreateLoaded_Entry)(
763     CreateLoaded_In             *in,
764     CreateLoaded_Out            *out
765 );
766 typedef const struct {
767     CreateLoaded_Entry      *entry;
768     UINT16                  inSize;
769     UINT16                  outSize;
770     UINT16                  offsetOfTypes;
771     UINT16                  paramOffsets[5];
772     BYTE                    types[9];
773 } CreateLoaded_COMMAND_DESCRIPTOR_t;
774 CreateLoaded_COMMAND_DESCRIPTOR_t _CreateLoadedData = {
775     /* entry         */     &TPM2_CreateLoaded,
776     /* inSize        */     (UINT16)(sizeof(CreateLoaded_In)),
777     /* outSize       */     (UINT16)(sizeof(CreateLoaded_Out)),
778     /* offsetOfTypes */     offsetof(CreateLoaded_COMMAND_DESCRIPTOR_t, types),
779     /* offsets       */     {(UINT16)(offsetof(CreateLoaded_In, inSensitive)),
780                              (UINT16)(offsetof(CreateLoaded_In, inPublic)),
781                              (UINT16)(offsetof(CreateLoaded_Out, outPrivate)),
782                              (UINT16)(offsetof(CreateLoaded_Out, outPublic)),
783                              (UINT16)(offsetof(CreateLoaded_Out, name))},
784     /* types         */     {TPMI_DH_PARENT_H_UNMARSHAL + ADD_FLAG,
785                              TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
786                              TPM2B_TEMPLATE_P_UNMARSHAL,
787                              END_OF_LIST,
788                              TPM_HANDLE_H_MARSHAL,
789                              TPM2B_PRIVATE_P_MARSHAL,
790                              TPM2B_PUBLIC_P_MARSHAL,
791                              TPM2B_NAME_P_MARSHAL,
792                              END_OF_LIST}
793 };
794 #define _CreateLoadedDataAddress (&_CreateLoadedData)
795 #else
796 #define _CreateLoadedDataAddress 0
797 #endif // CC_CreateLoaded
798 #if CC_Duplicate
799 #include "Duplicate_fp.h"
800 typedef TPM_RC  (Duplicate_Entry)(
801     Duplicate_In                *in,
802     Duplicate_Out               *out
803 );
804 typedef const struct {
805     Duplicate_Entry         *entry;
806     UINT16                  inSize;
807     UINT16                  outSize;
808     UINT16                  offsetOfTypes;
809     UINT16                  paramOffsets[5];
810     BYTE                    types[9];
811 } Duplicate_COMMAND_DESCRIPTOR_t;
812 Duplicate_COMMAND_DESCRIPTOR_t _DuplicateData = {
813     /* entry         */     &TPM2_Duplicate,
814     /* inSize        */     (UINT16)(sizeof(Duplicate_In)),
815     /* outSize       */     (UINT16)(sizeof(Duplicate_Out)),
816     /* offsetOfTypes */     offsetof(Duplicate_COMMAND_DESCRIPTOR_t, types),
817     /* offsets       */     {(UINT16)(offsetof(Duplicate_In, newParentHandle)),
818                              (UINT16)(offsetof(Duplicate_In, encryptionKeyIn)),
819                              (UINT16)(offsetof(Duplicate_In, symmetricAlg)),
820                              (UINT16)(offsetof(Duplicate_Out, duplicate)),
821                              (UINT16)(offsetof(Duplicate_Out, outSymSeed))},
822     /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
823                              TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
824                              TPM2B_DATA_P_UNMARSHAL,
825                              TPMT_SYM_DEF_OBJECT_P_UNMARSHAL + ADD_FLAG,
```


 Copyright © TCG 2006-2020

```
826                             END_OF_LIST,
827                             TPM2B_DATA_P_MARSHAL,
828                             TPM2B_PRIVATE_P_MARSHAL,
829                             TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
830                             END_OF_LIST}
831 };
832 #define _DuplicateDataAddress (&_DuplicateData)
833 #else
834 #define _DuplicateDataAddress 0
835 #endif // CC_Duplicate
836 #if CC_Rewrap
837 #include "Rewrap_fp.h"
838 typedef TPM_RC  (Rewrap_Entry)(
839     Rewrap_In                      *in,
840     Rewrap_Out                     *out
841 );
842 typedef const struct {
843     Rewrap_Entry               *entry;
844     UINT16                     inSize;
845     UINT16                     outSize;
846     UINT16                     offsetOfTypes;
847     UINT16                     paramOffsets[5];
848     BYTE                       types[9];
849 } Rewrap_COMMAND_DESCRIPTOR_t;
850 Rewrap_COMMAND_DESCRIPTOR_t _RewrapData = {
851     /* entry         */      &TPM2_Rewrap,
852     /* inSize        */      (UINT16)(sizeof(Rewrap_In)),
853     /* outSize       */      (UINT16)(sizeof(Rewrap_Out)),
854     /* offsetOfTypes */      offsetof(Rewrap_COMMAND_DESCRIPTOR_t, types),
855     /* offsets       */      {(UINT16)(offsetof(Rewrap_In, newParent)),
856                               (UINT16)(offsetof(Rewrap_In, inDuplicate)),
857                               (UINT16)(offsetof(Rewrap_In, name)),
858                               (UINT16)(offsetof(Rewrap_In, inSymSeed)),
859                               (UINT16)(offsetof(Rewrap_Out, outSymSeed))},
860     /* types         */      {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
861                              TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
862                              TPM2B_PRIVATE_P_UNMARSHAL,
863                              TPM2B_NAME_P_UNMARSHAL,
864                              TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
865                              END_OF_LIST,
866                              TPM2B_PRIVATE_P_MARSHAL,
867                              TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
868                              END_OF_LIST}
869 };
870 #define _RewrapDataAddress (&_RewrapData)
871 #else
872 #define _RewrapDataAddress 0
873 #endif // CC_Rewrap
874 #if CC_Import
875 #include "Import_fp.h"
876 typedef TPM_RC  (Import_Entry)(
877     Import_In                      *in,
878     Import_Out                     *out
879 );
880 typedef const struct {
881     Import_Entry               *entry;
882     UINT16                     inSize;
883     UINT16                     outSize;
884     UINT16                     offsetOfTypes;
885     UINT16                     paramOffsets[5];
886     BYTE                       types[9];
887 } Import_COMMAND_DESCRIPTOR_t;
888 Import_COMMAND_DESCRIPTOR_t _ImportData = {
889     /* entry         */      &TPM2_Import,
890     /* inSize        */      (UINT16)(sizeof(Import_In)),
891     /* outSize       */      (UINT16)(sizeof(Import_Out)),
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
892     /* offsetOfTypes */     offsetof(Import_COMMAND_DESCRIPTOR_t, types),
893     /* offsets       */     {(UINT16)(offsetof(Import_In, encryptionKey)),
894                              (UINT16)(offsetof(Import_In, objectPublic)),
895                              (UINT16)(offsetof(Import_In, duplicate)),
896                              (UINT16)(offsetof(Import_In, inSymSeed)),
897                              (UINT16)(offsetof(Import_In, symmetricAlg))},
898     /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
899                              TPM2B_DATA_P_UNMARSHAL,
900                              TPM2B_PUBLIC_P_UNMARSHAL,
901                              TPM2B_PRIVATE_P_UNMARSHAL,
902                              TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
903                              TPMT_SYM_DEF_OBJECT_P_UNMARSHAL + ADD_FLAG,
904                              END_OF_LIST,
905                              TPM2B_PRIVATE_P_MARSHAL,
906                              END_OF_LIST}
907 };
908 #define _ImportDataAddress (&_ImportData)
909 #else
910 #define _ImportDataAddress 0
911 #endif // CC_Import
912 #if CC_RSA_Encrypt
913 #include "RSA_Encrypt_fp.h"
914 typedef TPM_RC  (RSA_Encrypt_Entry)(
915     RSA_Encrypt_In              *in,
916     RSA_Encrypt_Out             *out
917 );
918 typedef const struct {
919     RSA_Encrypt_Entry       *entry;
920     UINT16                  inSize;
921     UINT16                  outSize;
922     UINT16                  offsetOfTypes;
923     UINT16                  paramOffsets[3];
924     BYTE                    types[7];
925 } RSA_Encrypt_COMMAND_DESCRIPTOR_t;
926 RSA_Encrypt_COMMAND_DESCRIPTOR_t _RSA_EncryptData = {
927     /* entry         */     &TPM2_RSA_Encrypt,
928     /* inSize        */     (UINT16)(sizeof(RSA_Encrypt_In)),
929     /* outSize       */     (UINT16)(sizeof(RSA_Encrypt_Out)),
930     /* offsetOfTypes */     offsetof(RSA_Encrypt_COMMAND_DESCRIPTOR_t, types),
931     /* offsets       */     {(UINT16)(offsetof(RSA_Encrypt_In, message)),
932                              (UINT16)(offsetof(RSA_Encrypt_In, inScheme)),
933                              (UINT16)(offsetof(RSA_Encrypt_In, label))},
934     /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
935                              TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL,
936                              TPMT_RSA_DECRYPT_P_UNMARSHAL + ADD_FLAG,
937                              TPM2B_DATA_P_UNMARSHAL,
938                              END_OF_LIST,
939                              TPM2B_PUBLIC_KEY_RSA_P_MARSHAL,
940                              END_OF_LIST}
941 };
942 #define _RSA_EncryptDataAddress (&_RSA_EncryptData)
943 #else
944 #define _RSA_EncryptDataAddress 0
945 #endif // CC_RSA_Encrypt
946 #if CC_RSA_Decrypt
947 #include "RSA_Decrypt_fp.h"
948 typedef TPM_RC  (RSA_Decrypt_Entry)(
949     RSA_Decrypt_In              *in,
950     RSA_Decrypt_Out             *out
951 );
952 typedef const struct {
953     RSA_Decrypt_Entry       *entry;
954     UINT16                  inSize;
955     UINT16                  outSize;
956     UINT16                  offsetOfTypes;
957     UINT16                  paramOffsets[3];
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
958         BYTE                    types[7];
959     } RSA_Decrypt_COMMAND_DESCRIPTOR_t;
960     RSA_Decrypt_COMMAND_DESCRIPTOR_t _RSA_DecryptData = {
961         /* entry         */     &TPM2_RSA_Decrypt,
962         /* inSize        */     (UINT16)(sizeof(RSA_Decrypt_In)),
963         /* outSize       */     (UINT16)(sizeof(RSA_Decrypt_Out)),
964         /* offsetOfTypes */     offsetof(RSA_Decrypt_COMMAND_DESCRIPTOR_t, types),
965         /* offsets       */     {(UINT16)(offsetof(RSA_Decrypt_In, cipherText)),
966                                  (UINT16)(offsetof(RSA_Decrypt_In, inScheme)),
967                                  (UINT16)(offsetof(RSA_Decrypt_In, label))},
968         /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
969                                  TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL,
970                                  TPMT_RSA_DECRYPT_P_UNMARSHAL + ADD_FLAG,
971                                  TPM2B_DATA_P_UNMARSHAL,
972                                  END_OF_LIST,
973                                  TPM2B_PUBLIC_KEY_RSA_P_MARSHAL,
974                                  END_OF_LIST}
975     };
976     #define _RSA_DecryptDataAddress (&_RSA_DecryptData)
977     #else
978     #define _RSA_DecryptDataAddress 0
979     #endif // CC_RSA_Decrypt
980     #if CC_ECDH_KeyGen
981     #include "ECDH_KeyGen_fp.h"
982     typedef TPM_RC  (ECDH_KeyGen_Entry)(
983         ECDH_KeyGen_In                *in,
984         ECDH_KeyGen_Out               *out
985     );
986     typedef const struct {
987         ECDH_KeyGen_Entry       *entry;
988         UINT16                  inSize;
989         UINT16                  outSize;
990         UINT16                  offsetOfTypes;
991         UINT16                  paramOffsets[1];
992         BYTE                    types[5];
993     } ECDH_KeyGen_COMMAND_DESCRIPTOR_t;
994     ECDH_KeyGen_COMMAND_DESCRIPTOR_t _ECDH_KeyGenData = {
995         /* entry         */     &TPM2_ECDH_KeyGen,
996         /* inSize        */     (UINT16)(sizeof(ECDH_KeyGen_In)),
997         /* outSize       */     (UINT16)(sizeof(ECDH_KeyGen_Out)),
998         /* offsetOfTypes */     offsetof(ECDH_KeyGen_COMMAND_DESCRIPTOR_t, types),
999         /* offsets       */     {(UINT16)(offsetof(ECDH_KeyGen_Out, pubPoint))},
1000        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1001                                 END_OF_LIST,
1002                                 TPM2B_ECC_POINT_P_MARSHAL,
1003                                 TPM2B_ECC_POINT_P_MARSHAL,
1004                                 END_OF_LIST}
1005    };
1006    #define _ECDH_KeyGenDataAddress (&_ECDH_KeyGenData)
1007    #else
1008    #define _ECDH_KeyGenDataAddress 0
1009    #endif // CC_ECDH_KeyGen
1010    #if CC_ECDH_ZGen
1011    #include "ECDH_ZGen_fp.h"
1012    typedef TPM_RC  (ECDH_ZGen_Entry)(
1013        ECDH_ZGen_In                  *in,
1014        ECDH_ZGen_Out                 *out
1015    );
1016    typedef const struct {
1017        ECDH_ZGen_Entry         *entry;
1018        UINT16                  inSize;
1019        UINT16                  outSize;
1020        UINT16                  offsetOfTypes;
1021        UINT16                  paramOffsets[1];
1022        BYTE                    types[5];
1023    } ECDH_ZGen_COMMAND_DESCRIPTOR_t;
```


Level 00 Revision 01.59	Copyright © TCG 2006-2020	November 8, 2019

```
1024 ECDH_ZGen_COMMAND_DESCRIPTOR_t _ECDH_ZGenData = {
1025     /* entry          */     &TPM2_ECDH_ZGen,
1026     /* inSize         */     (UINT16)(sizeof(ECDH_ZGen_In)),
1027     /* outSize        */     (UINT16)(sizeof(ECDH_ZGen_Out)),
1028     /* offsetOfTypes */     offsetof(ECDH_ZGen_COMMAND_DESCRIPTOR_t, types),
1029     /* offsets        */     {(UINT16)(offsetof(ECDH_ZGen_In, inPoint))},
1030     /* types          */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1031                              TPM2B_ECC_POINT_P_UNMARSHAL,
1032                              END_OF_LIST,
1033                              TPM2B_ECC_POINT_P_MARSHAL,
1034                              END_OF_LIST}
1035 };
1036 #define _ECDH_ZGenDataAddress (&_ECDH_ZGenData)
1037 #else
1038 #define _ECDH_ZGenDataAddress 0
1039 #endif // CC_ECDH_ZGen
1040 #if CC_ECC_Parameters
1041 #include "ECC_Parameters_fp.h"
1042 typedef TPM_RC  (ECC_Parameters_Entry)(
1043     ECC_Parameters_In            *in,
1044     ECC_Parameters_Out           *out
1045 );
1046 typedef const struct {
1047     ECC_Parameters_Entry    *entry;
1048     UINT16                  inSize;
1049     UINT16                  outSize;
1050     UINT16                  offsetOfTypes;
1051     BYTE                    types[4];
1052 } ECC_Parameters_COMMAND_DESCRIPTOR_t;
1053 ECC_Parameters_COMMAND_DESCRIPTOR_t _ECC_ParametersData = {
1054     /* entry          */     &TPM2_ECC_Parameters,
1055     /* inSize         */     (UINT16)(sizeof(ECC_Parameters_In)),
1056     /* outSize        */     (UINT16)(sizeof(ECC_Parameters_Out)),
1057     /* offsetOfTypes */     offsetof(ECC_Parameters_COMMAND_DESCRIPTOR_t, types),
1058     /* offsets        */     // No parameter offsets;
1059     /* types          */     {TPMI_ECC_CURVE_P_UNMARSHAL,
1060                              END_OF_LIST,
1061                              TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL,
1062                              END_OF_LIST}
1063 };
1064 #define _ECC_ParametersDataAddress (&_ECC_ParametersData)
1065 #else
1066 #define _ECC_ParametersDataAddress 0
1067 #endif // CC_ECC_Parameters
1068 #if CC_ZGen_2Phase
1069 #include "ZGen_2Phase_fp.h"
1070 typedef TPM_RC  (ZGen_2Phase_Entry)(
1071     ZGen_2Phase_In               *in,
1072     ZGen_2Phase_Out              *out
1073 );
1074 typedef const struct {
1075     ZGen_2Phase_Entry       *entry;
1076     UINT16                  inSize;
1077     UINT16                  outSize;
1078     UINT16                  offsetOfTypes;
1079     UINT16                  paramOffsets[5];
1080     BYTE                    types[9];
1081 } ZGen_2Phase_COMMAND_DESCRIPTOR_t;
1082 ZGen_2Phase_COMMAND_DESCRIPTOR_t _ZGen_2PhaseData = {
1083     /* entry          */     &TPM2_ZGen_2Phase,
1084     /* inSize         */     (UINT16)(sizeof(ZGen_2Phase_In)),
1085     /* outSize        */     (UINT16)(sizeof(ZGen_2Phase_Out)),
1086     /* offsetOfTypes */     offsetof(ZGen_2Phase_COMMAND_DESCRIPTOR_t, types),
1087     /* offsets        */     {(UINT16)(offsetof(ZGen_2Phase_In, inQsB)),
1088                               (UINT16)(offsetof(ZGen_2Phase_In, inQeB)),
1089                               (UINT16)(offsetof(ZGen_2Phase_In, inScheme)),
```

```
1090                                 (UINT16)(offsetof(ZGen_2Phase_In, counter)),
1091                                 (UINT16)(offsetof(ZGen_2Phase_Out, outZ2))},
1092     /* types          */        {TPMI_DH_OBJECT_H_UNMARSHAL,
1093                                 TPM2B_ECC_POINT_P_UNMARSHAL,
1094                                 TPM2B_ECC_POINT_P_UNMARSHAL,
1095                                 TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL,
1096                                 UINT16_P_UNMARSHAL,
1097                                 END_OF_LIST,
1098                                 TPM2B_ECC_POINT_P_MARSHAL,
1099                                 TPM2B_ECC_POINT_P_MARSHAL,
1100                                 END_OF_LIST}
1101 };
1102 #define _ZGen_2PhaseDataAddress (&_ZGen_2PhaseData)
1103 #else
1104 #define _ZGen_2PhaseDataAddress 0
1105 #endif // CC_ZGen_2Phase
1106 #if CC_EncryptDecrypt
1107 #include "EncryptDecrypt_fp.h"
1108 typedef TPM_RC  (EncryptDecrypt_Entry)(
1109     EncryptDecrypt_In           *in,
1110     EncryptDecrypt_Out          *out
1111 );
1112 typedef const struct {
1113     EncryptDecrypt_Entry    *entry;
1114     UINT16                  inSize;
1115     UINT16                  outSize;
1116     UINT16                  offsetOfTypes;
1117     UINT16                  paramOffsets[5];
1118     BYTE                    types[9];
1119 } EncryptDecrypt_COMMAND_DESCRIPTOR_t;
1120 EncryptDecrypt_COMMAND_DESCRIPTOR_t _EncryptDecryptData = {
1121     /* entry          */        &TPM2_EncryptDecrypt,
1122     /* inSize         */        (UINT16)(sizeof(EncryptDecrypt_In)),
1123     /* outSize        */        (UINT16)(sizeof(EncryptDecrypt_Out)),
1124     /* offsetOfTypes */         offsetof(EncryptDecrypt_COMMAND_DESCRIPTOR_t, types),
1125     /* offsets        */        {(UINT16)(offsetof(EncryptDecrypt_In, decrypt)),
1126                                 (UINT16)(offsetof(EncryptDecrypt_In, mode)),
1127                                 (UINT16)(offsetof(EncryptDecrypt_In, ivIn)),
1128                                 (UINT16)(offsetof(EncryptDecrypt_In, inData)),
1129                                 (UINT16)(offsetof(EncryptDecrypt_Out, ivOut))},
1130     /* types          */        {TPMI_DH_OBJECT_H_UNMARSHAL,
1131                                 TPMI_YES_NO_P_UNMARSHAL,
1132                                 TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + ADD_FLAG,
1133                                 TPM2B_IV_P_UNMARSHAL,
1134                                 TPM2B_MAX_BUFFER_P_UNMARSHAL,
1135                                 END_OF_LIST,
1136                                 TPM2B_MAX_BUFFER_P_MARSHAL,
1137                                 TPM2B_IV_P_MARSHAL,
1138                                 END_OF_LIST}
1139 };
1140 #define _EncryptDecryptDataAddress (&_EncryptDecryptData)
1141 #else
1142 #define _EncryptDecryptDataAddress 0
1143 #endif // CC_EncryptDecrypt
1144 #if CC_EncryptDecrypt2
1145 #include "EncryptDecrypt2_fp.h"
1146 typedef TPM_RC  (EncryptDecrypt2_Entry)(
1147     EncryptDecrypt2_In          *in,
1148     EncryptDecrypt2_Out         *out
1149 );
1150 typedef const struct {
1151     EncryptDecrypt2_Entry   *entry;
1152     UINT16                  inSize;
1153     UINT16                  outSize;
1154     UINT16                  offsetOfTypes;
1155     UINT16                  paramOffsets[5];
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1156        BYTE                        types[9];
1157    } EncryptDecrypt2_COMMAND_DESCRIPTOR_t;
1158    EncryptDecrypt2_COMMAND_DESCRIPTOR_t _EncryptDecrypt2Data = {
1159        /* entry         */     &TPM2_EncryptDecrypt2,
1160        /* inSize        */     (UINT16)(sizeof(EncryptDecrypt2_In)),
1161        /* outSize       */     (UINT16)(sizeof(EncryptDecrypt2_Out)),
1162        /* offsetOfTypes */     offsetof(EncryptDecrypt2_COMMAND_DESCRIPTOR_t, types),
1163        /* offsets       */     {(UINT16)(offsetof(EncryptDecrypt2_In, inData)),
1164                                 (UINT16)(offsetof(EncryptDecrypt2_In, decrypt)),
1165                                 (UINT16)(offsetof(EncryptDecrypt2_In, mode)),
1166                                 (UINT16)(offsetof(EncryptDecrypt2_In, ivIn)),
1167                                 (UINT16)(offsetof(EncryptDecrypt2_Out, ivOut))},
1168        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1169                                 TPM2B_MAX_BUFFER_P_UNMARSHAL,
1170                                 TPMI_YES_NO_P_UNMARSHAL,
1171                                 TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + ADD_FLAG,
1172                                 TPM2B_IV_P_UNMARSHAL,
1173                                 END_OF_LIST,
1174                                 TPM2B_MAX_BUFFER_P_MARSHAL,
1175                                 TPM2B_IV_P_MARSHAL,
1176                                 END_OF_LIST}
1177    };
1178    #define _EncryptDecrypt2DataAddress (&_EncryptDecrypt2Data)
1179    #else
1180    #define _EncryptDecrypt2DataAddress 0
1181    #endif // CC_EncryptDecrypt2
1182    #if CC_Hash
1183    #include "Hash_fp.h"
1184    typedef TPM_RC  (Hash_Entry)(
1185        Hash_In                         *in,
1186        Hash_Out                        *out
1187    );
1188    typedef const struct {
1189        Hash_Entry                  *entry;
1190        UINT16                      inSize;
1191        UINT16                      outSize;
1192        UINT16                      offsetOfTypes;
1193        UINT16                      paramOffsets[3];
1194        BYTE                        types[7];
1195    } Hash_COMMAND_DESCRIPTOR_t;
1196    Hash_COMMAND_DESCRIPTOR_t _HashData = {
1197        /* entry         */     &TPM2_Hash,
1198        /* inSize        */     (UINT16)(sizeof(Hash_In)),
1199        /* outSize       */     (UINT16)(sizeof(Hash_Out)),
1200        /* offsetOfTypes */     offsetof(Hash_COMMAND_DESCRIPTOR_t, types),
1201        /* offsets       */     {(UINT16)(offsetof(Hash_In, hashAlg)),
1202                                 (UINT16)(offsetof(Hash_In, hierarchy)),
1203                                 (UINT16)(offsetof(Hash_Out, validation))},
1204        /* types         */     {TPM2B_MAX_BUFFER_P_UNMARSHAL,
1205                                 TPMI_ALG_HASH_P_UNMARSHAL,
1206                                 TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
1207                                 END_OF_LIST,
1208                                 TPM2B_DIGEST_P_MARSHAL,
1209                                 TPMT_TK_HASHCHECK_P_MARSHAL,
1210                                 END_OF_LIST}
1211    };
1212    #define _HashDataAddress (&_HashData)
1213    #else
1214    #define _HashDataAddress 0
1215    #endif // CC_Hash
1216    #if CC_HMAC
1217    #include "HMAC_fp.h"
1218    typedef TPM_RC  (HMAC_Entry)(
1219        HMAC_In                         *in,
1220        HMAC_Out                        *out
1221    );
```

 Copyright © TCG 2006-2020

```
1222    typedef const struct {
1223        HMAC_Entry              *entry;
1224        UINT16                  inSize;
1225        UINT16                  outSize;
1226        UINT16                  offsetOfTypes;
1227        UINT16                  paramOffsets[2];
1228        BYTE                    types[6];
1229    } HMAC_COMMAND_DESCRIPTOR_t;
1230    HMAC_COMMAND_DESCRIPTOR_t _HMACData = {
1231        /* entry         */      &TPM2_HMAC,
1232        /* inSize        */      (UINT16)(sizeof(HMAC_In)),
1233        /* outSize       */      (UINT16)(sizeof(HMAC_Out)),
1234        /* offsetOfTypes */      offsetof(HMAC_COMMAND_DESCRIPTOR_t, types),
1235        /* offsets       */      {(UINT16)(offsetof(HMAC_In, buffer)),
1236                                  (UINT16)(offsetof(HMAC_In, hashAlg))},
1237        /* types         */      {TPMI_DH_OBJECT_H_UNMARSHAL,
1238                                  TPM2B_MAX_BUFFER_P_UNMARSHAL,
1239                                  TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1240                                  END_OF_LIST,
1241                                  TPM2B_DIGEST_P_MARSHAL,
1242                                  END_OF_LIST}
1243    };
1244    #define _HMACDataAddress (&_HMACData)
1245    #else
1246    #define _HMACDataAddress 0
1247    #endif // CC_HMAC
1248    #if CC_MAC
1249    #include "MAC_fp.h"
1250    typedef TPM_RC  (MAC_Entry)(
1251        MAC_In                      *in,
1252        MAC_Out                     *out
1253    );
1254    typedef const struct {
1255        MAC_Entry               *entry;
1256        UINT16                  inSize;
1257        UINT16                  outSize;
1258        UINT16                  offsetOfTypes;
1259        UINT16                  paramOffsets[2];
1260        BYTE                    types[6];
1261    } MAC_COMMAND_DESCRIPTOR_t;
1262    MAC_COMMAND_DESCRIPTOR_t _MACData = {
1263        /* entry         */      &TPM2_MAC,
1264        /* inSize        */      (UINT16)(sizeof(MAC_In)),
1265        /* outSize       */      (UINT16)(sizeof(MAC_Out)),
1266        /* offsetOfTypes */      offsetof(MAC_COMMAND_DESCRIPTOR_t, types),
1267        /* offsets       */      {(UINT16)(offsetof(MAC_In, buffer)),
1268                                  (UINT16)(offsetof(MAC_In, inScheme))},
1269        /* types         */      {TPMI_DH_OBJECT_H_UNMARSHAL,
1270                                  TPM2B_MAX_BUFFER_P_UNMARSHAL,
1271                                  TPMI_ALG_MAC_SCHEME_P_UNMARSHAL + ADD_FLAG,
1272                                  END_OF_LIST,
1273                                  TPM2B_DIGEST_P_MARSHAL,
1274                                  END_OF_LIST}
1275    };
1276    #define _MACDataAddress (&_MACData)
1277    #else
1278    #define _MACDataAddress 0
1279    #endif // CC_MAC
1280    #if CC_GetRandom
1281    #include "GetRandom_fp.h"
1282    typedef TPM_RC  (GetRandom_Entry)(
1283        GetRandom_In                *in,
1284        GetRandom_Out               *out
1285    );
1286    typedef const struct {
1287        GetRandom_Entry         *entry;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1288        UINT16                  inSize;
1289        UINT16                  outSize;
1290        UINT16                  offsetOfTypes;
1291        BYTE                    types[4];
1292    } GetRandom_COMMAND_DESCRIPTOR_t;
1293    GetRandom_COMMAND_DESCRIPTOR_t _GetRandomData = {
1294        /* entry         */     &TPM2_GetRandom,
1295        /* inSize        */     (UINT16)(sizeof(GetRandom_In)),
1296        /* outSize       */     (UINT16)(sizeof(GetRandom_Out)),
1297        /* offsetOfTypes */     offsetof(GetRandom_COMMAND_DESCRIPTOR_t, types),
1298        /* offsets       */     // No parameter offsets;
1299        /* types         */     {UINT16_P_UNMARSHAL,
1300                                 END_OF_LIST,
1301                                 TPM2B_DIGEST_P_MARSHAL,
1302                                 END_OF_LIST}
1303    };
1304    #define _GetRandomDataAddress (&_GetRandomData)
1305    #else
1306    #define _GetRandomDataAddress 0
1307    #endif // CC_GetRandom
1308    #if CC_StirRandom
1309    #include "StirRandom_fp.h"
1310    typedef TPM_RC  (StirRandom_Entry)(
1311        StirRandom_In               *in
1312    );
1313    typedef const struct {
1314        StirRandom_Entry        *entry;
1315        UINT16                  inSize;
1316        UINT16                  outSize;
1317        UINT16                  offsetOfTypes;
1318        BYTE                    types[3];
1319    } StirRandom_COMMAND_DESCRIPTOR_t;
1320    StirRandom_COMMAND_DESCRIPTOR_t _StirRandomData = {
1321        /* entry         */     &TPM2_StirRandom,
1322        /* inSize        */     (UINT16)(sizeof(StirRandom_In)),
1323        /* outSize       */     0,
1324        /* offsetOfTypes */     offsetof(StirRandom_COMMAND_DESCRIPTOR_t, types),
1325        /* offsets       */     // No parameter offsets;
1326        /* types         */     {TPM2B_SENSITIVE_DATA_P_UNMARSHAL,
1327                                 END_OF_LIST,
1328                                 END_OF_LIST}
1329    };
1330    #define _StirRandomDataAddress (&_StirRandomData)
1331    #else
1332    #define _StirRandomDataAddress 0
1333    #endif // CC_StirRandom
1334    #if CC_HMAC_Start
1335    #include "HMAC_Start_fp.h"
1336    typedef TPM_RC  (HMAC_Start_Entry)(
1337        HMAC_Start_In               *in,
1338        HMAC_Start_Out              *out
1339    );
1340    typedef const struct {
1341        HMAC_Start_Entry        *entry;
1342        UINT16                  inSize;
1343        UINT16                  outSize;
1344        UINT16                  offsetOfTypes;
1345        UINT16                  paramOffsets[2];
1346        BYTE                    types[6];
1347    } HMAC_Start_COMMAND_DESCRIPTOR_t;
1348    HMAC_Start_COMMAND_DESCRIPTOR_t _HMAC_StartData = {
1349        /* entry         */     &TPM2_HMAC_Start,
1350        /* inSize        */     (UINT16)(sizeof(HMAC_Start_In)),
1351        /* outSize       */     (UINT16)(sizeof(HMAC_Start_Out)),
1352        /* offsetOfTypes */     offsetof(HMAC_Start_COMMAND_DESCRIPTOR_t, types),
1353        /* offsets       */     {(UINT16)(offsetof(HMAC_Start_In, auth)),
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
1354                                   (UINT16)(offsetof(HMAC_Start_In, hashAlg))},
1355        /* types          */      {TPMI_DH_OBJECT_H_UNMARSHAL,
1356                                   TPM2B_AUTH_P_UNMARSHAL,
1357                                   TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1358                                   END_OF_LIST,
1359                                   TPMI_DH_OBJECT_H_MARSHAL,
1360                                   END_OF_LIST}
1361    };
1362    #define _HMAC_StartDataAddress (&_HMAC_StartData)
1363    #else
1364    #define _HMAC_StartDataAddress 0
1365    #endif // CC_HMAC_Start
1366    #if CC_MAC_Start
1367    #include "MAC_Start_fp.h"
1368    typedef TPM_RC  (MAC_Start_Entry)(
1369        MAC_Start_In                  *in,
1370        MAC_Start_Out                 *out
1371    );
1372    typedef const struct {
1373        MAC_Start_Entry           *entry;
1374        UINT16                    inSize;
1375        UINT16                    outSize;
1376        UINT16                    offsetOfTypes;
1377        UINT16                    paramOffsets[2];
1378        BYTE                      types[6];
1379    } MAC_Start_COMMAND_DESCRIPTOR_t;
1380    MAC_Start_COMMAND_DESCRIPTOR_t _MAC_StartData = {
1381        /* entry          */      &TPM2_MAC_Start,
1382        /* inSize         */      (UINT16)(sizeof(MAC_Start_In)),
1383        /* outSize        */      (UINT16)(sizeof(MAC_Start_Out)),
1384        /* offsetOfTypes */       offsetof(MAC_Start_COMMAND_DESCRIPTOR_t, types),
1385        /* offsets        */      {(UINT16)(offsetof(MAC_Start_In, auth)),
1386                                   (UINT16)(offsetof(MAC_Start_In, inScheme))},
1387        /* types          */      {TPMI_DH_OBJECT_H_UNMARSHAL,
1388                                   TPM2B_AUTH_P_UNMARSHAL,
1389                                   TPMI_ALG_MAC_SCHEME_P_UNMARSHAL + ADD_FLAG,
1390                                   END_OF_LIST,
1391                                   TPMI_DH_OBJECT_H_MARSHAL,
1392                                   END_OF_LIST}
1393    };
1394    #define _MAC_StartDataAddress (&_MAC_StartData)
1395    #else
1396    #define _MAC_StartDataAddress 0
1397    #endif // CC_MAC_Start
1398    #if CC_HashSequenceStart
1399    #include "HashSequenceStart_fp.h"
1400    typedef TPM_RC  (HashSequenceStart_Entry)(
1401        HashSequenceStart_In              *in,
1402        HashSequenceStart_Out             *out
1403    );
1404    typedef const struct {
1405        HashSequenceStart_Entry       *entry;
1406        UINT16                        inSize;
1407        UINT16                        outSize;
1408        UINT16                        offsetOfTypes;
1409        UINT16                        paramOffsets[1];
1410        BYTE                          types[5];
1411    } HashSequenceStart_COMMAND_DESCRIPTOR_t;
1412    HashSequenceStart_COMMAND_DESCRIPTOR_t _HashSequenceStartData = {
1413        /* entry          */          &TPM2_HashSequenceStart,
1414        /* inSize         */          (UINT16)(sizeof(HashSequenceStart_In)),
1415        /* outSize        */          (UINT16)(sizeof(HashSequenceStart_Out)),
1416        /* offsetOfTypes */           offsetof(HashSequenceStart_COMMAND_DESCRIPTOR_t,
        types),
1417        /* offsets        */          {(UINT16)(offsetof(HashSequenceStart_In, hashAlg))},
1418        /* types          */          {TPM2B_AUTH_P_UNMARSHAL,
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1419                                    TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1420                                    END_OF_LIST,
1421                                    TPMI_DH_OBJECT_H_MARSHAL,
1422                                    END_OF_LIST}
1423    };
1424    #define _HashSequenceStartDataAddress (&_HashSequenceStartData)
1425    #else
1426    #define _HashSequenceStartDataAddress 0
1427    #endif // CC_HashSequenceStart
1428    #if CC_SequenceUpdate
1429    #include "SequenceUpdate_fp.h"
1430    typedef TPM_RC  (SequenceUpdate_Entry)(
1431        SequenceUpdate_In            *in
1432    );
1433    typedef const struct {
1434        SequenceUpdate_Entry    *entry;
1435        UINT16                  inSize;
1436        UINT16                  outSize;
1437        UINT16                  offsetOfTypes;
1438        UINT16                  paramOffsets[1];
1439        BYTE                    types[4];
1440    } SequenceUpdate_COMMAND_DESCRIPTOR_t;
1441    SequenceUpdate_COMMAND_DESCRIPTOR_t _SequenceUpdateData = {
1442        /* entry         */     &TPM2_SequenceUpdate,
1443        /* inSize        */     (UINT16)(sizeof(SequenceUpdate_In)),
1444        /* outSize       */     0,
1445        /* offsetOfTypes */     offsetof(SequenceUpdate_COMMAND_DESCRIPTOR_t, types),
1446        /* offsets       */     {(UINT16)(offsetof(SequenceUpdate_In, buffer))},
1447        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1448                                 TPM2B_MAX_BUFFER_P_UNMARSHAL,
1449                                 END_OF_LIST,
1450                                 END_OF_LIST}
1451    };
1452    #define _SequenceUpdateDataAddress (&_SequenceUpdateData)
1453    #else
1454    #define _SequenceUpdateDataAddress 0
1455    #endif // CC_SequenceUpdate
1456    #if CC_SequenceComplete
1457    #include "SequenceComplete_fp.h"
1458    typedef TPM_RC  (SequenceComplete_Entry)(
1459        SequenceComplete_In          *in,
1460        SequenceComplete_Out         *out
1461    );
1462    typedef const struct {
1463        SequenceComplete_Entry  *entry;
1464        UINT16                  inSize;
1465        UINT16                  outSize;
1466        UINT16                  offsetOfTypes;
1467        UINT16                  paramOffsets[3];
1468        BYTE                    types[7];
1469    } SequenceComplete_COMMAND_DESCRIPTOR_t;
1470    SequenceComplete_COMMAND_DESCRIPTOR_t _SequenceCompleteData = {
1471        /* entry         */     &TPM2_SequenceComplete,
1472        /* inSize        */     (UINT16)(sizeof(SequenceComplete_In)),
1473        /* outSize       */     (UINT16)(sizeof(SequenceComplete_Out)),
1474        /* offsetOfTypes */     offsetof(SequenceComplete_COMMAND_DESCRIPTOR_t, types),
1475        /* offsets       */     {(UINT16)(offsetof(SequenceComplete_In, buffer)),
1476                                 (UINT16)(offsetof(SequenceComplete_In, hierarchy)),
1477                                 (UINT16)(offsetof(SequenceComplete_Out, validation))},
1478        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1479                                 TPM2B_MAX_BUFFER_P_UNMARSHAL,
1480                                 TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
1481                                 END_OF_LIST,
1482                                 TPM2B_DIGEST_P_MARSHAL,
1483                                 TPMT_TK_HASHCHECK_P_MARSHAL,
1484                                 END_OF_LIST}
```

```
1485    };
1486    #define _SequenceCompleteDataAddress (&_SequenceCompleteData)
1487    #else
1488    #define _SequenceCompleteDataAddress 0
1489    #endif // CC_SequenceComplete
1490    #if CC_EventSequenceComplete
1491    #include "EventSequenceComplete_fp.h"
1492    typedef TPM_RC  (EventSequenceComplete_Entry)(
1493        EventSequenceComplete_In             *in,
1494        EventSequenceComplete_Out            *out
1495    );
1496    typedef const struct {
1497        EventSequenceComplete_Entry     *entry;
1498        UINT16                          inSize;
1499        UINT16                          outSize;
1500        UINT16                          offsetOfTypes;
1501        UINT16                          paramOffsets[2];
1502        BYTE                            types[6];
1503    } EventSequenceComplete_COMMAND_DESCRIPTOR_t;
1504    EventSequenceComplete_COMMAND_DESCRIPTOR_t _EventSequenceCompleteData = {
1505        /* entry         */             &TPM2_EventSequenceComplete,
1506        /* inSize        */             (UINT16)(sizeof(EventSequenceComplete_In)),
1507        /* outSize       */             (UINT16)(sizeof(EventSequenceComplete_Out)),
1508        /* offsetOfTypes */
        offsetof(EventSequenceComplete_COMMAND_DESCRIPTOR_t, types),
1509        /* offsets       */             {(UINT16)(offsetof(EventSequenceComplete_In,
        sequenceHandle)),
1510                                         (UINT16)(offsetof(EventSequenceComplete_In,
        buffer))},
1511        /* types         */             {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
1512                                         TPMI_DH_OBJECT_H_UNMARSHAL,
1513                                         TPM2B_MAX_BUFFER_P_UNMARSHAL,
1514                                         END_OF_LIST,
1515                                         TPML_DIGEST_VALUES_P_MARSHAL,
1516                                         END_OF_LIST}
1517    };
1518    #define _EventSequenceCompleteDataAddress (&_EventSequenceCompleteData)
1519    #else
1520    #define _EventSequenceCompleteDataAddress 0
1521    #endif // CC_EventSequenceComplete
1522    #if CC_Certify
1523    #include "Certify_fp.h"
1524    typedef TPM_RC  (Certify_Entry)(
1525        Certify_In                      *in,
1526        Certify_Out                     *out
1527    );
1528    typedef const struct {
1529        Certify_Entry            *entry;
1530        UINT16                   inSize;
1531        UINT16                   outSize;
1532        UINT16                   offsetOfTypes;
1533        UINT16                   paramOffsets[4];
1534        BYTE                     types[8];
1535    } Certify_COMMAND_DESCRIPTOR_t;
1536    Certify_COMMAND_DESCRIPTOR_t _CertifyData = {
1537        /* entry         */      &TPM2_Certify,
1538        /* inSize        */      (UINT16)(sizeof(Certify_In)),
1539        /* outSize       */      (UINT16)(sizeof(Certify_Out)),
1540        /* offsetOfTypes */      offsetof(Certify_COMMAND_DESCRIPTOR_t, types),
1541        /* offsets       */      {(UINT16)(offsetof(Certify_In, signHandle)),
1542                                  (UINT16)(offsetof(Certify_In, qualifyingData)),
1543                                  (UINT16)(offsetof(Certify_In, inScheme)),
1544                                  (UINT16)(offsetof(Certify_Out, signature))},
1545        /* types         */      {TPMI_DH_OBJECT_H_UNMARSHAL,
1546                                  TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1547                                  TPM2B_DATA_P_UNMARSHAL,
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1548                               TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1549                               END_OF_LIST,
1550                               TPM2B_ATTEST_P_MARSHAL,
1551                               TPMT_SIGNATURE_P_MARSHAL,
1552                               END_OF_LIST}
1553   };
1554   #define _CertifyDataAddress (&_CertifyData)
1555   #else
1556   #define _CertifyDataAddress 0
1557   #endif // CC_Certify
1558   #if CC_CertifyCreation
1559   #include "CertifyCreation_fp.h"
1560   typedef TPM_RC  (CertifyCreation_Entry)(
1561       CertifyCreation_In           *in,
1562       CertifyCreation_Out          *out
1563   );
1564   typedef const struct {
1565       CertifyCreation_Entry    *entry;
1566       UINT16                   inSize;
1567       UINT16                   outSize;
1568       UINT16                   offsetOfTypes;
1569       UINT16                   paramOffsets[6];
1570       BYTE                     types[10];
1571   } CertifyCreation_COMMAND_DESCRIPTOR_t;
1572   CertifyCreation_COMMAND_DESCRIPTOR_t _CertifyCreationData = {
1573       /* entry         */      &TPM2_CertifyCreation,
1574       /* inSize        */      (UINT16)(sizeof(CertifyCreation_In)),
1575       /* outSize       */      (UINT16)(sizeof(CertifyCreation_Out)),
1576       /* offsetOfTypes */      offsetof(CertifyCreation_COMMAND_DESCRIPTOR_t, types),
1577       /* offsets       */      {(UINT16)(offsetof(CertifyCreation_In, objectHandle)),
1578                                (UINT16)(offsetof(CertifyCreation_In, qualifyingData)),
1579                                (UINT16)(offsetof(CertifyCreation_In, creationHash)),
1580                                (UINT16)(offsetof(CertifyCreation_In, inScheme)),
1581                                (UINT16)(offsetof(CertifyCreation_In, creationTicket)),
1582                                (UINT16)(offsetof(CertifyCreation_Out, signature))},
1583       /* types         */      {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1584                               TPMI_DH_OBJECT_H_UNMARSHAL,
1585                               TPM2B_DATA_P_UNMARSHAL,
1586                               TPM2B_DIGEST_P_UNMARSHAL,
1587                               TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1588                               TPMT_TK_CREATION_P_UNMARSHAL,
1589                               END_OF_LIST,
1590                               TPM2B_ATTEST_P_MARSHAL,
1591                               TPMT_SIGNATURE_P_MARSHAL,
1592                               END_OF_LIST}
1593   };
1594   #define _CertifyCreationDataAddress (&_CertifyCreationData)
1595   #else
1596   #define _CertifyCreationDataAddress 0
1597   #endif // CC_CertifyCreation
1598   #if CC_Quote
1599   #include "Quote_fp.h"
1600   typedef TPM_RC  (Quote_Entry)(
1601       Quote_In                     *in,
1602       Quote_Out                    *out
1603   );
1604   typedef const struct {
1605       Quote_Entry              *entry;
1606       UINT16                   inSize;
1607       UINT16                   outSize;
1608       UINT16                   offsetOfTypes;
1609       UINT16                   paramOffsets[4];
1610       BYTE                     types[8];
1611   } Quote_COMMAND_DESCRIPTOR_t;
1612   Quote_COMMAND_DESCRIPTOR_t _QuoteData = {
1613       /* entry         */      &TPM2_Quote,
```

```
1614     /* inSize        */      (UINT16)(sizeof(Quote_In)),
1615     /* outSize       */      (UINT16)(sizeof(Quote_Out)),
1616     /* offsetOfTypes */      offsetof(Quote_COMMAND_DESCRIPTOR_t, types),
1617     /* offsets       */      {(UINT16)(offsetof(Quote_In, qualifyingData)),
1618                               (UINT16)(offsetof(Quote_In, inScheme)),
1619                               (UINT16)(offsetof(Quote_In, PCRselect)),
1620                               (UINT16)(offsetof(Quote_Out, signature))},
1621     /* types         */      {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1622                               TPM2B_DATA_P_UNMARSHAL,
1623                               TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1624                               TPML_PCR_SELECTION_P_UNMARSHAL,
1625                               END_OF_LIST,
1626                               TPM2B_ATTEST_P_MARSHAL,
1627                               TPMT_SIGNATURE_P_MARSHAL,
1628                               END_OF_LIST}
1629 };
1630 #define _QuoteDataAddress (&_QuoteData)
1631 #else
1632 #define _QuoteDataAddress 0
1633 #endif // CC_Quote
1634 #if CC_GetSessionAuditDigest
1635 #include "GetSessionAuditDigest_fp.h"
1636 typedef TPM_RC  (GetSessionAuditDigest_Entry)(
1637     GetSessionAuditDigest_In             *in,
1638     GetSessionAuditDigest_Out            *out
1639 );
1640 typedef const struct {
1641     GetSessionAuditDigest_Entry     *entry;
1642     UINT16                          inSize;
1643     UINT16                          outSize;
1644     UINT16                          offsetOfTypes;
1645     UINT16                          paramOffsets[5];
1646     BYTE                            types[9];
1647 } GetSessionAuditDigest_COMMAND_DESCRIPTOR_t;
1648 GetSessionAuditDigest_COMMAND_DESCRIPTOR_t _GetSessionAuditDigestData = {
1649     /* entry         */             &TPM2_GetSessionAuditDigest,
1650     /* inSize        */             (UINT16)(sizeof(GetSessionAuditDigest_In)),
1651     /* outSize       */             (UINT16)(sizeof(GetSessionAuditDigest_Out)),
1652     /* offsetOfTypes */
     offsetof(GetSessionAuditDigest_COMMAND_DESCRIPTOR_t, types),
1653     /* offsets       */             {(UINT16)(offsetof(GetSessionAuditDigest_In,
     signHandle)),
1654                                      (UINT16)(offsetof(GetSessionAuditDigest_In,
     sessionHandle)),
1655                                      (UINT16)(offsetof(GetSessionAuditDigest_In,
     qualifyingData)),
1656                                      (UINT16)(offsetof(GetSessionAuditDigest_In,
     inScheme)),
1657                                      (UINT16)(offsetof(GetSessionAuditDigest_Out,
     signature))},
1658     /* types         */             {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
1659                                      TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1660                                      TPMI_SH_HMAC_H_UNMARSHAL,
1661                                      TPM2B_DATA_P_UNMARSHAL,
1662                                      TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1663                                      END_OF_LIST,
1664                                      TPM2B_ATTEST_P_MARSHAL,
1665                                      TPMT_SIGNATURE_P_MARSHAL,
1666                                      END_OF_LIST}
1667 };
1668 #define _GetSessionAuditDigestDataAddress (&_GetSessionAuditDigestData)
1669 #else
1670 #define _GetSessionAuditDigestDataAddress 0
1671 #endif // CC_GetSessionAuditDigest
1672 #if CC_GetCommandAuditDigest
1673 #include "GetCommandAuditDigest_fp.h"
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1674 typedef TPM_RC  (GetCommandAuditDigest_Entry)(
1675     GetCommandAuditDigest_In             *in,
1676     GetCommandAuditDigest_Out            *out
1677 );
1678 typedef const struct {
1679     GetCommandAuditDigest_Entry     *entry;
1680     UINT16                          inSize;
1681     UINT16                          outSize;
1682     UINT16                          offsetOfTypes;
1683     UINT16                          paramOffsets[4];
1684     BYTE                            types[8];
1685 } GetCommandAuditDigest_COMMAND_DESCRIPTOR_t;
1686 GetCommandAuditDigest_COMMAND_DESCRIPTOR_t _GetCommandAuditDigestData = {
1687     /* entry         */             &TPM2_GetCommandAuditDigest,
1688     /* inSize        */             (UINT16)(sizeof(GetCommandAuditDigest_In)),
1689     /* outSize       */             (UINT16)(sizeof(GetCommandAuditDigest_Out)),
1690     /* offsetOfTypes */
     offsetof(GetCommandAuditDigest_COMMAND_DESCRIPTOR_t, types),
1691     /* offsets       */             {(UINT16)(offsetof(GetCommandAuditDigest_In,
     signHandle)),
1692                                      (UINT16)(offsetof(GetCommandAuditDigest_In,
     qualifyingData)),
1693                                      (UINT16)(offsetof(GetCommandAuditDigest_In,
     inScheme)),
1694                                      (UINT16)(offsetof(GetCommandAuditDigest_Out,
     signature))},
1695     /* types         */             {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
1696                                      TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1697                                      TPM2B_DATA_P_UNMARSHAL,
1698                                      TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1699                                      END_OF_LIST,
1700                                      TPM2B_ATTEST_P_MARSHAL,
1701                                      TPMT_SIGNATURE_P_MARSHAL,
1702                                      END_OF_LIST}
1703 };
1704 #define _GetCommandAuditDigestDataAddress (&_GetCommandAuditDigestData)
1705 #else
1706 #define _GetCommandAuditDigestDataAddress 0
1707 #endif // CC_GetCommandAuditDigest
1708 #if CC_GetTime
1709 #include "GetTime_fp.h"
1710 typedef TPM_RC  (GetTime_Entry)(
1711     GetTime_In                      *in,
1712     GetTime_Out                     *out
1713 );
1714 typedef const struct {
1715     GetTime_Entry               *entry;
1716     UINT16                      inSize;
1717     UINT16                      outSize;
1718     UINT16                      offsetOfTypes;
1719     UINT16                      paramOffsets[4];
1720     BYTE                        types[8];
1721 } GetTime_COMMAND_DESCRIPTOR_t;
1722 GetTime_COMMAND_DESCRIPTOR_t _GetTimeData = {
1723     /* entry         */     &TPM2_GetTime,
1724     /* inSize        */     (UINT16)(sizeof(GetTime_In)),
1725     /* outSize       */     (UINT16)(sizeof(GetTime_Out)),
1726     /* offsetOfTypes */     offsetof(GetTime_COMMAND_DESCRIPTOR_t, types),
1727     /* offsets       */     {(UINT16)(offsetof(GetTime_In, signHandle)),
1728                              (UINT16)(offsetof(GetTime_In, qualifyingData)),
1729                              (UINT16)(offsetof(GetTime_In, inScheme)),
1730                              (UINT16)(offsetof(GetTime_Out, signature))},
1731     /* types         */     {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
1732                              TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1733                              TPM2B_DATA_P_UNMARSHAL,
1734                              TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
```

 Copyright © TCG 2006-2020

```
1735                                 END_OF_LIST,
1736                                 TPM2B_ATTEST_P_MARSHAL,
1737                                 TPMT_SIGNATURE_P_MARSHAL,
1738                                 END_OF_LIST}
1739   };
1740   #define _GetTimeDataAddress (&_GetTimeData)
1741   #else
1742   #define _GetTimeDataAddress 0
1743   #endif // CC_GetTime
1744   #if CC_CertifyX509
1745   #include "CertifyX509_fp.h"
1746   typedef TPM_RC  (CertifyX509_Entry)(
1747       CertifyX509_In                *in,
1748       CertifyX509_Out               *out
1749   );
1750   typedef const struct {
1751       CertifyX509_Entry       *entry;
1752       UINT16                  inSize;
1753       UINT16                  outSize;
1754       UINT16                  offsetOfTypes;
1755       UINT16                  paramOffsets[6];
1756       BYTE                    types[10];
1757   } CertifyX509_COMMAND_DESCRIPTOR_t;
1758   CertifyX509_COMMAND_DESCRIPTOR_t _CertifyX509Data = {
1759       /* entry         */     &TPM2_CertifyX509,
1760       /* inSize        */     (UINT16)(sizeof(CertifyX509_In)),
1761       /* outSize       */     (UINT16)(sizeof(CertifyX509_Out)),
1762       /* offsetOfTypes */     offsetof(CertifyX509_COMMAND_DESCRIPTOR_t, types),
1763       /* offsets       */     {(UINT16)(offsetof(CertifyX509_In, signHandle)),
1764                              (UINT16)(offsetof(CertifyX509_In, reserved)),
1765                              (UINT16)(offsetof(CertifyX509_In, inScheme)),
1766                              (UINT16)(offsetof(CertifyX509_In, partialCertificate)),
1767                              (UINT16)(offsetof(CertifyX509_Out, tbsDigest)),
1768                              (UINT16)(offsetof(CertifyX509_Out, signature))},
1769       /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1770                              TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1771                              TPM2B_DATA_P_UNMARSHAL,
1772                              TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1773                              TPM2B_MAX_BUFFER_P_UNMARSHAL,
1774                              END_OF_LIST,
1775                              TPM2B_MAX_BUFFER_P_MARSHAL,
1776                              TPM2B_DIGEST_P_MARSHAL,
1777                              TPMT_SIGNATURE_P_MARSHAL,
1778                              END_OF_LIST}
1779   };
1780   #define _CertifyX509DataAddress (&_CertifyX509Data)
1781   #else
1782   #define _CertifyX509DataAddress 0
1783   #endif // CC_CertifyX509
1784   #if CC_Commit
1785   #include "Commit_fp.h"
1786   typedef TPM_RC  (Commit_Entry)(
1787       Commit_In                     *in,
1788       Commit_Out                    *out
1789   );
1790   typedef const struct {
1791       Commit_Entry            *entry;
1792       UINT16                  inSize;
1793       UINT16                  outSize;
1794       UINT16                  offsetOfTypes;
1795       UINT16                  paramOffsets[6];
1796       BYTE                    types[10];
1797   } Commit_COMMAND_DESCRIPTOR_t;
1798   Commit_COMMAND_DESCRIPTOR_t _CommitData = {
1799       /* entry         */     &TPM2_Commit,
1800       /* inSize        */     (UINT16)(sizeof(Commit_In)),
```

```
1801        /* outSize       */      (UINT16)(sizeof(Commit_Out)),
1802        /* offsetOfTypes */      offsetof(Commit_COMMAND_DESCRIPTOR_t, types),
1803        /* offsets       */      {(UINT16)(offsetof(Commit_In, P1)),
1804                                  (UINT16)(offsetof(Commit_In, s2)),
1805                                  (UINT16)(offsetof(Commit_In, y2)),
1806                                  (UINT16)(offsetof(Commit_Out, L)),
1807                                  (UINT16)(offsetof(Commit_Out, E)),
1808                                  (UINT16)(offsetof(Commit_Out, counter))},
1809        /* types         */      {TPMI_DH_OBJECT_H_UNMARSHAL,
1810                                  TPM2B_ECC_POINT_P_UNMARSHAL,
1811                                  TPM2B_SENSITIVE_DATA_P_UNMARSHAL,
1812                                  TPM2B_ECC_PARAMETER_P_UNMARSHAL,
1813                                  END_OF_LIST,
1814                                  TPM2B_ECC_POINT_P_MARSHAL,
1815                                  TPM2B_ECC_POINT_P_MARSHAL,
1816                                  TPM2B_ECC_POINT_P_MARSHAL,
1817                                  UINT16_P_MARSHAL,
1818                                  END_OF_LIST}
1819    };
1820    #define _CommitDataAddress (&_CommitData)
1821    #else
1822    #define _CommitDataAddress 0
1823    #endif // CC_Commit
1824    #if CC_EC_Ephemeral
1825    #include "EC_Ephemeral_fp.h"
1826    typedef TPM_RC  (EC_Ephemeral_Entry)(
1827        EC_Ephemeral_In              *in,
1828        EC_Ephemeral_Out             *out
1829    );
1830    typedef const struct {
1831        EC_Ephemeral_Entry      *entry;
1832        UINT16                  inSize;
1833        UINT16                  outSize;
1834        UINT16                  offsetOfTypes;
1835        UINT16                  paramOffsets[1];
1836        BYTE                    types[5];
1837    } EC_Ephemeral_COMMAND_DESCRIPTOR_t;
1838    EC_Ephemeral_COMMAND_DESCRIPTOR_t _EC_EphemeralData = {
1839        /* entry         */      &TPM2_EC_Ephemeral,
1840        /* inSize        */      (UINT16)(sizeof(EC_Ephemeral_In)),
1841        /* outSize       */      (UINT16)(sizeof(EC_Ephemeral_Out)),
1842        /* offsetOfTypes */      offsetof(EC_Ephemeral_COMMAND_DESCRIPTOR_t, types),
1843        /* offsets       */      {(UINT16)(offsetof(EC_Ephemeral_Out, counter))},
1844        /* types         */      {TPMI_ECC_CURVE_P_UNMARSHAL,
1845                                  END_OF_LIST,
1846                                  TPM2B_ECC_POINT_P_MARSHAL,
1847                                  UINT16_P_MARSHAL,
1848                                  END_OF_LIST}
1849    };
1850    #define _EC_EphemeralDataAddress (&_EC_EphemeralData)
1851    #else
1852    #define _EC_EphemeralDataAddress 0
1853    #endif // CC_EC_Ephemeral
1854    #if CC_VerifySignature
1855    #include "VerifySignature_fp.h"
1856    typedef TPM_RC  (VerifySignature_Entry)(
1857        VerifySignature_In           *in,
1858        VerifySignature_Out          *out
1859    );
1860    typedef const struct {
1861        VerifySignature_Entry   *entry;
1862        UINT16                  inSize;
1863        UINT16                  outSize;
1864        UINT16                  offsetOfTypes;
1865        UINT16                  paramOffsets[2];
1866        BYTE                    types[6];
```

```
1867    } VerifySignature_COMMAND_DESCRIPTOR_t;
1868    VerifySignature_COMMAND_DESCRIPTOR_t _VerifySignatureData = {
1869        /* entry         */     &TPM2_VerifySignature,
1870        /* inSize        */     (UINT16)(sizeof(VerifySignature_In)),
1871        /* outSize       */     (UINT16)(sizeof(VerifySignature_Out)),
1872        /* offsetOfTypes */     offsetof(VerifySignature_COMMAND_DESCRIPTOR_t, types),
1873        /* offsets       */     {(UINT16)(offsetof(VerifySignature_In, digest)),
1874                                 (UINT16)(offsetof(VerifySignature_In, signature))},
1875        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1876                                 TPM2B_DIGEST_P_UNMARSHAL,
1877                                 TPMT_SIGNATURE_P_UNMARSHAL,
1878                                 END_OF_LIST,
1879                                 TPMT_TK_VERIFIED_P_MARSHAL,
1880                                 END_OF_LIST}
1881    };
1882    #define _VerifySignatureDataAddress (&_VerifySignatureData)
1883    #else
1884    #define _VerifySignatureDataAddress 0
1885    #endif // CC_VerifySignature
1886    #if CC_Sign
1887    #include "Sign_fp.h"
1888    typedef TPM_RC  (Sign_Entry)(
1889        Sign_In                       *in,
1890        Sign_Out                      *out
1891    );
1892    typedef const struct {
1893        Sign_Entry                *entry;
1894        UINT16                    inSize;
1895        UINT16                    outSize;
1896        UINT16                    offsetOfTypes;
1897        UINT16                    paramOffsets[3];
1898        BYTE                      types[7];
1899    } Sign_COMMAND_DESCRIPTOR_t;
1900    Sign_COMMAND_DESCRIPTOR_t _SignData = {
1901        /* entry         */     &TPM2_Sign,
1902        /* inSize        */     (UINT16)(sizeof(Sign_In)),
1903        /* outSize       */     (UINT16)(sizeof(Sign_Out)),
1904        /* offsetOfTypes */     offsetof(Sign_COMMAND_DESCRIPTOR_t, types),
1905        /* offsets       */     {(UINT16)(offsetof(Sign_In, digest)),
1906                                 (UINT16)(offsetof(Sign_In, inScheme)),
1907                                 (UINT16)(offsetof(Sign_In, validation))},
1908        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1909                                 TPM2B_DIGEST_P_UNMARSHAL,
1910                                 TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1911                                 TPMT_TK_HASHCHECK_P_UNMARSHAL,
1912                                 END_OF_LIST,
1913                                 TPMT_SIGNATURE_P_MARSHAL,
1914                                 END_OF_LIST}
1915    };
1916    #define _SignDataAddress (&_SignData)
1917    #else
1918    #define _SignDataAddress 0
1919    #endif // CC_Sign
1920    #if CC_SetCommandCodeAuditStatus
1921    #include "SetCommandCodeAuditStatus_fp.h"
1922    typedef TPM_RC  (SetCommandCodeAuditStatus_Entry)(
1923        SetCommandCodeAuditStatus_In            *in
1924    );
1925    typedef const struct {
1926        SetCommandCodeAuditStatus_Entry     *entry;
1927        UINT16                              inSize;
1928        UINT16                              outSize;
1929        UINT16                              offsetOfTypes;
1930        UINT16                              paramOffsets[3];
1931        BYTE                                types[6];
1932    } SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1933    SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t _SetCommandCodeAuditStatusData = {
1934        /* entry          */                    &TPM2_SetCommandCodeAuditStatus,
1935        /* inSize         */
        (UINT16)(sizeof(SetCommandCodeAuditStatus_In)),
1936        /* outSize        */                    0,
1937        /* offsetOfTypes */
        offsetof(SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t, types),
1938        /* offsets        */
        {(UINT16)(offsetof(SetCommandCodeAuditStatus_In, auditAlg)),
1939
        (UINT16)(offsetof(SetCommandCodeAuditStatus_In, setList)),
1940
        (UINT16)(offsetof(SetCommandCodeAuditStatus_In, clearList))},
1941        /* types          */                    {TPMI_RH_PROVISION_H_UNMARSHAL,
1942                                                 TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1943                                                 TPML_CC_P_UNMARSHAL,
1944                                                 TPML_CC_P_UNMARSHAL,
1945                                                 END_OF_LIST,
1946                                                 END_OF_LIST}
1947    };
1948    #define _SetCommandCodeAuditStatusDataAddress (&_SetCommandCodeAuditStatusData)
1949    #else
1950    #define _SetCommandCodeAuditStatusDataAddress 0
1951    #endif // CC_SetCommandCodeAuditStatus
1952    #if CC_PCR_Extend
1953    #include "PCR_Extend_fp.h"
1954    typedef TPM_RC  (PCR_Extend_Entry)(
1955        PCR_Extend_In                *in
1956    );
1957    typedef const struct {
1958        PCR_Extend_Entry         *entry;
1959        UINT16                   inSize;
1960        UINT16                   outSize;
1961        UINT16                   offsetOfTypes;
1962        UINT16                   paramOffsets[1];
1963        BYTE                     types[4];
1964    } PCR_Extend_COMMAND_DESCRIPTOR_t;
1965    PCR_Extend_COMMAND_DESCRIPTOR_t _PCR_ExtendData = {
1966        /* entry          */     &TPM2_PCR_Extend,
1967        /* inSize         */     (UINT16)(sizeof(PCR_Extend_In)),
1968        /* outSize        */     0,
1969        /* offsetOfTypes */     offsetof(PCR_Extend_COMMAND_DESCRIPTOR_t, types),
1970        /* offsets        */     {(UINT16)(offsetof(PCR_Extend_In, digests))},
1971        /* types          */     {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
1972                                  TPML_DIGEST_VALUES_P_UNMARSHAL,
1973                                  END_OF_LIST,
1974                                  END_OF_LIST}
1975    };
1976    #define _PCR_ExtendDataAddress (&_PCR_ExtendData)
1977    #else
1978    #define _PCR_ExtendDataAddress 0
1979    #endif // CC_PCR_Extend
1980    #if CC_PCR_Event
1981    #include "PCR_Event_fp.h"
1982    typedef TPM_RC  (PCR_Event_Entry)(
1983        PCR_Event_In                 *in,
1984        PCR_Event_Out                *out
1985    );
1986    typedef const struct {
1987        PCR_Event_Entry          *entry;
1988        UINT16                   inSize;
1989        UINT16                   outSize;
1990        UINT16                   offsetOfTypes;
1991        UINT16                   paramOffsets[1];
1992        BYTE                     types[5];
1993    } PCR_Event_COMMAND_DESCRIPTOR_t;
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
1994  PCR_Event_COMMAND_DESCRIPTOR_t _PCR_EventData = {
1995      /* entry         */     &TPM2_PCR_Event,
1996      /* inSize        */     (UINT16)(sizeof(PCR_Event_In)),
1997      /* outSize       */     (UINT16)(sizeof(PCR_Event_Out)),
1998      /* offsetOfTypes */     offsetof(PCR_Event_COMMAND_DESCRIPTOR_t, types),
1999      /* offsets       */     {(UINT16)(offsetof(PCR_Event_In, eventData))},
2000      /* types         */     {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
2001                              TPM2B_EVENT_P_UNMARSHAL,
2002                              END_OF_LIST,
2003                              TPML_DIGEST_VALUES_P_MARSHAL,
2004                              END_OF_LIST}
2005  };
2006  #define _PCR_EventDataAddress (&_PCR_EventData)
2007  #else
2008  #define _PCR_EventDataAddress 0
2009  #endif // CC_PCR_Event
2010  #if CC_PCR_Read
2011  #include "PCR_Read_fp.h"
2012  typedef TPM_RC  (PCR_Read_Entry)(
2013      PCR_Read_In                   *in,
2014      PCR_Read_Out                  *out
2015  );
2016  typedef const struct {
2017      PCR_Read_Entry          *entry;
2018      UINT16                  inSize;
2019      UINT16                  outSize;
2020      UINT16                  offsetOfTypes;
2021      UINT16                  paramOffsets[2];
2022      BYTE                    types[6];
2023  } PCR_Read_COMMAND_DESCRIPTOR_t;
2024  PCR_Read_COMMAND_DESCRIPTOR_t _PCR_ReadData = {
2025      /* entry         */     &TPM2_PCR_Read,
2026      /* inSize        */     (UINT16)(sizeof(PCR_Read_In)),
2027      /* outSize       */     (UINT16)(sizeof(PCR_Read_Out)),
2028      /* offsetOfTypes */     offsetof(PCR_Read_COMMAND_DESCRIPTOR_t, types),
2029      /* offsets       */     {(UINT16)(offsetof(PCR_Read_Out, pcrSelectionOut)),
2030                               (UINT16)(offsetof(PCR_Read_Out, pcrValues))},
2031      /* types         */     {TPML_PCR_SELECTION_P_UNMARSHAL,
2032                              END_OF_LIST,
2033                              UINT32_P_MARSHAL,
2034                              TPML_PCR_SELECTION_P_MARSHAL,
2035                              TPML_DIGEST_P_MARSHAL,
2036                              END_OF_LIST}
2037  };
2038  #define _PCR_ReadDataAddress (&_PCR_ReadData)
2039  #else
2040  #define _PCR_ReadDataAddress 0
2041  #endif // CC_PCR_Read
2042  #if CC_PCR_Allocate
2043  #include "PCR_Allocate_fp.h"
2044  typedef TPM_RC  (PCR_Allocate_Entry)(
2045      PCR_Allocate_In               *in,
2046      PCR_Allocate_Out              *out
2047  );
2048  typedef const struct {
2049      PCR_Allocate_Entry      *entry;
2050      UINT16                  inSize;
2051      UINT16                  outSize;
2052      UINT16                  offsetOfTypes;
2053      UINT16                  paramOffsets[4];
2054      BYTE                    types[8];
2055  } PCR_Allocate_COMMAND_DESCRIPTOR_t;
2056  PCR_Allocate_COMMAND_DESCRIPTOR_t _PCR_AllocateData = {
2057      /* entry         */     &TPM2_PCR_Allocate,
2058      /* inSize        */     (UINT16)(sizeof(PCR_Allocate_In)),
2059      /* outSize       */     (UINT16)(sizeof(PCR_Allocate_Out)),
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
2060     /* offsetOfTypes */     offsetof(PCR_Allocate_COMMAND_DESCRIPTOR_t, types),
2061     /* offsets       */     {(UINT16)(offsetof(PCR_Allocate_In, pcrAllocation)),
2062                               (UINT16)(offsetof(PCR_Allocate_Out, maxPCR)),
2063                               (UINT16)(offsetof(PCR_Allocate_Out, sizeNeeded)),
2064                               (UINT16)(offsetof(PCR_Allocate_Out, sizeAvailable))},
2065     /* types         */     {TPMI_RH_PLATFORM_H_UNMARSHAL,
2066                              TPML_PCR_SELECTION_P_UNMARSHAL,
2067                              END_OF_LIST,
2068                              TPMI_YES_NO_P_MARSHAL,
2069                              UINT32_P_MARSHAL,
2070                              UINT32_P_MARSHAL,
2071                              UINT32_P_MARSHAL,
2072                              END_OF_LIST}
2073 };
2074 #define _PCR_AllocateDataAddress (&_PCR_AllocateData)
2075 #else
2076 #define _PCR_AllocateDataAddress 0
2077 #endif // CC_PCR_Allocate
2078 #if CC_PCR_SetAuthPolicy
2079 #include "PCR_SetAuthPolicy_fp.h"
2080 typedef TPM_RC  (PCR_SetAuthPolicy_Entry)(
2081     PCR_SetAuthPolicy_In            *in
2082 );
2083 typedef const struct {
2084     PCR_SetAuthPolicy_Entry     *entry;
2085     UINT16                      inSize;
2086     UINT16                      outSize;
2087     UINT16                      offsetOfTypes;
2088     UINT16                      paramOffsets[3];
2089     BYTE                        types[6];
2090 } PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t;
2091 PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t _PCR_SetAuthPolicyData = {
2092     /* entry         */         &TPM2_PCR_SetAuthPolicy,
2093     /* inSize        */         (UINT16)(sizeof(PCR_SetAuthPolicy_In)),
2094     /* outSize       */         0,
2095     /* offsetOfTypes */         offsetof(PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t,
types),
2096     /* offsets       */         {(UINT16)(offsetof(PCR_SetAuthPolicy_In, authPolicy)),
2097                                  (UINT16)(offsetof(PCR_SetAuthPolicy_In, hashAlg)),
2098                                  (UINT16)(offsetof(PCR_SetAuthPolicy_In, pcrNum))},
2099     /* types         */         {TPMI_RH_PLATFORM_H_UNMARSHAL,
2100                                  TPM2B_DIGEST_P_UNMARSHAL,
2101                                  TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
2102                                  TPMI_DH_PCR_P_UNMARSHAL,
2103                                  END_OF_LIST,
2104                                  END_OF_LIST}
2105 };
2106 #define _PCR_SetAuthPolicyDataAddress (&_PCR_SetAuthPolicyData)
2107 #else
2108 #define _PCR_SetAuthPolicyDataAddress 0
2109 #endif // CC_PCR_SetAuthPolicy
2110 #if CC_PCR_SetAuthValue
2111 #include "PCR_SetAuthValue_fp.h"
2112 typedef TPM_RC  (PCR_SetAuthValue_Entry)(
2113     PCR_SetAuthValue_In         *in
2114 );
2115 typedef const struct {
2116     PCR_SetAuthValue_Entry  *entry;
2117     UINT16                  inSize;
2118     UINT16                  outSize;
2119     UINT16                  offsetOfTypes;
2120     UINT16                  paramOffsets[1];
2121     BYTE                    types[4];
2122 } PCR_SetAuthValue_COMMAND_DESCRIPTOR_t;
2123 PCR_SetAuthValue_COMMAND_DESCRIPTOR_t _PCR_SetAuthValueData = {
2124     /* entry         */     &TPM2_PCR_SetAuthValue,
```

```
2125        /* inSize         */      (UINT16)(sizeof(PCR_SetAuthValue_In)),
2126        /* outSize        */      0,
2127        /* offsetOfTypes */       offsetof(PCR_SetAuthValue_COMMAND_DESCRIPTOR_t, types),
2128        /* offsets        */      {(UINT16)(offsetof(PCR_SetAuthValue_In, auth))},
2129        /* types          */      {TPMI_DH_PCR_H_UNMARSHAL,
2130                                   TPM2B_DIGEST_P_UNMARSHAL,
2131                                   END_OF_LIST,
2132                                   END_OF_LIST}
2133    };
2134    #define _PCR_SetAuthValueDataAddress (&_PCR_SetAuthValueData)
2135    #else
2136    #define _PCR_SetAuthValueDataAddress 0
2137    #endif // CC_PCR_SetAuthValue
2138    #if CC_PCR_Reset
2139    #include "PCR_Reset_fp.h"
2140    typedef TPM_RC  (PCR_Reset_Entry)(
2141        PCR_Reset_In                    *in
2142    );
2143    typedef const struct {
2144        PCR_Reset_Entry            *entry;
2145        UINT16                     inSize;
2146        UINT16                     outSize;
2147        UINT16                     offsetOfTypes;
2148        BYTE                       types[3];
2149    } PCR_Reset_COMMAND_DESCRIPTOR_t;
2150    PCR_Reset_COMMAND_DESCRIPTOR_t _PCR_ResetData = {
2151        /* entry          */      &TPM2_PCR_Reset,
2152        /* inSize         */      (UINT16)(sizeof(PCR_Reset_In)),
2153        /* outSize        */      0,
2154        /* offsetOfTypes */       offsetof(PCR_Reset_COMMAND_DESCRIPTOR_t, types),
2155        /* offsets        */      // No parameter offsets;
2156        /* types          */      {TPMI_DH_PCR_H_UNMARSHAL,
2157                                   END_OF_LIST,
2158                                   END_OF_LIST}
2159    };
2160    #define _PCR_ResetDataAddress (&_PCR_ResetData)
2161    #else
2162    #define _PCR_ResetDataAddress 0
2163    #endif // CC_PCR_Reset
2164    #if CC_PolicySigned
2165    #include "PolicySigned_fp.h"
2166    typedef TPM_RC  (PolicySigned_Entry)(
2167        PolicySigned_In                 *in,
2168        PolicySigned_Out                *out
2169    );
2170    typedef const struct {
2171        PolicySigned_Entry         *entry;
2172        UINT16                     inSize;
2173        UINT16                     outSize;
2174        UINT16                     offsetOfTypes;
2175        UINT16                     paramOffsets[7];
2176        BYTE                       types[11];
2177    } PolicySigned_COMMAND_DESCRIPTOR_t;
2178    PolicySigned_COMMAND_DESCRIPTOR_t _PolicySignedData = {
2179        /* entry          */      &TPM2_PolicySigned,
2180        /* inSize         */      (UINT16)(sizeof(PolicySigned_In)),
2181        /* outSize        */      (UINT16)(sizeof(PolicySigned_Out)),
2182        /* offsetOfTypes */       offsetof(PolicySigned_COMMAND_DESCRIPTOR_t, types),
2183        /* offsets        */      {(UINT16)(offsetof(PolicySigned_In, policySession)),
2184                                   (UINT16)(offsetof(PolicySigned_In, nonceTPM)),
2185                                   (UINT16)(offsetof(PolicySigned_In, cpHashA)),
2186                                   (UINT16)(offsetof(PolicySigned_In, policyRef)),
2187                                   (UINT16)(offsetof(PolicySigned_In, expiration)),
2188                                   (UINT16)(offsetof(PolicySigned_In, auth)),
2189                                   (UINT16)(offsetof(PolicySigned_Out, policyTicket))},
2190        /* types          */      {TPMI_DH_OBJECT_H_UNMARSHAL,
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
2191                                 TPMI_SH_POLICY_H_UNMARSHAL,
2192                                 TPM2B_NONCE_P_UNMARSHAL,
2193                                 TPM2B_DIGEST_P_UNMARSHAL,
2194                                 TPM2B_NONCE_P_UNMARSHAL,
2195                                 INT32_P_UNMARSHAL,
2196                                 TPMT_SIGNATURE_P_UNMARSHAL,
2197                                 END_OF_LIST,
2198                                 TPM2B_TIMEOUT_P_MARSHAL,
2199                                 TPMT_TK_AUTH_P_MARSHAL,
2200                                 END_OF_LIST}
2201   };
2202   #define _PolicySignedDataAddress (&_PolicySignedData)
2203   #else
2204   #define _PolicySignedDataAddress 0
2205   #endif // CC_PolicySigned
2206   #if CC_PolicySecret
2207   #include "PolicySecret_fp.h"
2208   typedef TPM_RC  (PolicySecret_Entry)(
2209       PolicySecret_In             *in,
2210       PolicySecret_Out            *out
2211   );
2212   typedef const struct {
2213       PolicySecret_Entry      *entry;
2214       UINT16                  inSize;
2215       UINT16                  outSize;
2216       UINT16                  offsetOfTypes;
2217       UINT16                  paramOffsets[6];
2218       BYTE                    types[10];
2219   } PolicySecret_COMMAND_DESCRIPTOR_t;
2220   PolicySecret_COMMAND_DESCRIPTOR_t _PolicySecretData = {
2221       /* entry         */     &TPM2_PolicySecret,
2222       /* inSize        */     (UINT16)(sizeof(PolicySecret_In)),
2223       /* outSize       */     (UINT16)(sizeof(PolicySecret_Out)),
2224       /* offsetOfTypes */     offsetof(PolicySecret_COMMAND_DESCRIPTOR_t, types),
2225       /* offsets       */     {(UINT16)(offsetof(PolicySecret_In, policySession)),
2226                              (UINT16)(offsetof(PolicySecret_In, nonceTPM)),
2227                              (UINT16)(offsetof(PolicySecret_In, cpHashA)),
2228                              (UINT16)(offsetof(PolicySecret_In, policyRef)),
2229                              (UINT16)(offsetof(PolicySecret_In, expiration)),
2230                              (UINT16)(offsetof(PolicySecret_Out, policyTicket))},
2231       /* types         */     {TPMI_DH_ENTITY_H_UNMARSHAL,
2232                              TPMI_SH_POLICY_H_UNMARSHAL,
2233                              TPM2B_NONCE_P_UNMARSHAL,
2234                              TPM2B_DIGEST_P_UNMARSHAL,
2235                              TPM2B_NONCE_P_UNMARSHAL,
2236                              INT32_P_UNMARSHAL,
2237                              END_OF_LIST,
2238                              TPM2B_TIMEOUT_P_MARSHAL,
2239                              TPMT_TK_AUTH_P_MARSHAL,
2240                              END_OF_LIST}
2241   };
2242   #define _PolicySecretDataAddress (&_PolicySecretData)
2243   #else
2244   #define _PolicySecretDataAddress 0
2245   #endif // CC_PolicySecret
2246   #if CC_PolicyTicket
2247   #include "PolicyTicket_fp.h"
2248   typedef TPM_RC  (PolicyTicket_Entry)(
2249       PolicyTicket_In             *in
2250   );
2251   typedef const struct {
2252       PolicyTicket_Entry      *entry;
2253       UINT16                  inSize;
2254       UINT16                  outSize;
2255       UINT16                  offsetOfTypes;
2256       UINT16                  paramOffsets[5];
```

```
2257        BYTE                      types[8];
2258    } PolicyTicket_COMMAND_DESCRIPTOR_t;
2259    PolicyTicket_COMMAND_DESCRIPTOR_t _PolicyTicketData = {
2260        /* entry         */      &TPM2_PolicyTicket,
2261        /* inSize        */      (UINT16)(sizeof(PolicyTicket_In)),
2262        /* outSize       */      0,
2263        /* offsetOfTypes */      offsetof(PolicyTicket_COMMAND_DESCRIPTOR_t, types),
2264        /* offsets       */      {(UINT16)(offsetof(PolicyTicket_In, timeout)),
2265                                  (UINT16)(offsetof(PolicyTicket_In, cpHashA)),
2266                                  (UINT16)(offsetof(PolicyTicket_In, policyRef)),
2267                                  (UINT16)(offsetof(PolicyTicket_In, authName)),
2268                                  (UINT16)(offsetof(PolicyTicket_In, ticket))},
2269        /* types         */      {TPMI_SH_POLICY_H_UNMARSHAL,
2270                                  TPM2B_TIMEOUT_P_UNMARSHAL,
2271                                  TPM2B_DIGEST_P_UNMARSHAL,
2272                                  TPM2B_NONCE_P_UNMARSHAL,
2273                                  TPM2B_NAME_P_UNMARSHAL,
2274                                  TPMT_TK_AUTH_P_UNMARSHAL,
2275                                  END_OF_LIST,
2276                                  END_OF_LIST}
2277    };
2278    #define _PolicyTicketDataAddress (&_PolicyTicketData)
2279    #else
2280    #define _PolicyTicketDataAddress 0
2281    #endif // CC_PolicyTicket
2282    #if CC_PolicyOR
2283    #include "PolicyOR_fp.h"
2284    typedef TPM_RC  (PolicyOR_Entry)(
2285        PolicyOR_In                   *in
2286    );
2287    typedef const struct {
2288        PolicyOR_Entry            *entry;
2289        UINT16                    inSize;
2290        UINT16                    outSize;
2291        UINT16                    offsetOfTypes;
2292        UINT16                    paramOffsets[1];
2293        BYTE                      types[4];
2294    } PolicyOR_COMMAND_DESCRIPTOR_t;
2295    PolicyOR_COMMAND_DESCRIPTOR_t _PolicyORData = {
2296        /* entry         */      &TPM2_PolicyOR,
2297        /* inSize        */      (UINT16)(sizeof(PolicyOR_In)),
2298        /* outSize       */      0,
2299        /* offsetOfTypes */      offsetof(PolicyOR_COMMAND_DESCRIPTOR_t, types),
2300        /* offsets       */      {(UINT16)(offsetof(PolicyOR_In, pHashList))},
2301        /* types         */      {TPMI_SH_POLICY_H_UNMARSHAL,
2302                                  TPML_DIGEST_P_UNMARSHAL,
2303                                  END_OF_LIST,
2304                                  END_OF_LIST}
2305    };
2306    #define _PolicyORDataAddress (&_PolicyORData)
2307    #else
2308    #define _PolicyORDataAddress 0
2309    #endif // CC_PolicyOR
2310    #if CC_PolicyPCR
2311    #include "PolicyPCR_fp.h"
2312    typedef TPM_RC  (PolicyPCR_Entry)(
2313        PolicyPCR_In                  *in
2314    );
2315    typedef const struct {
2316        PolicyPCR_Entry           *entry;
2317        UINT16                    inSize;
2318        UINT16                    outSize;
2319        UINT16                    offsetOfTypes;
2320        UINT16                    paramOffsets[2];
2321        BYTE                      types[5];
2322    } PolicyPCR_COMMAND_DESCRIPTOR_t;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
2323 PolicyPCR_COMMAND_DESCRIPTOR_t _PolicyPCRData = {
2324     /* entry         */     &TPM2_PolicyPCR,
2325     /* inSize        */     (UINT16)(sizeof(PolicyPCR_In)),
2326     /* outSize       */     0,
2327     /* offsetOfTypes */     offsetof(PolicyPCR_COMMAND_DESCRIPTOR_t, types),
2328     /* offsets       */     {(UINT16)(offsetof(PolicyPCR_In, pcrDigest)),
2329                              (UINT16)(offsetof(PolicyPCR_In, pcrs))},
2330     /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2331                              TPM2B_DIGEST_P_UNMARSHAL,
2332                              TPML_PCR_SELECTION_P_UNMARSHAL,
2333                              END_OF_LIST,
2334                              END_OF_LIST}
2335 };
2336 #define _PolicyPCRDataAddress (&_PolicyPCRData)
2337 #else
2338 #define _PolicyPCRDataAddress 0
2339 #endif // CC_PolicyPCR
2340 #if CC_PolicyLocality
2341 #include "PolicyLocality_fp.h"
2342 typedef TPM_RC  (PolicyLocality_Entry)(
2343     PolicyLocality_In           *in
2344 );
2345 typedef const struct {
2346     PolicyLocality_Entry    *entry;
2347     UINT16                  inSize;
2348     UINT16                  outSize;
2349     UINT16                  offsetOfTypes;
2350     UINT16                  paramOffsets[1];
2351     BYTE                    types[4];
2352 } PolicyLocality_COMMAND_DESCRIPTOR_t;
2353 PolicyLocality_COMMAND_DESCRIPTOR_t _PolicyLocalityData = {
2354     /* entry         */     &TPM2_PolicyLocality,
2355     /* inSize        */     (UINT16)(sizeof(PolicyLocality_In)),
2356     /* outSize       */     0,
2357     /* offsetOfTypes */     offsetof(PolicyLocality_COMMAND_DESCRIPTOR_t, types),
2358     /* offsets       */     {(UINT16)(offsetof(PolicyLocality_In, locality))},
2359     /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2360                              TPMA_LOCALITY_P_UNMARSHAL,
2361                              END_OF_LIST,
2362                              END_OF_LIST}
2363 };
2364 #define _PolicyLocalityDataAddress (&_PolicyLocalityData)
2365 #else
2366 #define _PolicyLocalityDataAddress 0
2367 #endif // CC_PolicyLocality
2368 #if CC_PolicyNV
2369 #include "PolicyNV_fp.h"
2370 typedef TPM_RC  (PolicyNV_Entry)(
2371     PolicyNV_In                 *in
2372 );
2373 typedef const struct {
2374     PolicyNV_Entry          *entry;
2375     UINT16                  inSize;
2376     UINT16                  outSize;
2377     UINT16                  offsetOfTypes;
2378     UINT16                  paramOffsets[5];
2379     BYTE                    types[8];
2380 } PolicyNV_COMMAND_DESCRIPTOR_t;
2381 PolicyNV_COMMAND_DESCRIPTOR_t _PolicyNVData = {
2382     /* entry         */     &TPM2_PolicyNV,
2383     /* inSize        */     (UINT16)(sizeof(PolicyNV_In)),
2384     /* outSize       */     0,
2385     /* offsetOfTypes */     offsetof(PolicyNV_COMMAND_DESCRIPTOR_t, types),
2386     /* offsets       */     {(UINT16)(offsetof(PolicyNV_In, nvIndex)),
2387                              (UINT16)(offsetof(PolicyNV_In, policySession)),
2388                              (UINT16)(offsetof(PolicyNV_In, operandB)),
```

```
2389                                  (UINT16)(offsetof(PolicyNV_In, offset)),
2390                                  (UINT16)(offsetof(PolicyNV_In, operation))},
2391        /* types          */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
2392                                  TPMI_RH_NV_INDEX_H_UNMARSHAL,
2393                                  TPMI_SH_POLICY_H_UNMARSHAL,
2394                                  TPM2B_OPERAND_P_UNMARSHAL,
2395                                  UINT16_P_UNMARSHAL,
2396                                  TPM_EO_P_UNMARSHAL,
2397                                  END_OF_LIST,
2398                                  END_OF_LIST}
2399    };
2400    #define _PolicyNVDataAddress (&_PolicyNVData)
2401    #else
2402    #define _PolicyNVDataAddress 0
2403    #endif // CC_PolicyNV
2404    #if CC_PolicyCounterTimer
2405    #include "PolicyCounterTimer_fp.h"
2406    typedef TPM_RC  (PolicyCounterTimer_Entry)(
2407        PolicyCounterTimer_In           *in
2408    );
2409    typedef const struct {
2410        PolicyCounterTimer_Entry    *entry;
2411        UINT16                      inSize;
2412        UINT16                      outSize;
2413        UINT16                      offsetOfTypes;
2414        UINT16                      paramOffsets[3];
2415        BYTE                        types[6];
2416    } PolicyCounterTimer_COMMAND_DESCRIPTOR_t;
2417    PolicyCounterTimer_COMMAND_DESCRIPTOR_t _PolicyCounterTimerData = {
2418        /* entry          */        &TPM2_PolicyCounterTimer,
2419        /* inSize         */        (UINT16)(sizeof(PolicyCounterTimer_In)),
2420        /* outSize        */        0,
2421        /* offsetOfTypes */         offsetof(PolicyCounterTimer_COMMAND_DESCRIPTOR_t,
        types),
2422        /* offsets        */        {(UINT16)(offsetof(PolicyCounterTimer_In, operandB)),
2423                                     (UINT16)(offsetof(PolicyCounterTimer_In, offset)),
2424                                     (UINT16)(offsetof(PolicyCounterTimer_In,
        operation))},
2425        /* types          */        {TPMI_SH_POLICY_H_UNMARSHAL,
2426                                     TPM2B_OPERAND_P_UNMARSHAL,
2427                                     UINT16_P_UNMARSHAL,
2428                                     TPM_EO_P_UNMARSHAL,
2429                                     END_OF_LIST,
2430                                     END_OF_LIST}
2431    };
2432    #define _PolicyCounterTimerDataAddress (&_PolicyCounterTimerData)
2433    #else
2434    #define _PolicyCounterTimerDataAddress 0
2435    #endif // CC_PolicyCounterTimer
2436    #if CC_PolicyCommandCode
2437    #include "PolicyCommandCode_fp.h"
2438    typedef TPM_RC  (PolicyCommandCode_Entry)(
2439        PolicyCommandCode_In            *in
2440    );
2441    typedef const struct {
2442        PolicyCommandCode_Entry     *entry;
2443        UINT16                      inSize;
2444        UINT16                      outSize;
2445        UINT16                      offsetOfTypes;
2446        UINT16                      paramOffsets[1];
2447        BYTE                        types[4];
2448    } PolicyCommandCode_COMMAND_DESCRIPTOR_t;
2449    PolicyCommandCode_COMMAND_DESCRIPTOR_t _PolicyCommandCodeData = {
2450        /* entry          */        &TPM2_PolicyCommandCode,
2451        /* inSize         */        (UINT16)(sizeof(PolicyCommandCode_In)),
2452        /* outSize        */        0,
```


 Copyright © TCG 2006-2020

```
2453     /* offsetOfTypes */         offsetof(PolicyCommandCode_COMMAND_DESCRIPTOR_t,
types),
2454     /* offsets       */         {(UINT16)(offsetof(PolicyCommandCode_In, code))},
2455     /* types         */         {TPMI_SH_POLICY_H_UNMARSHAL,
2456                                  TPM_CC_P_UNMARSHAL,
2457                                  END_OF_LIST,
2458                                  END_OF_LIST}
2459 };
2460 #define _PolicyCommandCodeDataAddress (&_PolicyCommandCodeData)
2461 #else
2462 #define _PolicyCommandCodeDataAddress 0
2463 #endif // CC_PolicyCommandCode
2464 #if CC_PolicyPhysicalPresence
2465 #include "PolicyPhysicalPresence_fp.h"
2466 typedef TPM_RC  (PolicyPhysicalPresence_Entry)(
2467     PolicyPhysicalPresence_In           *in
2468 );
2469 typedef const struct {
2470     PolicyPhysicalPresence_Entry    *entry;
2471     UINT16                          inSize;
2472     UINT16                          outSize;
2473     UINT16                          offsetOfTypes;
2474     BYTE                            types[3];
2475 } PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t;
2476 PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t _PolicyPhysicalPresenceData = {
2477     /* entry         */             &TPM2_PolicyPhysicalPresence,
2478     /* inSize        */             (UINT16)(sizeof(PolicyPhysicalPresence_In)),
2479     /* outSize       */             0,
2480     /* offsetOfTypes */
offsetof(PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t, types),
2481     /* offsets       */             // No parameter offsets;
2482     /* types         */             {TPMI_SH_POLICY_H_UNMARSHAL,
2483                                      END_OF_LIST,
2484                                      END_OF_LIST}
2485 };
2486 #define _PolicyPhysicalPresenceDataAddress (&_PolicyPhysicalPresenceData)
2487 #else
2488 #define _PolicyPhysicalPresenceDataAddress 0
2489 #endif // CC_PolicyPhysicalPresence
2490 #if CC_PolicyCpHash
2491 #include "PolicyCpHash_fp.h"
2492 typedef TPM_RC  (PolicyCpHash_Entry)(
2493     PolicyCpHash_In             *in
2494 );
2495 typedef const struct {
2496     PolicyCpHash_Entry      *entry;
2497     UINT16                  inSize;
2498     UINT16                  outSize;
2499     UINT16                  offsetOfTypes;
2500     UINT16                  paramOffsets[1];
2501     BYTE                    types[4];
2502 } PolicyCpHash_COMMAND_DESCRIPTOR_t;
2503 PolicyCpHash_COMMAND_DESCRIPTOR_t _PolicyCpHashData = {
2504     /* entry         */     &TPM2_PolicyCpHash,
2505     /* inSize        */     (UINT16)(sizeof(PolicyCpHash_In)),
2506     /* outSize       */     0,
2507     /* offsetOfTypes */     offsetof(PolicyCpHash_COMMAND_DESCRIPTOR_t, types),
2508     /* offsets       */     {(UINT16)(offsetof(PolicyCpHash_In, cpHashA))},
2509     /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2510                              TPM2B_DIGEST_P_UNMARSHAL,
2511                              END_OF_LIST,
2512                              END_OF_LIST}
2513 };
2514 #define _PolicyCpHashDataAddress (&_PolicyCpHashData)
2515 #else
2516 #define _PolicyCpHashDataAddress 0
```

```
2517 #endif // CC_PolicyCpHash
2518 #if CC_PolicyNameHash
2519 #include "PolicyNameHash_fp.h"
2520 typedef TPM_RC  (PolicyNameHash_Entry)(
2521     PolicyNameHash_In           *in
2522 );
2523 typedef const struct {
2524     PolicyNameHash_Entry    *entry;
2525     UINT16                  inSize;
2526     UINT16                  outSize;
2527     UINT16                  offsetOfTypes;
2528     UINT16                  paramOffsets[1];
2529     BYTE                    types[4];
2530 } PolicyNameHash_COMMAND_DESCRIPTOR_t;
2531 PolicyNameHash_COMMAND_DESCRIPTOR_t _PolicyNameHashData = {
2532     /* entry         */     &TPM2_PolicyNameHash,
2533     /* inSize        */     (UINT16)(sizeof(PolicyNameHash_In)),
2534     /* outSize       */     0,
2535     /* offsetOfTypes */     offsetof(PolicyNameHash_COMMAND_DESCRIPTOR_t, types),
2536     /* offsets       */     {(UINT16)(offsetof(PolicyNameHash_In, nameHash))},
2537     /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2538                              TPM2B_DIGEST_P_UNMARSHAL,
2539                              END_OF_LIST,
2540                              END_OF_LIST}
2541 };
2542 #define _PolicyNameHashDataAddress (&_PolicyNameHashData)
2543 #else
2544 #define _PolicyNameHashDataAddress 0
2545 #endif // CC_PolicyNameHash
2546 #if CC_PolicyDuplicationSelect
2547 #include "PolicyDuplicationSelect_fp.h"
2548 typedef TPM_RC  (PolicyDuplicationSelect_Entry)(
2549     PolicyDuplicationSelect_In          *in
2550 );
2551 typedef const struct {
2552     PolicyDuplicationSelect_Entry   *entry;
2553     UINT16                          inSize;
2554     UINT16                          outSize;
2555     UINT16                          offsetOfTypes;
2556     UINT16                          paramOffsets[3];
2557     BYTE                            types[6];
2558 } PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t;
2559 PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t _PolicyDuplicationSelectData = {
2560     /* entry         */             &TPM2_PolicyDuplicationSelect,
2561     /* inSize        */             (UINT16)(sizeof(PolicyDuplicationSelect_In)),
2562     /* outSize       */             0,
2563     /* offsetOfTypes */
     offsetof(PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t, types),
2564     /* offsets       */             {(UINT16)(offsetof(PolicyDuplicationSelect_In,
     objectName)),
2565                                      (UINT16)(offsetof(PolicyDuplicationSelect_In,
     newParentName)),
2566                                      (UINT16)(offsetof(PolicyDuplicationSelect_In,
     includeObject))},
2567     /* types         */             {TPMI_SH_POLICY_H_UNMARSHAL,
2568                                      TPM2B_NAME_P_UNMARSHAL,
2569                                      TPM2B_NAME_P_UNMARSHAL,
2570                                      TPMI_YES_NO_P_UNMARSHAL,
2571                                      END_OF_LIST,
2572                                      END_OF_LIST}
2573 };
2574 #define _PolicyDuplicationSelectDataAddress (&_PolicyDuplicationSelectData)
2575 #else
2576 #define _PolicyDuplicationSelectDataAddress 0
2577 #endif // CC_PolicyDuplicationSelect
2578 #if CC_PolicyAuthorize
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
2579 #include "PolicyAuthorize_fp.h"
2580 typedef TPM_RC  (PolicyAuthorize_Entry)(
2581     PolicyAuthorize_In          *in
2582 );
2583 typedef const struct {
2584     PolicyAuthorize_Entry   *entry;
2585     UINT16                  inSize;
2586     UINT16                  outSize;
2587     UINT16                  offsetOfTypes;
2588     UINT16                  paramOffsets[4];
2589     BYTE                    types[7];
2590 } PolicyAuthorize_COMMAND_DESCRIPTOR_t;
2591 PolicyAuthorize_COMMAND_DESCRIPTOR_t _PolicyAuthorizeData = {
2592     /* entry         */     &TPM2_PolicyAuthorize,
2593     /* inSize        */     (UINT16)(sizeof(PolicyAuthorize_In)),
2594     /* outSize       */     0,
2595     /* offsetOfTypes */     offsetof(PolicyAuthorize_COMMAND_DESCRIPTOR_t, types),
2596     /* offsets       */     {(UINT16)(offsetof(PolicyAuthorize_In, approvedPolicy)),
2597                              (UINT16)(offsetof(PolicyAuthorize_In, policyRef)),
2598                              (UINT16)(offsetof(PolicyAuthorize_In, keySign)),
2599                              (UINT16)(offsetof(PolicyAuthorize_In, checkTicket))},
2600     /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2601                              TPM2B_DIGEST_P_UNMARSHAL,
2602                              TPM2B_NONCE_P_UNMARSHAL,
2603                              TPM2B_NAME_P_UNMARSHAL,
2604                              TPMT_TK_VERIFIED_P_UNMARSHAL,
2605                              END_OF_LIST,
2606                              END_OF_LIST}
2607 };
2608 #define _PolicyAuthorizeDataAddress (&_PolicyAuthorizeData)
2609 #else
2610 #define _PolicyAuthorizeDataAddress 0
2611 #endif // CC_PolicyAuthorize
2612 #if CC_PolicyAuthValue
2613 #include "PolicyAuthValue_fp.h"
2614 typedef TPM_RC  (PolicyAuthValue_Entry)(
2615     PolicyAuthValue_In          *in
2616 );
2617 typedef const struct {
2618     PolicyAuthValue_Entry   *entry;
2619     UINT16                  inSize;
2620     UINT16                  outSize;
2621     UINT16                  offsetOfTypes;
2622     BYTE                    types[3];
2623 } PolicyAuthValue_COMMAND_DESCRIPTOR_t;
2624 PolicyAuthValue_COMMAND_DESCRIPTOR_t _PolicyAuthValueData = {
2625     /* entry         */     &TPM2_PolicyAuthValue,
2626     /* inSize        */     (UINT16)(sizeof(PolicyAuthValue_In)),
2627     /* outSize       */     0,
2628     /* offsetOfTypes */     offsetof(PolicyAuthValue_COMMAND_DESCRIPTOR_t, types),
2629     /* offsets       */     // No parameter offsets;
2630     /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2631                              END_OF_LIST,
2632                              END_OF_LIST}
2633 };
2634 #define _PolicyAuthValueDataAddress (&_PolicyAuthValueData)
2635 #else
2636 #define _PolicyAuthValueDataAddress 0
2637 #endif // CC_PolicyAuthValue
2638 #if CC_PolicyPassword
2639 #include "PolicyPassword_fp.h"
2640 typedef TPM_RC  (PolicyPassword_Entry)(
2641     PolicyPassword_In           *in
2642 );
2643 typedef const struct {
2644     PolicyPassword_Entry    *entry;
```

 Copyright © TCG 2006-2020

```
2645     UINT16                  inSize;
2646     UINT16                  outSize;
2647     UINT16                  offsetOfTypes;
2648     BYTE                    types[3];
2649 } PolicyPassword_COMMAND_DESCRIPTOR_t;
2650 PolicyPassword_COMMAND_DESCRIPTOR_t _PolicyPasswordData = {
2651     /* entry         */     &TPM2_PolicyPassword,
2652     /* inSize        */     (UINT16)(sizeof(PolicyPassword_In)),
2653     /* outSize       */     0,
2654     /* offsetOfTypes */     offsetof(PolicyPassword_COMMAND_DESCRIPTOR_t, types),
2655     /* offsets       */     // No parameter offsets;
2656     /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2657                              END_OF_LIST,
2658                              END_OF_LIST}
2659 };
2660 #define _PolicyPasswordDataAddress (&_PolicyPasswordData)
2661 #else
2662 #define _PolicyPasswordDataAddress 0
2663 #endif // CC_PolicyPassword
2664 #if CC_PolicyGetDigest
2665 #include "PolicyGetDigest_fp.h"
2666 typedef TPM_RC  (PolicyGetDigest_Entry)(
2667     PolicyGetDigest_In          *in,
2668     PolicyGetDigest_Out         *out
2669 );
2670 typedef const struct {
2671     PolicyGetDigest_Entry   *entry;
2672     UINT16                  inSize;
2673     UINT16                  outSize;
2674     UINT16                  offsetOfTypes;
2675     BYTE                    types[4];
2676 } PolicyGetDigest_COMMAND_DESCRIPTOR_t;
2677 PolicyGetDigest_COMMAND_DESCRIPTOR_t _PolicyGetDigestData = {
2678     /* entry         */     &TPM2_PolicyGetDigest,
2679     /* inSize        */     (UINT16)(sizeof(PolicyGetDigest_In)),
2680     /* outSize       */     (UINT16)(sizeof(PolicyGetDigest_Out)),
2681     /* offsetOfTypes */     offsetof(PolicyGetDigest_COMMAND_DESCRIPTOR_t, types),
2682     /* offsets       */     // No parameter offsets;
2683     /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2684                              END_OF_LIST,
2685                              TPM2B_DIGEST_P_MARSHAL,
2686                              END_OF_LIST}
2687 };
2688 #define _PolicyGetDigestDataAddress (&_PolicyGetDigestData)
2689 #else
2690 #define _PolicyGetDigestDataAddress 0
2691 #endif // CC_PolicyGetDigest
2692 #if CC_PolicyNvWritten
2693 #include "PolicyNvWritten_fp.h"
2694 typedef TPM_RC  (PolicyNvWritten_Entry)(
2695     PolicyNvWritten_In          *in
2696 );
2697 typedef const struct {
2698     PolicyNvWritten_Entry   *entry;
2699     UINT16                  inSize;
2700     UINT16                  outSize;
2701     UINT16                  offsetOfTypes;
2702     UINT16                  paramOffsets[1];
2703     BYTE                    types[4];
2704 } PolicyNvWritten_COMMAND_DESCRIPTOR_t;
2705 PolicyNvWritten_COMMAND_DESCRIPTOR_t _PolicyNvWrittenData = {
2706     /* entry         */     &TPM2_PolicyNvWritten,
2707     /* inSize        */     (UINT16)(sizeof(PolicyNvWritten_In)),
2708     /* outSize       */     0,
2709     /* offsetOfTypes */     offsetof(PolicyNvWritten_COMMAND_DESCRIPTOR_t, types),
2710     /* offsets       */     {(UINT16)(offsetof(PolicyNvWritten_In, writtenSet))},
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
2711     /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2712                              TPMI_YES_NO_P_UNMARSHAL,
2713                              END_OF_LIST,
2714                              END_OF_LIST}
2715 };
2716 #define _PolicyNvWrittenDataAddress (&_PolicyNvWrittenData)
2717 #else
2718 #define _PolicyNvWrittenDataAddress 0
2719 #endif // CC_PolicyNvWritten
2720 #if CC_PolicyTemplate
2721 #include "PolicyTemplate_fp.h"
2722 typedef TPM_RC  (PolicyTemplate_Entry)(
2723     PolicyTemplate_In           *in
2724 );
2725 typedef const struct {
2726     PolicyTemplate_Entry    *entry;
2727     UINT16                  inSize;
2728     UINT16                  outSize;
2729     UINT16                  offsetOfTypes;
2730     UINT16                  paramOffsets[1];
2731     BYTE                    types[4];
2732 } PolicyTemplate_COMMAND_DESCRIPTOR_t;
2733 PolicyTemplate_COMMAND_DESCRIPTOR_t _PolicyTemplateData = {
2734     /* entry         */     &TPM2_PolicyTemplate,
2735     /* inSize        */     (UINT16)(sizeof(PolicyTemplate_In)),
2736     /* outSize       */     0,
2737     /* offsetOfTypes */     offsetof(PolicyTemplate_COMMAND_DESCRIPTOR_t, types),
2738     /* offsets       */     {(UINT16)(offsetof(PolicyTemplate_In, templateHash))},
2739     /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2740                              TPM2B_DIGEST_P_UNMARSHAL,
2741                              END_OF_LIST,
2742                              END_OF_LIST}
2743 };
2744 #define _PolicyTemplateDataAddress (&_PolicyTemplateData)
2745 #else
2746 #define _PolicyTemplateDataAddress 0
2747 #endif // CC_PolicyTemplate
2748 #if CC_PolicyAuthorizeNV
2749 #include "PolicyAuthorizeNV_fp.h"
2750 typedef TPM_RC  (PolicyAuthorizeNV_Entry)(
2751     PolicyAuthorizeNV_In            *in
2752 );
2753 typedef const struct {
2754     PolicyAuthorizeNV_Entry     *entry;
2755     UINT16                      inSize;
2756     UINT16                      outSize;
2757     UINT16                      offsetOfTypes;
2758     UINT16                      paramOffsets[2];
2759     BYTE                        types[5];
2760 } PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t;
2761 PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t _PolicyAuthorizeNVData = {
2762     /* entry         */         &TPM2_PolicyAuthorizeNV,
2763     /* inSize        */         (UINT16)(sizeof(PolicyAuthorizeNV_In)),
2764     /* outSize       */         0,
2765     /* offsetOfTypes */         offsetof(PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t,
types),
2766     /* offsets       */         {(UINT16)(offsetof(PolicyAuthorizeNV_In, nvIndex)),
2767                                  (UINT16)(offsetof(PolicyAuthorizeNV_In,
policySession))},
2768     /* types         */         {TPMI_RH_NV_AUTH_H_UNMARSHAL,
2769                                  TPMI_RH_NV_INDEX_H_UNMARSHAL,
2770                                  TPMI_SH_POLICY_H_UNMARSHAL,
2771                                  END_OF_LIST,
2772                                  END_OF_LIST}
2773 };
2774 #define _PolicyAuthorizeNVDataAddress (&_PolicyAuthorizeNVData)
```

```
2775    #else
2776    #define _PolicyAuthorizeNVDataAddress 0
2777    #endif // CC_PolicyAuthorizeNV
2778    #if CC_CreatePrimary
2779    #include "CreatePrimary_fp.h"
2780    typedef TPM_RC  (CreatePrimary_Entry)(
2781        CreatePrimary_In            *in,
2782        CreatePrimary_Out           *out
2783    );
2784    typedef const struct {
2785        CreatePrimary_Entry     *entry;
2786        UINT16                  inSize;
2787        UINT16                  outSize;
2788        UINT16                  offsetOfTypes;
2789        UINT16                  paramOffsets[9];
2790        BYTE                    types[13];
2791    } CreatePrimary_COMMAND_DESCRIPTOR_t;
2792    CreatePrimary_COMMAND_DESCRIPTOR_t _CreatePrimaryData = {
2793        /* entry         */     &TPM2_CreatePrimary,
2794        /* inSize        */     (UINT16)(sizeof(CreatePrimary_In)),
2795        /* outSize       */     (UINT16)(sizeof(CreatePrimary_Out)),
2796        /* offsetOfTypes */     offsetof(CreatePrimary_COMMAND_DESCRIPTOR_t, types),
2797        /* offsets       */     {(UINT16)(offsetof(CreatePrimary_In, inSensitive)),
2798                                 (UINT16)(offsetof(CreatePrimary_In, inPublic)),
2799                                 (UINT16)(offsetof(CreatePrimary_In, outsideInfo)),
2800                                 (UINT16)(offsetof(CreatePrimary_In, creationPCR)),
2801                                 (UINT16)(offsetof(CreatePrimary_Out, outPublic)),
2802                                 (UINT16)(offsetof(CreatePrimary_Out, creationData)),
2803                                 (UINT16)(offsetof(CreatePrimary_Out, creationHash)),
2804                                 (UINT16)(offsetof(CreatePrimary_Out, creationTicket)),
2805                                 (UINT16)(offsetof(CreatePrimary_Out, name))},
2806        /* types         */     {TPMI_RH_HIERARCHY_H_UNMARSHAL + ADD_FLAG,
2807                                 TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
2808                                 TPM2B_PUBLIC_P_UNMARSHAL,
2809                                 TPM2B_DATA_P_UNMARSHAL,
2810                                 TPML_PCR_SELECTION_P_UNMARSHAL,
2811                                 END_OF_LIST,
2812                                 TPM_HANDLE_H_MARSHAL,
2813                                 TPM2B_PUBLIC_P_MARSHAL,
2814                                 TPM2B_CREATION_DATA_P_MARSHAL,
2815                                 TPM2B_DIGEST_P_MARSHAL,
2816                                 TPMT_TK_CREATION_P_MARSHAL,
2817                                 TPM2B_NAME_P_MARSHAL,
2818                                 END_OF_LIST}
2819    };
2820    #define _CreatePrimaryDataAddress (&_CreatePrimaryData)
2821    #else
2822    #define _CreatePrimaryDataAddress 0
2823    #endif // CC_CreatePrimary
2824    #if CC_HierarchyControl
2825    #include "HierarchyControl_fp.h"
2826    typedef TPM_RC  (HierarchyControl_Entry)(
2827        HierarchyControl_In         *in
2828    );
2829    typedef const struct {
2830        HierarchyControl_Entry  *entry;
2831        UINT16                  inSize;
2832        UINT16                  outSize;
2833        UINT16                  offsetOfTypes;
2834        UINT16                  paramOffsets[2];
2835        BYTE                    types[5];
2836    } HierarchyControl_COMMAND_DESCRIPTOR_t;
2837    HierarchyControl_COMMAND_DESCRIPTOR_t _HierarchyControlData = {
2838        /* entry         */     &TPM2_HierarchyControl,
2839        /* inSize        */     (UINT16)(sizeof(HierarchyControl_In)),
2840        /* outSize       */     0,
```

```
2841      /* offsetOfTypes */      offsetof(HierarchyControl_COMMAND_DESCRIPTOR_t, types),
2842      /* offsets       */      {(UINT16)(offsetof(HierarchyControl_In, enable)),
2843                                (UINT16)(offsetof(HierarchyControl_In, state))},
2844      /* types         */      {TPMI_RH_HIERARCHY_H_UNMARSHAL,
2845                                TPMI_RH_ENABLES_P_UNMARSHAL,
2846                                TPMI_YES_NO_P_UNMARSHAL,
2847                                END_OF_LIST,
2848                                END_OF_LIST}
2849  };
2850  #define _HierarchyControlDataAddress (&_HierarchyControlData)
2851  #else
2852  #define _HierarchyControlDataAddress 0
2853  #endif // CC_HierarchyControl
2854  #if CC_SetPrimaryPolicy
2855  #include "SetPrimaryPolicy_fp.h"
2856  typedef TPM_RC  (SetPrimaryPolicy_Entry)(
2857      SetPrimaryPolicy_In          *in
2858  );
2859  typedef const struct {
2860      SetPrimaryPolicy_Entry   *entry;
2861      UINT16                   inSize;
2862      UINT16                   outSize;
2863      UINT16                   offsetOfTypes;
2864      UINT16                   paramOffsets[2];
2865      BYTE                     types[5];
2866  } SetPrimaryPolicy_COMMAND_DESCRIPTOR_t;
2867  SetPrimaryPolicy_COMMAND_DESCRIPTOR_t _SetPrimaryPolicyData = {
2868      /* entry         */      &TPM2_SetPrimaryPolicy,
2869      /* inSize        */      (UINT16)(sizeof(SetPrimaryPolicy_In)),
2870      /* outSize       */      0,
2871      /* offsetOfTypes */      offsetof(SetPrimaryPolicy_COMMAND_DESCRIPTOR_t, types),
2872      /* offsets       */      {(UINT16)(offsetof(SetPrimaryPolicy_In, authPolicy)),
2873                                (UINT16)(offsetof(SetPrimaryPolicy_In, hashAlg))},
2874      /* types         */      {TPMI_RH_HIERARCHY_POLICY_H_UNMARSHAL,
2875                                TPM2B_DIGEST_P_UNMARSHAL,
2876                                TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
2877                                END_OF_LIST,
2878                                END_OF_LIST}
2879  };
2880  #define _SetPrimaryPolicyDataAddress (&_SetPrimaryPolicyData)
2881  #else
2882  #define _SetPrimaryPolicyDataAddress 0
2883  #endif // CC_SetPrimaryPolicy
2884  #if CC_ChangePPS
2885  #include "ChangePPS_fp.h"
2886  typedef TPM_RC  (ChangePPS_Entry)(
2887      ChangePPS_In                 *in
2888  );
2889  typedef const struct {
2890      ChangePPS_Entry          *entry;
2891      UINT16                   inSize;
2892      UINT16                   outSize;
2893      UINT16                   offsetOfTypes;
2894      BYTE                     types[3];
2895  } ChangePPS_COMMAND_DESCRIPTOR_t;
2896  ChangePPS_COMMAND_DESCRIPTOR_t _ChangePPSData = {
2897      /* entry         */      &TPM2_ChangePPS,
2898      /* inSize        */      (UINT16)(sizeof(ChangePPS_In)),
2899      /* outSize       */      0,
2900      /* offsetOfTypes */      offsetof(ChangePPS_COMMAND_DESCRIPTOR_t, types),
2901      /* offsets       */      // No parameter offsets;
2902      /* types         */      {TPMI_RH_PLATFORM_H_UNMARSHAL,
2903                                END_OF_LIST,
2904                                END_OF_LIST}
2905  };
2906  #define _ChangePPSDataAddress (&_ChangePPSData)
```

```
2907 #else
2908 #define _ChangePPSDataAddress 0
2909 #endif // CC_ChangePPS
2910 #if CC_ChangeEPS
2911 #include "ChangeEPS_fp.h"
2912 typedef TPM_RC  (ChangeEPS_Entry)(
2913     ChangeEPS_In                *in
2914 );
2915 typedef const struct {
2916     ChangeEPS_Entry          *entry;
2917     UINT16                   inSize;
2918     UINT16                   outSize;
2919     UINT16                   offsetOfTypes;
2920     BYTE                     types[3];
2921 } ChangeEPS_COMMAND_DESCRIPTOR_t;
2922 ChangeEPS_COMMAND_DESCRIPTOR_t _ChangeEPSData = {
2923     /* entry         */      &TPM2_ChangeEPS,
2924     /* inSize        */      (UINT16)(sizeof(ChangeEPS_In)),
2925     /* outSize       */      0,
2926     /* offsetOfTypes */      offsetof(ChangeEPS_COMMAND_DESCRIPTOR_t, types),
2927     /* offsets       */      // No parameter offsets;
2928     /* types         */      {TPMI_RH_PLATFORM_H_UNMARSHAL,
2929                               END_OF_LIST,
2930                               END_OF_LIST}
2931 };
2932 #define _ChangeEPSDataAddress (&_ChangeEPSData)
2933 #else
2934 #define _ChangeEPSDataAddress 0
2935 #endif // CC_ChangeEPS
2936 #if CC_Clear
2937 #include "Clear_fp.h"
2938 typedef TPM_RC  (Clear_Entry)(
2939     Clear_In                    *in
2940 );
2941 typedef const struct {
2942     Clear_Entry              *entry;
2943     UINT16                   inSize;
2944     UINT16                   outSize;
2945     UINT16                   offsetOfTypes;
2946     BYTE                     types[3];
2947 } Clear_COMMAND_DESCRIPTOR_t;
2948 Clear_COMMAND_DESCRIPTOR_t _ClearData = {
2949     /* entry         */      &TPM2_Clear,
2950     /* inSize        */      (UINT16)(sizeof(Clear_In)),
2951     /* outSize       */      0,
2952     /* offsetOfTypes */      offsetof(Clear_COMMAND_DESCRIPTOR_t, types),
2953     /* offsets       */      // No parameter offsets;
2954     /* types         */      {TPMI_RH_CLEAR_H_UNMARSHAL,
2955                               END_OF_LIST,
2956                               END_OF_LIST}
2957 };
2958 #define _ClearDataAddress (&_ClearData)
2959 #else
2960 #define _ClearDataAddress 0
2961 #endif // CC_Clear
2962 #if CC_ClearControl
2963 #include "ClearControl_fp.h"
2964 typedef TPM_RC  (ClearControl_Entry)(
2965     ClearControl_In             *in
2966 );
2967 typedef const struct {
2968     ClearControl_Entry       *entry;
2969     UINT16                   inSize;
2970     UINT16                   outSize;
2971     UINT16                   offsetOfTypes;
2972     UINT16                   paramOffsets[1];
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
2973        BYTE                    types[4];
2974    } ClearControl_COMMAND_DESCRIPTOR_t;
2975    ClearControl_COMMAND_DESCRIPTOR_t _ClearControlData = {
2976        /* entry         */      &TPM2_ClearControl,
2977        /* inSize        */      (UINT16)(sizeof(ClearControl_In)),
2978        /* outSize       */      0,
2979        /* offsetOfTypes */      offsetof(ClearControl_COMMAND_DESCRIPTOR_t, types),
2980        /* offsets       */      {(UINT16)(offsetof(ClearControl_In, disable))},
2981        /* types         */      {TPMI_RH_CLEAR_H_UNMARSHAL,
2982                                  TPMI_YES_NO_P_UNMARSHAL,
2983                                  END_OF_LIST,
2984                                  END_OF_LIST}
2985    };
2986    #define _ClearControlDataAddress (&_ClearControlData)
2987    #else
2988    #define _ClearControlDataAddress 0
2989    #endif // CC_ClearControl
2990    #if CC_HierarchyChangeAuth
2991    #include "HierarchyChangeAuth_fp.h"
2992    typedef TPM_RC  (HierarchyChangeAuth_Entry)(
2993        HierarchyChangeAuth_In          *in
2994    );
2995    typedef const struct {
2996        HierarchyChangeAuth_Entry   *entry;
2997        UINT16                      inSize;
2998        UINT16                      outSize;
2999        UINT16                      offsetOfTypes;
3000        UINT16                      paramOffsets[1];
3001        BYTE                        types[4];
3002    } HierarchyChangeAuth_COMMAND_DESCRIPTOR_t;
3003    HierarchyChangeAuth_COMMAND_DESCRIPTOR_t _HierarchyChangeAuthData = {
3004        /* entry         */          &TPM2_HierarchyChangeAuth,
3005        /* inSize        */          (UINT16)(sizeof(HierarchyChangeAuth_In)),
3006        /* outSize       */          0,
3007        /* offsetOfTypes */          offsetof(HierarchyChangeAuth_COMMAND_DESCRIPTOR_t,
        types),
3008        /* offsets       */          {(UINT16)(offsetof(HierarchyChangeAuth_In, newAuth))},
3009        /* types         */          {TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL,
3010                                      TPM2B_AUTH_P_UNMARSHAL,
3011                                      END_OF_LIST,
3012                                      END_OF_LIST}
3013    };
3014    #define _HierarchyChangeAuthDataAddress (&_HierarchyChangeAuthData)
3015    #else
3016    #define _HierarchyChangeAuthDataAddress 0
3017    #endif // CC_HierarchyChangeAuth
3018    #if CC_DictionaryAttackLockReset
3019    #include "DictionaryAttackLockReset_fp.h"
3020    typedef TPM_RC  (DictionaryAttackLockReset_Entry)(
3021        DictionaryAttackLockReset_In            *in
3022    );
3023    typedef const struct {
3024        DictionaryAttackLockReset_Entry     *entry;
3025        UINT16                              inSize;
3026        UINT16                              outSize;
3027        UINT16                              offsetOfTypes;
3028        BYTE                                types[3];
3029    } DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t;
3030    DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t _DictionaryAttackLockResetData = {
3031        /* entry         */                  &TPM2_DictionaryAttackLockReset,
3032        /* inSize        */
        (UINT16)(sizeof(DictionaryAttackLockReset_In)),
3033        /* outSize       */                  0,
3034        /* offsetOfTypes */
        offsetof(DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t, types),
3035        /* offsets       */                  // No parameter offsets;
```

```
3036        /* types         */                    {TPMI_RH_LOCKOUT_H_UNMARSHAL,
3037                                                 END_OF_LIST,
3038                                                 END_OF_LIST}
3039    };
3040    #define _DictionaryAttackLockResetDataAddress (&_DictionaryAttackLockResetData)
3041    #else
3042    #define _DictionaryAttackLockResetDataAddress 0
3043    #endif // CC_DictionaryAttackLockReset
3044    #if CC_DictionaryAttackParameters
3045    #include "DictionaryAttackParameters_fp.h"
3046    typedef TPM_RC  (DictionaryAttackParameters_Entry)(
3047        DictionaryAttackParameters_In           *in
3048    );
3049    typedef const struct {
3050        DictionaryAttackParameters_Entry    *entry;
3051        UINT16                              inSize;
3052        UINT16                              outSize;
3053        UINT16                              offsetOfTypes;
3054        UINT16                              paramOffsets[3];
3055        BYTE                                types[6];
3056    } DictionaryAttackParameters_COMMAND_DESCRIPTOR_t;
3057    DictionaryAttackParameters_COMMAND_DESCRIPTOR_t _DictionaryAttackParametersData =
    {
3058        /* entry         */                    &TPM2_DictionaryAttackParameters,
3059        /* inSize        */
    (UINT16)(sizeof(DictionaryAttackParameters_In)),
3060        /* outSize       */                    0,
3061        /* offsetOfTypes */
    offsetof(DictionaryAttackParameters_COMMAND_DESCRIPTOR_t, types),
3062        /* offsets       */
    {(UINT16)(offsetof(DictionaryAttackParameters_In, newMaxTries)),
3063
    (UINT16)(offsetof(DictionaryAttackParameters_In, newRecoveryTime)),
3064
    (UINT16)(offsetof(DictionaryAttackParameters_In, lockoutRecovery))},
3065        /* types         */                    {TPMI_RH_LOCKOUT_H_UNMARSHAL,
3066                                                 UINT32_P_UNMARSHAL,
3067                                                 UINT32_P_UNMARSHAL,
3068                                                 UINT32_P_UNMARSHAL,
3069                                                 END_OF_LIST,
3070                                                 END_OF_LIST}
3071    };
3072    #define _DictionaryAttackParametersDataAddress (&_DictionaryAttackParametersData)
3073    #else
3074    #define _DictionaryAttackParametersDataAddress 0
3075    #endif // CC_DictionaryAttackParameters
3076    #if CC_PP_Commands
3077    #include "PP_Commands_fp.h"
3078    typedef TPM_RC  (PP_Commands_Entry)(
3079        PP_Commands_In              *in
3080    );
3081    typedef const struct {
3082        PP_Commands_Entry       *entry;
3083        UINT16                  inSize;
3084        UINT16                  outSize;
3085        UINT16                  offsetOfTypes;
3086        UINT16                  paramOffsets[2];
3087        BYTE                    types[5];
3088    } PP_Commands_COMMAND_DESCRIPTOR_t;
3089    PP_Commands_COMMAND_DESCRIPTOR_t _PP_CommandsData = {
3090        /* entry         */      &TPM2_PP_Commands,
3091        /* inSize        */      (UINT16)(sizeof(PP_Commands_In)),
3092        /* outSize       */      0,
3093        /* offsetOfTypes */      offsetof(PP_Commands_COMMAND_DESCRIPTOR_t, types),
3094        /* offsets       */      {(UINT16)(offsetof(PP_Commands_In, setList)),
3095                                  (UINT16)(offsetof(PP_Commands_In, clearList))},
```


 Copyright © TCG 2006-2020

```
3096     /* types          */      {TPMI_RH_PLATFORM_H_UNMARSHAL,
3097                                TPML_CC_P_UNMARSHAL,
3098                                TPML_CC_P_UNMARSHAL,
3099                                END_OF_LIST,
3100                                END_OF_LIST}
3101 };
3102 #define _PP_CommandsDataAddress (&_PP_CommandsData)
3103 #else
3104 #define _PP_CommandsDataAddress 0
3105 #endif // CC_PP_Commands
3106 #if CC_SetAlgorithmSet
3107 #include "SetAlgorithmSet_fp.h"
3108 typedef TPM_RC  (SetAlgorithmSet_Entry)(
3109     SetAlgorithmSet_In          *in
3110 );
3111 typedef const struct {
3112     SetAlgorithmSet_Entry   *entry;
3113     UINT16                  inSize;
3114     UINT16                  outSize;
3115     UINT16                  offsetOfTypes;
3116     UINT16                  paramOffsets[1];
3117     BYTE                    types[4];
3118 } SetAlgorithmSet_COMMAND_DESCRIPTOR_t;
3119 SetAlgorithmSet_COMMAND_DESCRIPTOR_t _SetAlgorithmSetData = {
3120     /* entry          */      &TPM2_SetAlgorithmSet,
3121     /* inSize         */      (UINT16)(sizeof(SetAlgorithmSet_In)),
3122     /* outSize        */      0,
3123     /* offsetOfTypes */       offsetof(SetAlgorithmSet_COMMAND_DESCRIPTOR_t, types),
3124     /* offsets        */      {(UINT16)(offsetof(SetAlgorithmSet_In, algorithmSet))},
3125     /* types          */      {TPMI_RH_PLATFORM_H_UNMARSHAL,
3126                                UINT32_P_UNMARSHAL,
3127                                END_OF_LIST,
3128                                END_OF_LIST}
3129 };
3130 #define _SetAlgorithmSetDataAddress (&_SetAlgorithmSetData)
3131 #else
3132 #define _SetAlgorithmSetDataAddress 0
3133 #endif // CC_SetAlgorithmSet
3134 #if CC_FieldUpgradeStart
3135 #include "FieldUpgradeStart_fp.h"
3136 typedef TPM_RC  (FieldUpgradeStart_Entry)(
3137     FieldUpgradeStart_In            *in
3138 );
3139 typedef const struct {
3140     FieldUpgradeStart_Entry     *entry;
3141     UINT16                      inSize;
3142     UINT16                      outSize;
3143     UINT16                      offsetOfTypes;
3144     UINT16                      paramOffsets[3];
3145     BYTE                        types[6];
3146 } FieldUpgradeStart_COMMAND_DESCRIPTOR_t;
3147 FieldUpgradeStart_COMMAND_DESCRIPTOR_t _FieldUpgradeStartData = {
3148     /* entry          */          &TPM2_FieldUpgradeStart,
3149     /* inSize         */          (UINT16)(sizeof(FieldUpgradeStart_In)),
3150     /* outSize        */          0,
3151     /* offsetOfTypes */           offsetof(FieldUpgradeStart_COMMAND_DESCRIPTOR_t,
     types),
3152     /* offsets        */          {(UINT16)(offsetof(FieldUpgradeStart_In, keyHandle)),
3153                                    (UINT16)(offsetof(FieldUpgradeStart_In, fuDigest)),
3154                                    (UINT16)(offsetof(FieldUpgradeStart_In,
     manifestSignature))},
3155     /* types          */          {TPMI_RH_PLATFORM_H_UNMARSHAL,
3156                                    TPMI_DH_OBJECT_H_UNMARSHAL,
3157                                    TPM2B_DIGEST_P_UNMARSHAL,
3158                                    TPMT_SIGNATURE_P_UNMARSHAL,
3159                                    END_OF_LIST,
```

```
3160                                  END_OF_LIST}
3161    };
3162    #define _FieldUpgradeStartDataAddress (&_FieldUpgradeStartData)
3163    #else
3164    #define _FieldUpgradeStartDataAddress 0
3165    #endif // CC_FieldUpgradeStart
3166    #if CC_FieldUpgradeData
3167    #include "FieldUpgradeData_fp.h"
3168    typedef TPM_RC  (FieldUpgradeData_Entry)(
3169        FieldUpgradeData_In          *in,
3170        FieldUpgradeData_Out         *out
3171    );
3172    typedef const struct {
3173        FieldUpgradeData_Entry   *entry;
3174        UINT16                   inSize;
3175        UINT16                   outSize;
3176        UINT16                   offsetOfTypes;
3177        UINT16                   paramOffsets[1];
3178        BYTE                     types[5];
3179    } FieldUpgradeData_COMMAND_DESCRIPTOR_t;
3180    FieldUpgradeData_COMMAND_DESCRIPTOR_t _FieldUpgradeDataData = {
3181        /* entry          */     &TPM2_FieldUpgradeData,
3182        /* inSize         */     (UINT16)(sizeof(FieldUpgradeData_In)),
3183        /* outSize        */     (UINT16)(sizeof(FieldUpgradeData_Out)),
3184        /* offsetOfTypes */      offsetof(FieldUpgradeData_COMMAND_DESCRIPTOR_t, types),
3185        /* offsets        */     {(UINT16)(offsetof(FieldUpgradeData_Out, firstDigest))},
3186        /* types          */     {TPM2B_MAX_BUFFER_P_UNMARSHAL,
3187                                  END_OF_LIST,
3188                                  TPMT_HA_P_MARSHAL,
3189                                  TPMT_HA_P_MARSHAL,
3190                                  END_OF_LIST}
3191    };
3192    #define _FieldUpgradeDataDataAddress (&_FieldUpgradeDataData)
3193    #else
3194    #define _FieldUpgradeDataDataAddress 0
3195    #endif // CC_FieldUpgradeData
3196    #if CC_FirmwareRead
3197    #include "FirmwareRead_fp.h"
3198    typedef TPM_RC  (FirmwareRead_Entry)(
3199        FirmwareRead_In              *in,
3200        FirmwareRead_Out             *out
3201    );
3202    typedef const struct {
3203        FirmwareRead_Entry       *entry;
3204        UINT16                   inSize;
3205        UINT16                   outSize;
3206        UINT16                   offsetOfTypes;
3207        BYTE                     types[4];
3208    } FirmwareRead_COMMAND_DESCRIPTOR_t;
3209    FirmwareRead_COMMAND_DESCRIPTOR_t _FirmwareReadData = {
3210        /* entry          */     &TPM2_FirmwareRead,
3211        /* inSize         */     (UINT16)(sizeof(FirmwareRead_In)),
3212        /* outSize        */     (UINT16)(sizeof(FirmwareRead_Out)),
3213        /* offsetOfTypes */      offsetof(FirmwareRead_COMMAND_DESCRIPTOR_t, types),
3214        /* offsets        */     // No parameter offsets;
3215        /* types          */     {UINT32_P_UNMARSHAL,
3216                                  END_OF_LIST,
3217                                  TPM2B_MAX_BUFFER_P_MARSHAL,
3218                                  END_OF_LIST}
3219    };
3220    #define _FirmwareReadDataAddress (&_FirmwareReadData)
3221    #else
3222    #define _FirmwareReadDataAddress 0
3223    #endif // CC_FirmwareRead
3224    #if CC_ContextSave
3225    #include "ContextSave_fp.h"
```

 Copyright © TCG 2006-2020

```
3226  typedef TPM_RC  (ContextSave_Entry)(
3227      ContextSave_In              *in,
3228      ContextSave_Out             *out
3229  );
3230  typedef const struct {
3231      ContextSave_Entry       *entry;
3232      UINT16                  inSize;
3233      UINT16                  outSize;
3234      UINT16                  offsetOfTypes;
3235      BYTE                    types[4];
3236  } ContextSave_COMMAND_DESCRIPTOR_t;
3237  ContextSave_COMMAND_DESCRIPTOR_t _ContextSaveData = {
3238      /* entry         */     &TPM2_ContextSave,
3239      /* inSize        */     (UINT16)(sizeof(ContextSave_In)),
3240      /* outSize       */     (UINT16)(sizeof(ContextSave_Out)),
3241      /* offsetOfTypes */     offsetof(ContextSave_COMMAND_DESCRIPTOR_t, types),
3242      /* offsets       */     // No parameter offsets;
3243      /* types         */     {TPMI_DH_CONTEXT_H_UNMARSHAL,
3244                               END_OF_LIST,
3245                               TPMS_CONTEXT_P_MARSHAL,
3246                               END_OF_LIST}
3247  };
3248  #define _ContextSaveDataAddress (&_ContextSaveData)
3249  #else
3250  #define _ContextSaveDataAddress 0
3251  #endif // CC_ContextSave
3252  #if CC_ContextLoad
3253  #include "ContextLoad_fp.h"
3254  typedef TPM_RC  (ContextLoad_Entry)(
3255      ContextLoad_In              *in,
3256      ContextLoad_Out             *out
3257  );
3258  typedef const struct {
3259      ContextLoad_Entry       *entry;
3260      UINT16                  inSize;
3261      UINT16                  outSize;
3262      UINT16                  offsetOfTypes;
3263      BYTE                    types[4];
3264  } ContextLoad_COMMAND_DESCRIPTOR_t;
3265  ContextLoad_COMMAND_DESCRIPTOR_t _ContextLoadData = {
3266      /* entry         */     &TPM2_ContextLoad,
3267      /* inSize        */     (UINT16)(sizeof(ContextLoad_In)),
3268      /* outSize       */     (UINT16)(sizeof(ContextLoad_Out)),
3269      /* offsetOfTypes */     offsetof(ContextLoad_COMMAND_DESCRIPTOR_t, types),
3270      /* offsets       */     // No parameter offsets;
3271      /* types         */     {TPMS_CONTEXT_P_UNMARSHAL,
3272                               END_OF_LIST,
3273                               TPMI_DH_CONTEXT_H_MARSHAL,
3274                               END_OF_LIST}
3275  };
3276  #define _ContextLoadDataAddress (&_ContextLoadData)
3277  #else
3278  #define _ContextLoadDataAddress 0
3279  #endif // CC_ContextLoad
3280  #if CC_FlushContext
3281  #include "FlushContext_fp.h"
3282  typedef TPM_RC  (FlushContext_Entry)(
3283      FlushContext_In             *in
3284  );
3285  typedef const struct {
3286      FlushContext_Entry      *entry;
3287      UINT16                  inSize;
3288      UINT16                  outSize;
3289      UINT16                  offsetOfTypes;
3290      BYTE                    types[3];
3291  } FlushContext_COMMAND_DESCRIPTOR_t;
```

```
3292 FlushContext_COMMAND_DESCRIPTOR_t _FlushContextData = {
3293     /* entry         */     &TPM2_FlushContext,
3294     /* inSize        */     (UINT16)(sizeof(FlushContext_In)),
3295     /* outSize       */     0,
3296     /* offsetOfTypes */     offsetof(FlushContext_COMMAND_DESCRIPTOR_t, types),
3297     /* offsets       */     // No parameter offsets;
3298     /* types         */     {TPMI_DH_CONTEXT_P_UNMARSHAL,
3299                              END_OF_LIST,
3300                              END_OF_LIST}
3301 };
3302 #define _FlushContextDataAddress (&_FlushContextData)
3303 #else
3304 #define _FlushContextDataAddress 0
3305 #endif // CC_FlushContext
3306 #if CC_EvictControl
3307 #include "EvictControl_fp.h"
3308 typedef TPM_RC  (EvictControl_Entry)(
3309     EvictControl_In              *in
3310 );
3311 typedef const struct {
3312     EvictControl_Entry      *entry;
3313     UINT16                  inSize;
3314     UINT16                  outSize;
3315     UINT16                  offsetOfTypes;
3316     UINT16                  paramOffsets[2];
3317     BYTE                    types[5];
3318 } EvictControl_COMMAND_DESCRIPTOR_t;
3319 EvictControl_COMMAND_DESCRIPTOR_t _EvictControlData = {
3320     /* entry         */     &TPM2_EvictControl,
3321     /* inSize        */     (UINT16)(sizeof(EvictControl_In)),
3322     /* outSize       */     0,
3323     /* offsetOfTypes */     offsetof(EvictControl_COMMAND_DESCRIPTOR_t, types),
3324     /* offsets       */     {(UINT16)(offsetof(EvictControl_In, objectHandle)),
3325                              (UINT16)(offsetof(EvictControl_In, persistentHandle))},
3326     /* types         */     {TPMI_RH_PROVISION_H_UNMARSHAL,
3327                              TPMI_DH_OBJECT_H_UNMARSHAL,
3328                              TPMI_DH_PERSISTENT_P_UNMARSHAL,
3329                              END_OF_LIST,
3330                              END_OF_LIST}
3331 };
3332 #define _EvictControlDataAddress (&_EvictControlData)
3333 #else
3334 #define _EvictControlDataAddress 0
3335 #endif // CC_EvictControl
3336 #if CC_ReadClock
3337 #include "ReadClock_fp.h"
3338 typedef TPM_RC  (ReadClock_Entry)(
3339     ReadClock_Out                *out
3340 );
3341 typedef const struct {
3342     ReadClock_Entry         *entry;
3343     UINT16                  inSize;
3344     UINT16                  outSize;
3345     UINT16                  offsetOfTypes;
3346     BYTE                    types[3];
3347 } ReadClock_COMMAND_DESCRIPTOR_t;
3348 ReadClock_COMMAND_DESCRIPTOR_t _ReadClockData = {
3349     /* entry         */     &TPM2_ReadClock,
3350     /* inSize        */     0,
3351     /* outSize       */     (UINT16)(sizeof(ReadClock_Out)),
3352     /* offsetOfTypes */     offsetof(ReadClock_COMMAND_DESCRIPTOR_t, types),
3353     /* offsets       */     // No parameter offsets;
3354     /* types         */     {END_OF_LIST,
3355                              TPMS_TIME_INFO_P_MARSHAL,
3356                              END_OF_LIST}
3357 };
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
3358 #define _ReadClockDataAddress (&_ReadClockData)
3359 #else
3360 #define _ReadClockDataAddress 0
3361 #endif // CC_ReadClock
3362 #if CC_ClockSet
3363 #include "ClockSet_fp.h"
3364 typedef TPM_RC  (ClockSet_Entry)(
3365     ClockSet_In                  *in
3366 );
3367 typedef const struct {
3368     ClockSet_Entry           *entry;
3369     UINT16                   inSize;
3370     UINT16                   outSize;
3371     UINT16                   offsetOfTypes;
3372     UINT16                   paramOffsets[1];
3373     BYTE                     types[4];
3374 } ClockSet_COMMAND_DESCRIPTOR_t;
3375 ClockSet_COMMAND_DESCRIPTOR_t _ClockSetData = {
3376     /* entry         */     &TPM2_ClockSet,
3377     /* inSize        */     (UINT16)(sizeof(ClockSet_In)),
3378     /* outSize       */     0,
3379     /* offsetOfTypes */     offsetof(ClockSet_COMMAND_DESCRIPTOR_t, types),
3380     /* offsets       */     {(UINT16)(offsetof(ClockSet_In, newTime))},
3381     /* types         */     {TPMI_RH_PROVISION_H_UNMARSHAL,
3382                              UINT64_P_UNMARSHAL,
3383                              END_OF_LIST,
3384                              END_OF_LIST}
3385 };
3386 #define _ClockSetDataAddress (&_ClockSetData)
3387 #else
3388 #define _ClockSetDataAddress 0
3389 #endif // CC_ClockSet
3390 #if CC_ClockRateAdjust
3391 #include "ClockRateAdjust_fp.h"
3392 typedef TPM_RC  (ClockRateAdjust_Entry)(
3393     ClockRateAdjust_In           *in
3394 );
3395 typedef const struct {
3396     ClockRateAdjust_Entry    *entry;
3397     UINT16                   inSize;
3398     UINT16                   outSize;
3399     UINT16                   offsetOfTypes;
3400     UINT16                   paramOffsets[1];
3401     BYTE                     types[4];
3402 } ClockRateAdjust_COMMAND_DESCRIPTOR_t;
3403 ClockRateAdjust_COMMAND_DESCRIPTOR_t _ClockRateAdjustData = {
3404     /* entry         */     &TPM2_ClockRateAdjust,
3405     /* inSize        */     (UINT16)(sizeof(ClockRateAdjust_In)),
3406     /* outSize       */     0,
3407     /* offsetOfTypes */     offsetof(ClockRateAdjust_COMMAND_DESCRIPTOR_t, types),
3408     /* offsets       */     {(UINT16)(offsetof(ClockRateAdjust_In, rateAdjust))},
3409     /* types         */     {TPMI_RH_PROVISION_H_UNMARSHAL,
3410                              TPM_CLOCK_ADJUST_P_UNMARSHAL,
3411                              END_OF_LIST,
3412                              END_OF_LIST}
3413 };
3414 #define _ClockRateAdjustDataAddress (&_ClockRateAdjustData)
3415 #else
3416 #define _ClockRateAdjustDataAddress 0
3417 #endif // CC_ClockRateAdjust
3418 #if CC_GetCapability
3419 #include "GetCapability_fp.h"
3420 typedef TPM_RC  (GetCapability_Entry)(
3421     GetCapability_In             *in,
3422     GetCapability_Out            *out
3423 );
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
3424 typedef const struct {
3425     GetCapability_Entry      *entry;
3426     UINT16                   inSize;
3427     UINT16                   outSize;
3428     UINT16                   offsetOfTypes;
3429     UINT16                   paramOffsets[3];
3430     BYTE                     types[7];
3431 } GetCapability_COMMAND_DESCRIPTOR_t;
3432 GetCapability_COMMAND_DESCRIPTOR_t _GetCapabilityData = {
3433     /* entry         */      &TPM2_GetCapability,
3434     /* inSize        */      (UINT16)(sizeof(GetCapability_In)),
3435     /* outSize       */      (UINT16)(sizeof(GetCapability_Out)),
3436     /* offsetOfTypes */      offsetof(GetCapability_COMMAND_DESCRIPTOR_t, types),
3437     /* offsets       */      {(UINT16)(offsetof(GetCapability_In, property)),
3438                               (UINT16)(offsetof(GetCapability_In, propertyCount)),
3439                               (UINT16)(offsetof(GetCapability_Out, capabilityData))},
3440     /* types         */      {TPM_CAP_P_UNMARSHAL,
3441                               UINT32_P_UNMARSHAL,
3442                               UINT32_P_UNMARSHAL,
3443                               END_OF_LIST,
3444                               TPMI_YES_NO_P_MARSHAL,
3445                               TPMS_CAPABILITY_DATA_P_MARSHAL,
3446                               END_OF_LIST}
3447 };
3448 #define _GetCapabilityDataAddress (&_GetCapabilityData)
3449 #else
3450 #define _GetCapabilityDataAddress 0
3451 #endif // CC_GetCapability
3452 #if CC_TestParms
3453 #include "TestParms_fp.h"
3454 typedef TPM_RC  (TestParms_Entry)(
3455     TestParms_In                 *in
3456 );
3457 typedef const struct {
3458     TestParms_Entry          *entry;
3459     UINT16                   inSize;
3460     UINT16                   outSize;
3461     UINT16                   offsetOfTypes;
3462     BYTE                     types[3];
3463 } TestParms_COMMAND_DESCRIPTOR_t;
3464 TestParms_COMMAND_DESCRIPTOR_t _TestParmsData = {
3465     /* entry         */      &TPM2_TestParms,
3466     /* inSize        */      (UINT16)(sizeof(TestParms_In)),
3467     /* outSize       */      0,
3468     /* offsetOfTypes */      offsetof(TestParms_COMMAND_DESCRIPTOR_t, types),
3469     /* offsets       */      // No parameter offsets;
3470     /* types         */      {TPMT_PUBLIC_PARMS_P_UNMARSHAL,
3471                               END_OF_LIST,
3472                               END_OF_LIST}
3473 };
3474 #define _TestParmsDataAddress (&_TestParmsData)
3475 #else
3476 #define _TestParmsDataAddress 0
3477 #endif // CC_TestParms
3478 #if CC_NV_DefineSpace
3479 #include "NV_DefineSpace_fp.h"
3480 typedef TPM_RC  (NV_DefineSpace_Entry)(
3481     NV_DefineSpace_In            *in
3482 );
3483 typedef const struct {
3484     NV_DefineSpace_Entry     *entry;
3485     UINT16                   inSize;
3486     UINT16                   outSize;
3487     UINT16                   offsetOfTypes;
3488     UINT16                   paramOffsets[2];
3489     BYTE                     types[5];
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
3490 } NV_DefineSpace_COMMAND_DESCRIPTOR_t;
3491 NV_DefineSpace_COMMAND_DESCRIPTOR_t _NV_DefineSpaceData = {
3492     /* entry         */     &TPM2_NV_DefineSpace,
3493     /* inSize        */     (UINT16)(sizeof(NV_DefineSpace_In)),
3494     /* outSize       */     0,
3495     /* offsetOfTypes */     offsetof(NV_DefineSpace_COMMAND_DESCRIPTOR_t, types),
3496     /* offsets       */     {(UINT16)(offsetof(NV_DefineSpace_In, auth)),
3497                              (UINT16)(offsetof(NV_DefineSpace_In, publicInfo))},
3498     /* types         */     {TPMI_RH_PROVISION_H_UNMARSHAL,
3499                              TPM2B_AUTH_P_UNMARSHAL,
3500                              TPM2B_NV_PUBLIC_P_UNMARSHAL,
3501                              END_OF_LIST,
3502                              END_OF_LIST}
3503 };
3504 #define _NV_DefineSpaceDataAddress (&_NV_DefineSpaceData)
3505 #else
3506 #define _NV_DefineSpaceDataAddress 0
3507 #endif // CC_NV_DefineSpace
3508 #if CC_NV_UndefineSpace
3509 #include "NV_UndefineSpace_fp.h"
3510 typedef TPM_RC  (NV_UndefineSpace_Entry)(
3511     NV_UndefineSpace_In          *in
3512 );
3513 typedef const struct {
3514     NV_UndefineSpace_Entry   *entry;
3515     UINT16                   inSize;
3516     UINT16                   outSize;
3517     UINT16                   offsetOfTypes;
3518     UINT16                   paramOffsets[1];
3519     BYTE                     types[4];
3520 } NV_UndefineSpace_COMMAND_DESCRIPTOR_t;
3521 NV_UndefineSpace_COMMAND_DESCRIPTOR_t _NV_UndefineSpaceData = {
3522     /* entry         */     &TPM2_NV_UndefineSpace,
3523     /* inSize        */     (UINT16)(sizeof(NV_UndefineSpace_In)),
3524     /* outSize       */     0,
3525     /* offsetOfTypes */     offsetof(NV_UndefineSpace_COMMAND_DESCRIPTOR_t, types),
3526     /* offsets       */     {(UINT16)(offsetof(NV_UndefineSpace_In, nvIndex))},
3527     /* types         */     {TPMI_RH_PROVISION_H_UNMARSHAL,
3528                              TPMI_RH_NV_INDEX_H_UNMARSHAL,
3529                              END_OF_LIST,
3530                              END_OF_LIST}
3531 };
3532 #define _NV_UndefineSpaceDataAddress (&_NV_UndefineSpaceData)
3533 #else
3534 #define _NV_UndefineSpaceDataAddress 0
3535 #endif // CC_NV_UndefineSpace
3536 #if CC_NV_UndefineSpaceSpecial
3537 #include "NV_UndefineSpaceSpecial_fp.h"
3538 typedef TPM_RC  (NV_UndefineSpaceSpecial_Entry)(
3539     NV_UndefineSpaceSpecial_In          *in
3540 );
3541 typedef const struct {
3542     NV_UndefineSpaceSpecial_Entry   *entry;
3543     UINT16                          inSize;
3544     UINT16                          outSize;
3545     UINT16                          offsetOfTypes;
3546     UINT16                          paramOffsets[1];
3547     BYTE                            types[4];
3548 } NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t;
3549 NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t _NV_UndefineSpaceSpecialData = {
3550     /* entry         */            &TPM2_NV_UndefineSpaceSpecial,
3551     /* inSize        */            (UINT16)(sizeof(NV_UndefineSpaceSpecial_In)),
3552     /* outSize       */            0,
3553     /* offsetOfTypes */
     offsetof(NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t, types),
```

```
3554     /* offsets        */              {(UINT16)(offsetof(NV_UndefineSpaceSpecial_In,
platform))},
3555     /* types          */              {TPMI_RH_NV_INDEX_H_UNMARSHAL,
3556                                        TPMI_RH_PLATFORM_H_UNMARSHAL,
3557                                        END_OF_LIST,
3558                                        END_OF_LIST}
3559 };
3560 #define _NV_UndefineSpaceSpecialDataAddress (&_NV_UndefineSpaceSpecialData)
3561 #else
3562 #define _NV_UndefineSpaceSpecialDataAddress 0
3563 #endif // CC_NV_UndefineSpaceSpecial
3564 #if CC_NV_ReadPublic
3565 #include "NV_ReadPublic_fp.h"
3566 typedef TPM_RC  (NV_ReadPublic_Entry)(
3567     NV_ReadPublic_In             *in,
3568     NV_ReadPublic_Out            *out
3569 );
3570 typedef const struct {
3571     NV_ReadPublic_Entry     *entry;
3572     UINT16                  inSize;
3573     UINT16                  outSize;
3574     UINT16                  offsetOfTypes;
3575     UINT16                  paramOffsets[1];
3576     BYTE                    types[5];
3577 } NV_ReadPublic_COMMAND_DESCRIPTOR_t;
3578 NV_ReadPublic_COMMAND_DESCRIPTOR_t _NV_ReadPublicData = {
3579     /* entry          */    &TPM2_NV_ReadPublic,
3580     /* inSize         */    (UINT16)(sizeof(NV_ReadPublic_In)),
3581     /* outSize        */    (UINT16)(sizeof(NV_ReadPublic_Out)),
3582     /* offsetOfTypes */     offsetof(NV_ReadPublic_COMMAND_DESCRIPTOR_t, types),
3583     /* offsets        */    {(UINT16)(offsetof(NV_ReadPublic_Out, nvName))},
3584     /* types          */    {TPMI_RH_NV_INDEX_H_UNMARSHAL,
3585                              END_OF_LIST,
3586                              TPM2B_NV_PUBLIC_P_MARSHAL,
3587                              TPM2B_NAME_P_MARSHAL,
3588                              END_OF_LIST}
3589 };
3590 #define _NV_ReadPublicDataAddress (&_NV_ReadPublicData)
3591 #else
3592 #define _NV_ReadPublicDataAddress 0
3593 #endif // CC_NV_ReadPublic
3594 #if CC_NV_Write
3595 #include "NV_Write_fp.h"
3596 typedef TPM_RC  (NV_Write_Entry)(
3597     NV_Write_In                  *in
3598 );
3599 typedef const struct {
3600     NV_Write_Entry          *entry;
3601     UINT16                  inSize;
3602     UINT16                  outSize;
3603     UINT16                  offsetOfTypes;
3604     UINT16                  paramOffsets[3];
3605     BYTE                    types[6];
3606 } NV_Write_COMMAND_DESCRIPTOR_t;
3607 NV_Write_COMMAND_DESCRIPTOR_t _NV_WriteData = {
3608     /* entry          */    &TPM2_NV_Write,
3609     /* inSize         */    (UINT16)(sizeof(NV_Write_In)),
3610     /* outSize        */    0,
3611     /* offsetOfTypes */     offsetof(NV_Write_COMMAND_DESCRIPTOR_t, types),
3612     /* offsets        */    {(UINT16)(offsetof(NV_Write_In, nvIndex)),
3613                              (UINT16)(offsetof(NV_Write_In, data)),
3614                              (UINT16)(offsetof(NV_Write_In, offset))},
3615     /* types          */    {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3616                              TPMI_RH_NV_INDEX_H_UNMARSHAL,
3617                              TPM2B_MAX_NV_BUFFER_P_UNMARSHAL,
3618                              UINT16_P_UNMARSHAL,
```

 Copyright © TCG 2006-2020

```
3619                                     END_OF_LIST,
3620                                     END_OF_LIST}
3621    };
3622    #define _NV_WriteDataAddress (&_NV_WriteData)
3623    #else
3624    #define _NV_WriteDataAddress 0
3625    #endif // CC_NV_Write
3626    #if CC_NV_Increment
3627    #include "NV_Increment_fp.h"
3628    typedef TPM_RC  (NV_Increment_Entry)(
3629        NV_Increment_In              *in
3630    );
3631    typedef const struct {
3632        NV_Increment_Entry      *entry;
3633        UINT16                  inSize;
3634        UINT16                  outSize;
3635        UINT16                  offsetOfTypes;
3636        UINT16                  paramOffsets[1];
3637        BYTE                    types[4];
3638    } NV_Increment_COMMAND_DESCRIPTOR_t;
3639    NV_Increment_COMMAND_DESCRIPTOR_t _NV_IncrementData = {
3640        /* entry         */     &TPM2_NV_Increment,
3641        /* inSize        */     (UINT16)(sizeof(NV_Increment_In)),
3642        /* outSize       */     0,
3643        /* offsetOfTypes */     offsetof(NV_Increment_COMMAND_DESCRIPTOR_t, types),
3644        /* offsets       */     {(UINT16)(offsetof(NV_Increment_In, nvIndex))},
3645        /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3646                                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3647                                 END_OF_LIST,
3648                                 END_OF_LIST}
3649    };
3650    #define _NV_IncrementDataAddress (&_NV_IncrementData)
3651    #else
3652    #define _NV_IncrementDataAddress 0
3653    #endif // CC_NV_Increment
3654    #if CC_NV_Extend
3655    #include "NV_Extend_fp.h"
3656    typedef TPM_RC  (NV_Extend_Entry)(
3657        NV_Extend_In                 *in
3658    );
3659    typedef const struct {
3660        NV_Extend_Entry         *entry;
3661        UINT16                  inSize;
3662        UINT16                  outSize;
3663        UINT16                  offsetOfTypes;
3664        UINT16                  paramOffsets[2];
3665        BYTE                    types[5];
3666    } NV_Extend_COMMAND_DESCRIPTOR_t;
3667    NV_Extend_COMMAND_DESCRIPTOR_t _NV_ExtendData = {
3668        /* entry         */     &TPM2_NV_Extend,
3669        /* inSize        */     (UINT16)(sizeof(NV_Extend_In)),
3670        /* outSize       */     0,
3671        /* offsetOfTypes */     offsetof(NV_Extend_COMMAND_DESCRIPTOR_t, types),
3672        /* offsets       */     {(UINT16)(offsetof(NV_Extend_In, nvIndex)),
3673                                 (UINT16)(offsetof(NV_Extend_In, data))},
3674        /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3675                                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3676                                 TPM2B_MAX_NV_BUFFER_P_UNMARSHAL,
3677                                 END_OF_LIST,
3678                                 END_OF_LIST}
3679    };
3680    #define _NV_ExtendDataAddress (&_NV_ExtendData)
3681    #else
3682    #define _NV_ExtendDataAddress 0
3683    #endif // CC_NV_Extend
3684    #if CC_NV_SetBits
```

```
3685    #include "NV_SetBits_fp.h"
3686    typedef TPM_RC  (NV_SetBits_Entry)(
3687        NV_SetBits_In               *in
3688    );
3689    typedef const struct {
3690        NV_SetBits_Entry        *entry;
3691        UINT16                  inSize;
3692        UINT16                  outSize;
3693        UINT16                  offsetOfTypes;
3694        UINT16                  paramOffsets[2];
3695        BYTE                    types[5];
3696    } NV_SetBits_COMMAND_DESCRIPTOR_t;
3697    NV_SetBits_COMMAND_DESCRIPTOR_t _NV_SetBitsData = {
3698        /* entry         */     &TPM2_NV_SetBits,
3699        /* inSize        */     (UINT16)(sizeof(NV_SetBits_In)),
3700        /* outSize       */     0,
3701        /* offsetOfTypes */     offsetof(NV_SetBits_COMMAND_DESCRIPTOR_t, types),
3702        /* offsets       */     {(UINT16)(offsetof(NV_SetBits_In, nvIndex)),
3703                                 (UINT16)(offsetof(NV_SetBits_In, bits))},
3704        /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3705                                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3706                                 UINT64_P_UNMARSHAL,
3707                                 END_OF_LIST,
3708                                 END_OF_LIST}
3709    };
3710    #define _NV_SetBitsDataAddress (&_NV_SetBitsData)
3711    #else
3712    #define _NV_SetBitsDataAddress 0
3713    #endif // CC_NV_SetBits
3714    #if CC_NV_WriteLock
3715    #include "NV_WriteLock_fp.h"
3716    typedef TPM_RC  (NV_WriteLock_Entry)(
3717        NV_WriteLock_In             *in
3718    );
3719    typedef const struct {
3720        NV_WriteLock_Entry      *entry;
3721        UINT16                  inSize;
3722        UINT16                  outSize;
3723        UINT16                  offsetOfTypes;
3724        UINT16                  paramOffsets[1];
3725        BYTE                    types[4];
3726    } NV_WriteLock_COMMAND_DESCRIPTOR_t;
3727    NV_WriteLock_COMMAND_DESCRIPTOR_t _NV_WriteLockData = {
3728        /* entry         */     &TPM2_NV_WriteLock,
3729        /* inSize        */     (UINT16)(sizeof(NV_WriteLock_In)),
3730        /* outSize       */     0,
3731        /* offsetOfTypes */     offsetof(NV_WriteLock_COMMAND_DESCRIPTOR_t, types),
3732        /* offsets       */     {(UINT16)(offsetof(NV_WriteLock_In, nvIndex))},
3733        /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3734                                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3735                                 END_OF_LIST,
3736                                 END_OF_LIST}
3737    };
3738    #define _NV_WriteLockDataAddress (&_NV_WriteLockData)
3739    #else
3740    #define _NV_WriteLockDataAddress 0
3741    #endif // CC_NV_WriteLock
3742    #if CC_NV_GlobalWriteLock
3743    #include "NV_GlobalWriteLock_fp.h"
3744    typedef TPM_RC  (NV_GlobalWriteLock_Entry)(
3745        NV_GlobalWriteLock_In           *in
3746    );
3747    typedef const struct {
3748        NV_GlobalWriteLock_Entry    *entry;
3749        UINT16                      inSize;
3750        UINT16                      outSize;
```

```
3751     UINT16                    offsetOfTypes;
3752     BYTE                      types[3];
3753 } NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t;
3754 NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t _NV_GlobalWriteLockData = {
3755     /* entry         */        &TPM2_NV_GlobalWriteLock,
3756     /* inSize        */        (UINT16)(sizeof(NV_GlobalWriteLock_In)),
3757     /* outSize       */        0,
3758     /* offsetOfTypes */        offsetof(NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t,
types),
3759     /* offsets       */        // No parameter offsets;
3760     /* types         */        {TPMI_RH_PROVISION_H_UNMARSHAL,
3761                                 END_OF_LIST,
3762                                 END_OF_LIST}
3763 };
3764 #define _NV_GlobalWriteLockDataAddress (&_NV_GlobalWriteLockData)
3765 #else
3766 #define _NV_GlobalWriteLockDataAddress 0
3767 #endif // CC_NV_GlobalWriteLock
3768 #if CC_NV_Read
3769 #include "NV_Read_fp.h"
3770 typedef TPM_RC  (NV_Read_Entry)(
3771     NV_Read_In                *in,
3772     NV_Read_Out               *out
3773 );
3774 typedef const struct {
3775     NV_Read_Entry           *entry;
3776     UINT16                  inSize;
3777     UINT16                  outSize;
3778     UINT16                  offsetOfTypes;
3779     UINT16                  paramOffsets[3];
3780     BYTE                    types[7];
3781 } NV_Read_COMMAND_DESCRIPTOR_t;
3782 NV_Read_COMMAND_DESCRIPTOR_t _NV_ReadData = {
3783     /* entry         */     &TPM2_NV_Read,
3784     /* inSize        */     (UINT16)(sizeof(NV_Read_In)),
3785     /* outSize       */     (UINT16)(sizeof(NV_Read_Out)),
3786     /* offsetOfTypes */     offsetof(NV_Read_COMMAND_DESCRIPTOR_t, types),
3787     /* offsets       */     {(UINT16)(offsetof(NV_Read_In, nvIndex)),
3788                              (UINT16)(offsetof(NV_Read_In, size)),
3789                              (UINT16)(offsetof(NV_Read_In, offset))},
3790     /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3791                              TPMI_RH_NV_INDEX_H_UNMARSHAL,
3792                              UINT16_P_UNMARSHAL,
3793                              UINT16_P_UNMARSHAL,
3794                              END_OF_LIST,
3795                              TPM2B_MAX_NV_BUFFER_P_MARSHAL,
3796                              END_OF_LIST}
3797 };
3798 #define _NV_ReadDataAddress (&_NV_ReadData)
3799 #else
3800 #define _NV_ReadDataAddress 0
3801 #endif // CC_NV_Read
3802 #if CC_NV_ReadLock
3803 #include "NV_ReadLock_fp.h"
3804 typedef TPM_RC  (NV_ReadLock_Entry)(
3805     NV_ReadLock_In            *in
3806 );
3807 typedef const struct {
3808     NV_ReadLock_Entry       *entry;
3809     UINT16                  inSize;
3810     UINT16                  outSize;
3811     UINT16                  offsetOfTypes;
3812     UINT16                  paramOffsets[1];
3813     BYTE                    types[4];
3814 } NV_ReadLock_COMMAND_DESCRIPTOR_t;
3815 NV_ReadLock_COMMAND_DESCRIPTOR_t _NV_ReadLockData = {
```

 Copyright © TCG 2006-2020

```
3816      /* entry          */      &TPM2_NV_ReadLock,
3817      /* inSize         */      (UINT16)(sizeof(NV_ReadLock_In)),
3818      /* outSize        */      0,
3819      /* offsetOfTypes */       offsetof(NV_ReadLock_COMMAND_DESCRIPTOR_t, types),
3820      /* offsets        */      {(UINT16)(offsetof(NV_ReadLock_In, nvIndex))},
3821      /* types          */      {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3822                               TPMI_RH_NV_INDEX_H_UNMARSHAL,
3823                               END_OF_LIST,
3824                               END_OF_LIST}
3825  };
3826  #define _NV_ReadLockDataAddress (&_NV_ReadLockData)
3827  #else
3828  #define _NV_ReadLockDataAddress 0
3829  #endif // CC_NV_ReadLock
3830  #if CC_NV_ChangeAuth
3831  #include "NV_ChangeAuth_fp.h"
3832  typedef TPM_RC  (NV_ChangeAuth_Entry)(
3833      NV_ChangeAuth_In            *in
3834  );
3835  typedef const struct {
3836      NV_ChangeAuth_Entry     *entry;
3837      UINT16                  inSize;
3838      UINT16                  outSize;
3839      UINT16                  offsetOfTypes;
3840      UINT16                  paramOffsets[1];
3841      BYTE                    types[4];
3842  } NV_ChangeAuth_COMMAND_DESCRIPTOR_t;
3843  NV_ChangeAuth_COMMAND_DESCRIPTOR_t _NV_ChangeAuthData = {
3844      /* entry          */      &TPM2_NV_ChangeAuth,
3845      /* inSize         */      (UINT16)(sizeof(NV_ChangeAuth_In)),
3846      /* outSize        */      0,
3847      /* offsetOfTypes */       offsetof(NV_ChangeAuth_COMMAND_DESCRIPTOR_t, types),
3848      /* offsets        */      {(UINT16)(offsetof(NV_ChangeAuth_In, newAuth))},
3849      /* types          */      {TPMI_RH_NV_INDEX_H_UNMARSHAL,
3850                               TPM2B_AUTH_P_UNMARSHAL,
3851                               END_OF_LIST,
3852                               END_OF_LIST}
3853  };
3854  #define _NV_ChangeAuthDataAddress (&_NV_ChangeAuthData)
3855  #else
3856  #define _NV_ChangeAuthDataAddress 0
3857  #endif // CC_NV_ChangeAuth
3858  #if CC_NV_Certify
3859  #include "NV_Certify_fp.h"
3860  typedef TPM_RC  (NV_Certify_Entry)(
3861      NV_Certify_In               *in,
3862      NV_Certify_Out              *out
3863  );
3864  typedef const struct {
3865      NV_Certify_Entry        *entry;
3866      UINT16                  inSize;
3867      UINT16                  outSize;
3868      UINT16                  offsetOfTypes;
3869      UINT16                  paramOffsets[7];
3870      BYTE                    types[11];
3871  } NV_Certify_COMMAND_DESCRIPTOR_t;
3872  NV_Certify_COMMAND_DESCRIPTOR_t _NV_CertifyData = {
3873      /* entry          */      &TPM2_NV_Certify,
3874      /* inSize         */      (UINT16)(sizeof(NV_Certify_In)),
3875      /* outSize        */      (UINT16)(sizeof(NV_Certify_Out)),
3876      /* offsetOfTypes */       offsetof(NV_Certify_COMMAND_DESCRIPTOR_t, types),
3877      /* offsets        */      {(UINT16)(offsetof(NV_Certify_In, authHandle)),
3878                                (UINT16)(offsetof(NV_Certify_In, nvIndex)),
3879                                (UINT16)(offsetof(NV_Certify_In, qualifyingData)),
3880                                (UINT16)(offsetof(NV_Certify_In, inScheme)),
3881                                (UINT16)(offsetof(NV_Certify_In, size)),
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
3882                                 (UINT16)(offsetof(NV_Certify_In, offset)),
3883                                 (UINT16)(offsetof(NV_Certify_Out, signature))},
3884        /* types           */   {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
3885                                 TPMI_RH_NV_AUTH_H_UNMARSHAL,
3886                                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3887                                 TPM2B_DATA_P_UNMARSHAL,
3888                                 TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
3889                                 UINT16_P_UNMARSHAL,
3890                                 UINT16_P_UNMARSHAL,
3891                                 END_OF_LIST,
3892                                 TPM2B_ATTEST_P_MARSHAL,
3893                                 TPMT_SIGNATURE_P_MARSHAL,
3894                                 END_OF_LIST}
3895    };
3896    #define _NV_CertifyDataAddress (&_NV_CertifyData)
3897    #else
3898    #define _NV_CertifyDataAddress 0
3899    #endif // CC_NV_Certify
3900    #if CC_AC_GetCapability
3901    #include "AC_GetCapability_fp.h"
3902    typedef TPM_RC  (AC_GetCapability_Entry)(
3903        AC_GetCapability_In          *in,
3904        AC_GetCapability_Out         *out
3905    );
3906    typedef const struct {
3907        AC_GetCapability_Entry   *entry;
3908        UINT16                   inSize;
3909        UINT16                   outSize;
3910        UINT16                   offsetOfTypes;
3911        UINT16                   paramOffsets[3];
3912        BYTE                     types[7];
3913    } AC_GetCapability_COMMAND_DESCRIPTOR_t;
3914    AC_GetCapability_COMMAND_DESCRIPTOR_t _AC_GetCapabilityData = {
3915        /* entry           */   &TPM2_AC_GetCapability,
3916        /* inSize          */   (UINT16)(sizeof(AC_GetCapability_In)),
3917        /* outSize         */   (UINT16)(sizeof(AC_GetCapability_Out)),
3918        /* offsetOfTypes */     offsetof(AC_GetCapability_COMMAND_DESCRIPTOR_t, types),
3919        /* offsets         */   {(UINT16)(offsetof(AC_GetCapability_In, capability)),
3920                                 (UINT16)(offsetof(AC_GetCapability_In, count)),
3921                                 (UINT16)(offsetof(AC_GetCapability_Out,
        capabilitiesData))},
3922        /* types           */   {TPMI_RH_AC_H_UNMARSHAL,
3923                                 TPM_AT_P_UNMARSHAL,
3924                                 UINT32_P_UNMARSHAL,
3925                                 END_OF_LIST,
3926                                 TPMI_YES_NO_P_MARSHAL,
3927                                 TPML_AC_CAPABILITIES_P_MARSHAL,
3928                                 END_OF_LIST}
3929    };
3930    #define _AC_GetCapabilityDataAddress (&_AC_GetCapabilityData)
3931    #else
3932    #define _AC_GetCapabilityDataAddress 0
3933    #endif // CC_AC_GetCapability
3934    #if CC_AC_Send
3935    #include "AC_Send_fp.h"
3936    typedef TPM_RC  (AC_Send_Entry)(
3937        AC_Send_In                   *in,
3938        AC_Send_Out                  *out
3939    );
3940    typedef const struct {
3941        AC_Send_Entry            *entry;
3942        UINT16                   inSize;
3943        UINT16                   outSize;
3944        UINT16                   offsetOfTypes;
3945        UINT16                   paramOffsets[3];
3946        BYTE                     types[7];
```

```
3947 } AC_Send_COMMAND_DESCRIPTOR_t;
3948 AC_Send_COMMAND_DESCRIPTOR_t _AC_SendData = {
3949     /* entry          */      &TPM2_AC_Send,
3950     /* inSize         */      (UINT16)(sizeof(AC_Send_In)),
3951     /* outSize        */      (UINT16)(sizeof(AC_Send_Out)),
3952     /* offsetOfTypes */      offsetof(AC_Send_COMMAND_DESCRIPTOR_t, types),
3953     /* offsets        */      {(UINT16)(offsetof(AC_Send_In, authHandle)),
3954                               (UINT16)(offsetof(AC_Send_In, ac)),
3955                               (UINT16)(offsetof(AC_Send_In, acDataIn))},
3956     /* types          */      {TPMI_DH_OBJECT_H_UNMARSHAL,
3957                               TPMI_RH_NV_AUTH_H_UNMARSHAL,
3958                               TPMI_RH_AC_H_UNMARSHAL,
3959                               TPM2B_MAX_BUFFER_P_UNMARSHAL,
3960                               END_OF_LIST,
3961                               TPMS_AC_OUTPUT_P_MARSHAL,
3962                               END_OF_LIST}
3963 };
3964 #define _AC_SendDataAddress (&_AC_SendData)
3965 #else
3966 #define _AC_SendDataAddress 0
3967 #endif // CC_AC_Send
3968 #if CC_Policy_AC_SendSelect
3969 #include "Policy_AC_SendSelect_fp.h"
3970 typedef TPM_RC  (Policy_AC_SendSelect_Entry)(
3971     Policy_AC_SendSelect_In         *in
3972 );
3973 typedef const struct {
3974     Policy_AC_SendSelect_Entry  *entry;
3975     UINT16                      inSize;
3976     UINT16                      outSize;
3977     UINT16                      offsetOfTypes;
3978     UINT16                      paramOffsets[4];
3979     BYTE                        types[7];
3980 } Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t;
3981 Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t _Policy_AC_SendSelectData = {
3982     /* entry          */         &TPM2_Policy_AC_SendSelect,
3983     /* inSize         */         (UINT16)(sizeof(Policy_AC_SendSelect_In)),
3984     /* outSize        */         0,
3985     /* offsetOfTypes */         offsetof(Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t,
     types),
3986     /* offsets        */         {(UINT16)(offsetof(Policy_AC_SendSelect_In,
     objectName)),
3987                                  (UINT16)(offsetof(Policy_AC_SendSelect_In,
     authHandleName)),
3988                                  (UINT16)(offsetof(Policy_AC_SendSelect_In, acName)),
3989                                  (UINT16)(offsetof(Policy_AC_SendSelect_In,
     includeObject))},
3990     /* types          */         {TPMI_SH_POLICY_H_UNMARSHAL,
3991                                  TPM2B_NAME_P_UNMARSHAL,
3992                                  TPM2B_NAME_P_UNMARSHAL,
3993                                  TPM2B_NAME_P_UNMARSHAL,
3994                                  TPMI_YES_NO_P_UNMARSHAL,
3995                                  END_OF_LIST,
3996                                  END_OF_LIST}
3997 };
3998 #define _Policy_AC_SendSelectDataAddress (&_Policy_AC_SendSelectData)
3999 #else
4000 #define _Policy_AC_SendSelectDataAddress 0
4001 #endif // CC_Policy_AC_SendSelect
4002 #if CC_ACT_SetTimeout
4003 #include "ACT_SetTimeout_fp.h"
4004 typedef TPM_RC  (ACT_SetTimeout_Entry)(
4005     ACT_SetTimeout_In           *in
4006 );
4007 typedef const struct {
4008     ACT_SetTimeout_Entry    *entry;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
4009        UINT16                      inSize;
4010        UINT16                      outSize;
4011        UINT16                      offsetOfTypes;
4012        UINT16                      paramOffsets[1];
4013        BYTE                        types[4];
4014    } ACT_SetTimeout_COMMAND_DESCRIPTOR_t;
4015    ACT_SetTimeout_COMMAND_DESCRIPTOR_t _ACT_SetTimeoutData = {
4016        /* entry         */     &TPM2_ACT_SetTimeout,
4017        /* inSize        */     (UINT16)(sizeof(ACT_SetTimeout_In)),
4018        /* outSize       */     0,
4019        /* offsetOfTypes */     offsetof(ACT_SetTimeout_COMMAND_DESCRIPTOR_t, types),
4020        /* offsets       */     {(UINT16)(offsetof(ACT_SetTimeout_In, startTimeout))},
4021        /* types         */     {TPMI_RH_ACT_H_UNMARSHAL,
4022                                 UINT32_P_UNMARSHAL,
4023                                 END_OF_LIST,
4024                                 END_OF_LIST}
4025    };
4026    #define _ACT_SetTimeoutDataAddress (&_ACT_SetTimeoutData)
4027    #else
4028    #define _ACT_SetTimeoutDataAddress 0
4029    #endif // CC_ACT_SetTimeout
4030    #if CC_Vendor_TCG_Test
4031    #include "Vendor_TCG_Test_fp.h"
4032    typedef TPM_RC  (Vendor_TCG_Test_Entry)(
4033        Vendor_TCG_Test_In          *in,
4034        Vendor_TCG_Test_Out         *out
4035    );
4036    typedef const struct {
4037        Vendor_TCG_Test_Entry   *entry;
4038        UINT16                  inSize;
4039        UINT16                  outSize;
4040        UINT16                  offsetOfTypes;
4041        BYTE                    types[4];
4042    } Vendor_TCG_Test_COMMAND_DESCRIPTOR_t;
4043    Vendor_TCG_Test_COMMAND_DESCRIPTOR_t _Vendor_TCG_TestData = {
4044        /* entry         */     &TPM2_Vendor_TCG_Test,
4045        /* inSize        */     (UINT16)(sizeof(Vendor_TCG_Test_In)),
4046        /* outSize       */     (UINT16)(sizeof(Vendor_TCG_Test_Out)),
4047        /* offsetOfTypes */     offsetof(Vendor_TCG_Test_COMMAND_DESCRIPTOR_t, types),
4048        /* offsets       */     // No parameter offsets;
4049        /* types         */     {TPM2B_DATA_P_UNMARSHAL,
4050                                 END_OF_LIST,
4051                                 TPM2B_DATA_P_MARSHAL,
4052                                 END_OF_LIST}
4053    };
4054    #define _Vendor_TCG_TestDataAddress (&_Vendor_TCG_TestData)
4055    #else
4056    #define _Vendor_TCG_TestDataAddress 0
4057    #endif // CC_Vendor_TCG_Test
4058    COMMAND_DESCRIPTOR_t *s_CommandDataArray[] = {
4059    #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
4060            (COMMAND_DESCRIPTOR_t *)_NV_UndefineSpaceSpecialDataAddress,
4061    #endif // CC_NV_UndefineSpaceSpecial
4062    #if (PAD_LIST || CC_EvictControl)
4063            (COMMAND_DESCRIPTOR_t *)_EvictControlDataAddress,
4064    #endif // CC_EvictControl
4065    #if (PAD_LIST || CC_HierarchyControl)
4066            (COMMAND_DESCRIPTOR_t *)_HierarchyControlDataAddress,
4067    #endif // CC_HierarchyControl
4068    #if (PAD_LIST || CC_NV_UndefineSpace)
4069            (COMMAND_DESCRIPTOR_t *)_NV_UndefineSpaceDataAddress,
4070    #endif // CC_NV_UndefineSpace
4071    #if (PAD_LIST)
4072            (COMMAND_DESCRIPTOR_t *)0,
4073    #endif //
4074    #if (PAD_LIST || CC_ChangeEPS)
```

```
4075            (COMMAND_DESCRIPTOR_t *)_ChangeEPSDataAddress,
4076    #endif // CC_ChangeEPS
4077    #if (PAD_LIST || CC_ChangePPS)
4078            (COMMAND_DESCRIPTOR_t *)_ChangePPSDataAddress,
4079    #endif // CC_ChangePPS
4080    #if (PAD_LIST || CC_Clear)
4081            (COMMAND_DESCRIPTOR_t *)_ClearDataAddress,
4082    #endif // CC_Clear
4083    #if (PAD_LIST || CC_ClearControl)
4084            (COMMAND_DESCRIPTOR_t *)_ClearControlDataAddress,
4085    #endif // CC_ClearControl
4086    #if (PAD_LIST || CC_ClockSet)
4087            (COMMAND_DESCRIPTOR_t *)_ClockSetDataAddress,
4088    #endif // CC_ClockSet
4089    #if (PAD_LIST || CC_HierarchyChangeAuth)
4090            (COMMAND_DESCRIPTOR_t *)_HierarchyChangeAuthDataAddress,
4091    #endif // CC_HierarchyChangeAuth
4092    #if (PAD_LIST || CC_NV_DefineSpace)
4093            (COMMAND_DESCRIPTOR_t *)_NV_DefineSpaceDataAddress,
4094    #endif // CC_NV_DefineSpace
4095    #if (PAD_LIST || CC_PCR_Allocate)
4096            (COMMAND_DESCRIPTOR_t *)_PCR_AllocateDataAddress,
4097    #endif // CC_PCR_Allocate
4098    #if (PAD_LIST || CC_PCR_SetAuthPolicy)
4099            (COMMAND_DESCRIPTOR_t *)_PCR_SetAuthPolicyDataAddress,
4100    #endif // CC_PCR_SetAuthPolicy
4101    #if (PAD_LIST || CC_PP_Commands)
4102            (COMMAND_DESCRIPTOR_t *)_PP_CommandsDataAddress,
4103    #endif // CC_PP_Commands
4104    #if (PAD_LIST || CC_SetPrimaryPolicy)
4105            (COMMAND_DESCRIPTOR_t *)_SetPrimaryPolicyDataAddress,
4106    #endif // CC_SetPrimaryPolicy
4107    #if (PAD_LIST || CC_FieldUpgradeStart)
4108            (COMMAND_DESCRIPTOR_t *)_FieldUpgradeStartDataAddress,
4109    #endif // CC_FieldUpgradeStart
4110    #if (PAD_LIST || CC_ClockRateAdjust)
4111            (COMMAND_DESCRIPTOR_t *)_ClockRateAdjustDataAddress,
4112    #endif // CC_ClockRateAdjust
4113    #if (PAD_LIST || CC_CreatePrimary)
4114            (COMMAND_DESCRIPTOR_t *)_CreatePrimaryDataAddress,
4115    #endif // CC_CreatePrimary
4116    #if (PAD_LIST || CC_NV_GlobalWriteLock)
4117            (COMMAND_DESCRIPTOR_t *)_NV_GlobalWriteLockDataAddress,
4118    #endif // CC_NV_GlobalWriteLock
4119    #if (PAD_LIST || CC_GetCommandAuditDigest)
4120            (COMMAND_DESCRIPTOR_t *)_GetCommandAuditDigestDataAddress,
4121    #endif // CC_GetCommandAuditDigest
4122    #if (PAD_LIST || CC_NV_Increment)
4123            (COMMAND_DESCRIPTOR_t *)_NV_IncrementDataAddress,
4124    #endif // CC_NV_Increment
4125    #if (PAD_LIST || CC_NV_SetBits)
4126            (COMMAND_DESCRIPTOR_t *)_NV_SetBitsDataAddress,
4127    #endif // CC_NV_SetBits
4128    #if (PAD_LIST || CC_NV_Extend)
4129            (COMMAND_DESCRIPTOR_t *)_NV_ExtendDataAddress,
4130    #endif // CC_NV_Extend
4131    #if (PAD_LIST || CC_NV_Write)
4132            (COMMAND_DESCRIPTOR_t *)_NV_WriteDataAddress,
4133    #endif // CC_NV_Write
4134    #if (PAD_LIST || CC_NV_WriteLock)
4135            (COMMAND_DESCRIPTOR_t *)_NV_WriteLockDataAddress,
4136    #endif // CC_NV_WriteLock
4137    #if (PAD_LIST || CC_DictionaryAttackLockReset)
4138            (COMMAND_DESCRIPTOR_t *)_DictionaryAttackLockResetDataAddress,
4139    #endif // CC_DictionaryAttackLockReset
4140    #if (PAD_LIST || CC_DictionaryAttackParameters)
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
4141            (COMMAND_DESCRIPTOR_t *)_DictionaryAttackParametersDataAddress,
4142    #endif // CC_DictionaryAttackParameters
4143    #if (PAD_LIST || CC_NV_ChangeAuth)
4144            (COMMAND_DESCRIPTOR_t *)_NV_ChangeAuthDataAddress,
4145    #endif // CC_NV_ChangeAuth
4146    #if (PAD_LIST || CC_PCR_Event)
4147            (COMMAND_DESCRIPTOR_t *)_PCR_EventDataAddress,
4148    #endif // CC_PCR_Event
4149    #if (PAD_LIST || CC_PCR_Reset)
4150            (COMMAND_DESCRIPTOR_t *)_PCR_ResetDataAddress,
4151    #endif // CC_PCR_Reset
4152    #if (PAD_LIST || CC_SequenceComplete)
4153            (COMMAND_DESCRIPTOR_t *)_SequenceCompleteDataAddress,
4154    #endif // CC_SequenceComplete
4155    #if (PAD_LIST || CC_SetAlgorithmSet)
4156            (COMMAND_DESCRIPTOR_t *)_SetAlgorithmSetDataAddress,
4157    #endif // CC_SetAlgorithmSet
4158    #if (PAD_LIST || CC_SetCommandCodeAuditStatus)
4159            (COMMAND_DESCRIPTOR_t *)_SetCommandCodeAuditStatusDataAddress,
4160    #endif // CC_SetCommandCodeAuditStatus
4161    #if (PAD_LIST || CC_FieldUpgradeData)
4162            (COMMAND_DESCRIPTOR_t *)_FieldUpgradeDataDataAddress,
4163    #endif // CC_FieldUpgradeData
4164    #if (PAD_LIST || CC_IncrementalSelfTest)
4165            (COMMAND_DESCRIPTOR_t *)_IncrementalSelfTestDataAddress,
4166    #endif // CC_IncrementalSelfTest
4167    #if (PAD_LIST || CC_SelfTest)
4168            (COMMAND_DESCRIPTOR_t *)_SelfTestDataAddress,
4169    #endif // CC_SelfTest
4170    #if (PAD_LIST || CC_Startup)
4171            (COMMAND_DESCRIPTOR_t *)_StartupDataAddress,
4172    #endif // CC_Startup
4173    #if (PAD_LIST || CC_Shutdown)
4174            (COMMAND_DESCRIPTOR_t *)_ShutdownDataAddress,
4175    #endif // CC_Shutdown
4176    #if (PAD_LIST || CC_StirRandom)
4177            (COMMAND_DESCRIPTOR_t *)_StirRandomDataAddress,
4178    #endif // CC_StirRandom
4179    #if (PAD_LIST || CC_ActivateCredential)
4180            (COMMAND_DESCRIPTOR_t *)_ActivateCredentialDataAddress,
4181    #endif // CC_ActivateCredential
4182    #if (PAD_LIST || CC_Certify)
4183            (COMMAND_DESCRIPTOR_t *)_CertifyDataAddress,
4184    #endif // CC_Certify
4185    #if (PAD_LIST || CC_PolicyNV)
4186            (COMMAND_DESCRIPTOR_t *)_PolicyNVDataAddress,
4187    #endif // CC_PolicyNV
4188    #if (PAD_LIST || CC_CertifyCreation)
4189            (COMMAND_DESCRIPTOR_t *)_CertifyCreationDataAddress,
4190    #endif // CC_CertifyCreation
4191    #if (PAD_LIST || CC_Duplicate)
4192            (COMMAND_DESCRIPTOR_t *)_DuplicateDataAddress,
4193    #endif // CC_Duplicate
4194    #if (PAD_LIST || CC_GetTime)
4195            (COMMAND_DESCRIPTOR_t *)_GetTimeDataAddress,
4196    #endif // CC_GetTime
4197    #if (PAD_LIST || CC_GetSessionAuditDigest)
4198            (COMMAND_DESCRIPTOR_t *)_GetSessionAuditDigestDataAddress,
4199    #endif // CC_GetSessionAuditDigest
4200    #if (PAD_LIST || CC_NV_Read)
4201            (COMMAND_DESCRIPTOR_t *)_NV_ReadDataAddress,
4202    #endif // CC_NV_Read
4203    #if (PAD_LIST || CC_NV_ReadLock)
4204            (COMMAND_DESCRIPTOR_t *)_NV_ReadLockDataAddress,
4205    #endif // CC_NV_ReadLock
4206    #if (PAD_LIST || CC_ObjectChangeAuth)
```

 Copyright © TCG 2006-2020

```
4207            (COMMAND_DESCRIPTOR_t *)_ObjectChangeAuthDataAddress,
4208    #endif // CC_ObjectChangeAuth
4209    #if (PAD_LIST || CC_PolicySecret)
4210            (COMMAND_DESCRIPTOR_t *)_PolicySecretDataAddress,
4211    #endif // CC_PolicySecret
4212    #if (PAD_LIST || CC_Rewrap)
4213            (COMMAND_DESCRIPTOR_t *)_RewrapDataAddress,
4214    #endif // CC_Rewrap
4215    #if (PAD_LIST || CC_Create)
4216            (COMMAND_DESCRIPTOR_t *)_CreateDataAddress,
4217    #endif // CC_Create
4218    #if (PAD_LIST || CC_ECDH_ZGen)
4219            (COMMAND_DESCRIPTOR_t *)_ECDH_ZGenDataAddress,
4220    #endif // CC_ECDH_ZGen
4221    #if (PAD_LIST || (CC_HMAC || CC_MAC))
4222    #    if CC_HMAC
4223            (COMMAND_DESCRIPTOR_t *)_HMACDataAddress,
4224    #    endif
4225    #    if CC_MAC
4226            (COMMAND_DESCRIPTOR_t *)_MACDataAddress,
4227    #    endif
4228    #    if (CC_HMAC || CC_MAC) > 1
4229    #        error "More than one aliased command defined"
4230    #    endif
4231    #endif // CC_HMAC CC_MAC
4232    #if (PAD_LIST || CC_Import)
4233            (COMMAND_DESCRIPTOR_t *)_ImportDataAddress,
4234    #endif // CC_Import
4235    #if (PAD_LIST || CC_Load)
4236            (COMMAND_DESCRIPTOR_t *)_LoadDataAddress,
4237    #endif // CC_Load
4238    #if (PAD_LIST || CC_Quote)
4239            (COMMAND_DESCRIPTOR_t *)_QuoteDataAddress,
4240    #endif // CC_Quote
4241    #if (PAD_LIST || CC_RSA_Decrypt)
4242            (COMMAND_DESCRIPTOR_t *)_RSA_DecryptDataAddress,
4243    #endif // CC_RSA_Decrypt
4244    #if (PAD_LIST)
4245            (COMMAND_DESCRIPTOR_t *)0,
4246    #endif //
4247    #if (PAD_LIST || (CC_HMAC_Start || CC_MAC_Start))
4248    #    if CC_HMAC_Start
4249            (COMMAND_DESCRIPTOR_t *)_HMAC_StartDataAddress,
4250    #    endif
4251    #    if CC_MAC_Start
4252            (COMMAND_DESCRIPTOR_t *)_MAC_StartDataAddress,
4253    #    endif
4254    #    if (CC_HMAC_Start || CC_MAC_Start) > 1
4255    #        error "More than one aliased command defined"
4256    #    endif
4257    #endif // CC_HMAC_Start CC_MAC_Start
4258    #if (PAD_LIST || CC_SequenceUpdate)
4259            (COMMAND_DESCRIPTOR_t *)_SequenceUpdateDataAddress,
4260    #endif // CC_SequenceUpdate
4261    #if (PAD_LIST || CC_Sign)
4262            (COMMAND_DESCRIPTOR_t *)_SignDataAddress,
4263    #endif // CC_Sign
4264    #if (PAD_LIST || CC_Unseal)
4265            (COMMAND_DESCRIPTOR_t *)_UnsealDataAddress,
4266    #endif // CC_Unseal
4267    #if (PAD_LIST)
4268            (COMMAND_DESCRIPTOR_t *)0,
4269    #endif //
4270    #if (PAD_LIST || CC_PolicySigned)
4271            (COMMAND_DESCRIPTOR_t *)_PolicySignedDataAddress,
4272    #endif // CC_PolicySigned
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
4273 #if (PAD_LIST || CC_ContextLoad)
4274         (COMMAND_DESCRIPTOR_t *)_ContextLoadDataAddress,
4275 #endif // CC_ContextLoad
4276 #if (PAD_LIST || CC_ContextSave)
4277         (COMMAND_DESCRIPTOR_t *)_ContextSaveDataAddress,
4278 #endif // CC_ContextSave
4279 #if (PAD_LIST || CC_ECDH_KeyGen)
4280         (COMMAND_DESCRIPTOR_t *)_ECDH_KeyGenDataAddress,
4281 #endif // CC_ECDH_KeyGen
4282 #if (PAD_LIST || CC_EncryptDecrypt)
4283         (COMMAND_DESCRIPTOR_t *)_EncryptDecryptDataAddress,
4284 #endif // CC_EncryptDecrypt
4285 #if (PAD_LIST || CC_FlushContext)
4286         (COMMAND_DESCRIPTOR_t *)_FlushContextDataAddress,
4287 #endif // CC_FlushContext
4288 #if (PAD_LIST)
4289         (COMMAND_DESCRIPTOR_t *)0,
4290 #endif //
4291 #if (PAD_LIST || CC_LoadExternal)
4292         (COMMAND_DESCRIPTOR_t *)_LoadExternalDataAddress,
4293 #endif // CC_LoadExternal
4294 #if (PAD_LIST || CC_MakeCredential)
4295         (COMMAND_DESCRIPTOR_t *)_MakeCredentialDataAddress,
4296 #endif // CC_MakeCredential
4297 #if (PAD_LIST || CC_NV_ReadPublic)
4298         (COMMAND_DESCRIPTOR_t *)_NV_ReadPublicDataAddress,
4299 #endif // CC_NV_ReadPublic
4300 #if (PAD_LIST || CC_PolicyAuthorize)
4301         (COMMAND_DESCRIPTOR_t *)_PolicyAuthorizeDataAddress,
4302 #endif // CC_PolicyAuthorize
4303 #if (PAD_LIST || CC_PolicyAuthValue)
4304         (COMMAND_DESCRIPTOR_t *)_PolicyAuthValueDataAddress,
4305 #endif // CC_PolicyAuthValue
4306 #if (PAD_LIST || CC_PolicyCommandCode)
4307         (COMMAND_DESCRIPTOR_t *)_PolicyCommandCodeDataAddress,
4308 #endif // CC_PolicyCommandCode
4309 #if (PAD_LIST || CC_PolicyCounterTimer)
4310         (COMMAND_DESCRIPTOR_t *)_PolicyCounterTimerDataAddress,
4311 #endif // CC_PolicyCounterTimer
4312 #if (PAD_LIST || CC_PolicyCpHash)
4313         (COMMAND_DESCRIPTOR_t *)_PolicyCpHashDataAddress,
4314 #endif // CC_PolicyCpHash
4315 #if (PAD_LIST || CC_PolicyLocality)
4316         (COMMAND_DESCRIPTOR_t *)_PolicyLocalityDataAddress,
4317 #endif // CC_PolicyLocality
4318 #if (PAD_LIST || CC_PolicyNameHash)
4319         (COMMAND_DESCRIPTOR_t *)_PolicyNameHashDataAddress,
4320 #endif // CC_PolicyNameHash
4321 #if (PAD_LIST || CC_PolicyOR)
4322         (COMMAND_DESCRIPTOR_t *)_PolicyORDataAddress,
4323 #endif // CC_PolicyOR
4324 #if (PAD_LIST || CC_PolicyTicket)
4325         (COMMAND_DESCRIPTOR_t *)_PolicyTicketDataAddress,
4326 #endif // CC_PolicyTicket
4327 #if (PAD_LIST || CC_ReadPublic)
4328         (COMMAND_DESCRIPTOR_t *)_ReadPublicDataAddress,
4329 #endif // CC_ReadPublic
4330 #if (PAD_LIST || CC_RSA_Encrypt)
4331         (COMMAND_DESCRIPTOR_t *)_RSA_EncryptDataAddress,
4332 #endif // CC_RSA_Encrypt
4333 #if (PAD_LIST)
4334         (COMMAND_DESCRIPTOR_t *)0,
4335 #endif //
4336 #if (PAD_LIST || CC_StartAuthSession)
4337         (COMMAND_DESCRIPTOR_t *)_StartAuthSessionDataAddress,
4338 #endif // CC_StartAuthSession
```

 Copyright © TCG 2006-2020

```
4339 #if (PAD_LIST || CC_VerifySignature)
4340         (COMMAND_DESCRIPTOR_t *)_VerifySignatureDataAddress,
4341 #endif // CC_VerifySignature
4342 #if (PAD_LIST || CC_ECC_Parameters)
4343         (COMMAND_DESCRIPTOR_t *)_ECC_ParametersDataAddress,
4344 #endif // CC_ECC_Parameters
4345 #if (PAD_LIST || CC_FirmwareRead)
4346         (COMMAND_DESCRIPTOR_t *)_FirmwareReadDataAddress,
4347 #endif // CC_FirmwareRead
4348 #if (PAD_LIST || CC_GetCapability)
4349         (COMMAND_DESCRIPTOR_t *)_GetCapabilityDataAddress,
4350 #endif // CC_GetCapability
4351 #if (PAD_LIST || CC_GetRandom)
4352         (COMMAND_DESCRIPTOR_t *)_GetRandomDataAddress,
4353 #endif // CC_GetRandom
4354 #if (PAD_LIST || CC_GetTestResult)
4355         (COMMAND_DESCRIPTOR_t *)_GetTestResultDataAddress,
4356 #endif // CC_GetTestResult
4357 #if (PAD_LIST || CC_Hash)
4358         (COMMAND_DESCRIPTOR_t *)_HashDataAddress,
4359 #endif // CC_Hash
4360 #if (PAD_LIST || CC_PCR_Read)
4361         (COMMAND_DESCRIPTOR_t *)_PCR_ReadDataAddress,
4362 #endif // CC_PCR_Read
4363 #if (PAD_LIST || CC_PolicyPCR)
4364         (COMMAND_DESCRIPTOR_t *)_PolicyPCRDataAddress,
4365 #endif // CC_PolicyPCR
4366 #if (PAD_LIST || CC_PolicyRestart)
4367         (COMMAND_DESCRIPTOR_t *)_PolicyRestartDataAddress,
4368 #endif // CC_PolicyRestart
4369 #if (PAD_LIST || CC_ReadClock)
4370         (COMMAND_DESCRIPTOR_t *)_ReadClockDataAddress,
4371 #endif // CC_ReadClock
4372 #if (PAD_LIST || CC_PCR_Extend)
4373         (COMMAND_DESCRIPTOR_t *)_PCR_ExtendDataAddress,
4374 #endif // CC_PCR_Extend
4375 #if (PAD_LIST || CC_PCR_SetAuthValue)
4376         (COMMAND_DESCRIPTOR_t *)_PCR_SetAuthValueDataAddress,
4377 #endif // CC_PCR_SetAuthValue
4378 #if (PAD_LIST || CC_NV_Certify)
4379         (COMMAND_DESCRIPTOR_t *)_NV_CertifyDataAddress,
4380 #endif // CC_NV_Certify
4381 #if (PAD_LIST || CC_EventSequenceComplete)
4382         (COMMAND_DESCRIPTOR_t *)_EventSequenceCompleteDataAddress,
4383 #endif // CC_EventSequenceComplete
4384 #if (PAD_LIST || CC_HashSequenceStart)
4385         (COMMAND_DESCRIPTOR_t *)_HashSequenceStartDataAddress,
4386 #endif // CC_HashSequenceStart
4387 #if (PAD_LIST || CC_PolicyPhysicalPresence)
4388         (COMMAND_DESCRIPTOR_t *)_PolicyPhysicalPresenceDataAddress,
4389 #endif // CC_PolicyPhysicalPresence
4390 #if (PAD_LIST || CC_PolicyDuplicationSelect)
4391         (COMMAND_DESCRIPTOR_t *)_PolicyDuplicationSelectDataAddress,
4392 #endif // CC_PolicyDuplicationSelect
4393 #if (PAD_LIST || CC_PolicyGetDigest)
4394         (COMMAND_DESCRIPTOR_t *)_PolicyGetDigestDataAddress,
4395 #endif // CC_PolicyGetDigest
4396 #if (PAD_LIST || CC_TestParms)
4397         (COMMAND_DESCRIPTOR_t *)_TestParmsDataAddress,
4398 #endif // CC_TestParms
4399 #if (PAD_LIST || CC_Commit)
4400         (COMMAND_DESCRIPTOR_t *)_CommitDataAddress,
4401 #endif // CC_Commit
4402 #if (PAD_LIST || CC_PolicyPassword)
4403         (COMMAND_DESCRIPTOR_t *)_PolicyPasswordDataAddress,
4404 #endif // CC_PolicyPassword
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
4405 #if (PAD_LIST || CC_ZGen_2Phase)
4406         (COMMAND_DESCRIPTOR_t *)_ZGen_2PhaseDataAddress,
4407 #endif // CC_ZGen_2Phase
4408 #if (PAD_LIST || CC_EC_Ephemeral)
4409         (COMMAND_DESCRIPTOR_t *)_EC_EphemeralDataAddress,
4410 #endif // CC_EC_Ephemeral
4411 #if (PAD_LIST || CC_PolicyNvWritten)
4412         (COMMAND_DESCRIPTOR_t *)_PolicyNvWrittenDataAddress,
4413 #endif // CC_PolicyNvWritten
4414 #if (PAD_LIST || CC_PolicyTemplate)
4415         (COMMAND_DESCRIPTOR_t *)_PolicyTemplateDataAddress,
4416 #endif // CC_PolicyTemplate
4417 #if (PAD_LIST || CC_CreateLoaded)
4418         (COMMAND_DESCRIPTOR_t *)_CreateLoadedDataAddress,
4419 #endif // CC_CreateLoaded
4420 #if (PAD_LIST || CC_PolicyAuthorizeNV)
4421         (COMMAND_DESCRIPTOR_t *)_PolicyAuthorizeNVDataAddress,
4422 #endif // CC_PolicyAuthorizeNV
4423 #if (PAD_LIST || CC_EncryptDecrypt2)
4424         (COMMAND_DESCRIPTOR_t *)_EncryptDecrypt2DataAddress,
4425 #endif // CC_EncryptDecrypt2
4426 #if (PAD_LIST || CC_AC_GetCapability)
4427         (COMMAND_DESCRIPTOR_t *)_AC_GetCapabilityDataAddress,
4428 #endif // CC_AC_GetCapability
4429 #if (PAD_LIST || CC_AC_Send)
4430         (COMMAND_DESCRIPTOR_t *)_AC_SendDataAddress,
4431 #endif // CC_AC_Send
4432 #if (PAD_LIST || CC_Policy_AC_SendSelect)
4433         (COMMAND_DESCRIPTOR_t *)_Policy_AC_SendSelectDataAddress,
4434 #endif // CC_Policy_AC_SendSelect
4435 #if (PAD_LIST || CC_CertifyX509)
4436         (COMMAND_DESCRIPTOR_t *)_CertifyX509DataAddress,
4437 #endif // CC_CertifyX509
4438 #if (PAD_LIST || CC_ACT_SetTimeout)
4439         (COMMAND_DESCRIPTOR_t *)_ACT_SetTimeoutDataAddress,
4440 #endif // CC_ACT_SetTimeout
4441 #if (PAD_LIST || CC_Vendor_TCG_Test)
4442         (COMMAND_DESCRIPTOR_t *)_Vendor_TCG_TestDataAddress,
4443 #endif // CC_Vendor_TCG_Test
4444         0
4445 };
4446 #endif  // _COMMAND_TABLE_DISPATCH_
```

 Copyright © TCG 2006-2020

5.7 Commands.h

```
1   #ifndef _COMMANDS_H_
2   #define _COMMANDS_H_
```

Start-up

```
3   #ifdef TPM_CC_Startup
4   #include "Startup_fp.h"
5   #endif
6   #ifdef TPM_CC_Shutdown
7   #include "Shutdown_fp.h"
8   #endif
```

Testing

```
9   #ifdef TPM_CC_SelfTest
10  #include "SelfTest_fp.h"
11  #endif
12  #ifdef TPM_CC_IncrementalSelfTest
13  #include "IncrementalSelfTest_fp.h"
14  #endif
15  #ifdef TPM_CC_GetTestResult
16  #include "GetTestResult_fp.h"
17  #endif
```

Session Commands

```
18  #ifdef TPM_CC_StartAuthSession
19  #include "StartAuthSession_fp.h"
20  #endif
21  #ifdef TPM_CC_PolicyRestart
22  #include "PolicyRestart_fp.h"
23  #endif
```

Object Commands

```
24  #ifdef TPM_CC_Create
25  #include "Create_fp.h"
26  #endif
27  #ifdef TPM_CC_Load
28  #include "Load_fp.h"
29  #endif
30  #ifdef TPM_CC_LoadExternal
31  #include "LoadExternal_fp.h"
32  #endif
33  #ifdef TPM_CC_ReadPublic
34  #include "ReadPublic_fp.h"
35  #endif
36  #ifdef TPM_CC_ActivateCredential
37  #include "ActivateCredential_fp.h"
38  #endif
39  #ifdef TPM_CC_MakeCredential
40  #include "MakeCredential_fp.h"
41  #endif
42  #ifdef TPM_CC_Unseal
43  #include "Unseal_fp.h"
44  #endif
45  #ifdef TPM_CC_ObjectChangeAuth
46  #include "ObjectChangeAuth_fp.h"
47  #endif
48  #ifdef TPM_CC_CreateLoaded
49  #include "CreateLoaded_fp.h"
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
50 #endif
```

Duplication Commands

```
51 #ifdef TPM_CC_Duplicate
52 #include "Duplicate_fp.h"
53 #endif
54 #ifdef TPM_CC_Rewrap
55 #include "Rewrap_fp.h"
56 #endif
57 #ifdef TPM_CC_Import
58 #include "Import_fp.h"
59 #endif
```

Asymmetric Primitives

```
60 #ifdef TPM_CC_RSA_Encrypt
61 #include "RSA_Encrypt_fp.h"
62 #endif
63 #ifdef TPM_CC_RSA_Decrypt
64 #include "RSA_Decrypt_fp.h"
65 #endif
66 #ifdef TPM_CC_ECDH_KeyGen
67 #include "ECDH_KeyGen_fp.h"
68 #endif
69 #ifdef TPM_CC_ECDH_ZGen
70 #include "ECDH_ZGen_fp.h"
71 #endif
72 #ifdef TPM_CC_ECC_Parameters
73 #include "ECC_Parameters_fp.h"
74 #endif
75 #ifdef TPM_CC_ZGen_2Phase
76 #include "ZGen_2Phase_fp.h"
77 #endif
```

Symmetric Primitives

```
78 #ifdef TPM_CC_EncryptDecrypt
79 #include "EncryptDecrypt_fp.h"
80 #endif
81 #ifdef TPM_CC_EncryptDecrypt2
82 #include "EncryptDecrypt2_fp.h"
83 #endif
84 #ifdef TPM_CC_Hash
85 #include "Hash_fp.h"
86 #endif
87 #ifdef TPM_CC_HMAC
88 #include "HMAC_fp.h"
89 #endif
90 #ifdef TPM_CC_MAC
91 #include "MAC_fp.h"
92 #endif
```

Random Number Generator

```
93 #ifdef TPM_CC_GetRandom
94 #include "GetRandom_fp.h"
95 #endif
96 #ifdef TPM_CC_StirRandom
97 #include "StirRandom_fp.h"
98 #endif
```

Hash/HMAC/Event Sequences

 Copyright © TCG 2006-2020

```
 99 #ifdef TPM_CC_HMAC_Start
100 #include "HMAC_Start_fp.h"
101 #endif
102 #ifdef TPM_CC_MAC_Start
103 #include "MAC_Start_fp.h"
104 #endif
105 #ifdef TPM_CC_HashSequenceStart
106 #include "HashSequenceStart_fp.h"
107 #endif
108 #ifdef TPM_CC_SequenceUpdate
109 #include "SequenceUpdate_fp.h"
110 #endif
111 #ifdef TPM_CC_SequenceComplete
112 #include "SequenceComplete_fp.h"
113 #endif
114 #ifdef TPM_CC_EventSequenceComplete
115 #include "EventSequenceComplete_fp.h"
116 #endif
```

Attestation Commands

```
117 #ifdef TPM_CC_Certify
118 #include "Certify_fp.h"
119 #endif
120 #ifdef TPM_CC_CertifyCreation
121 #include "CertifyCreation_fp.h"
122 #endif
123 #ifdef TPM_CC_Quote
124 #include "Quote_fp.h"
125 #endif
126 #ifdef TPM_CC_GetSessionAuditDigest
127 #include "GetSessionAuditDigest_fp.h"
128 #endif
129 #ifdef TPM_CC_GetCommandAuditDigest
130 #include "GetCommandAuditDigest_fp.h"
131 #endif
132 #ifdef TPM_CC_GetTime
133 #include "GetTime_fp.h"
134 #endif
135 #ifdef TPM_CC_CertifyX509
136 #include "CertifyX509_fp.h"
137 #endif
```

Ephemeral EC Keys

```
138 #ifdef TPM_CC_Commit
139 #include "Commit_fp.h"
140 #endif
141 #ifdef TPM_CC_EC_Ephemeral
142 #include "EC_Ephemeral_fp.h"
143 #endif
```

Signing and Signature Verification

```
144 #ifdef TPM_CC_VerifySignature
145 #include "VerifySignature_fp.h"
146 #endif
147 #ifdef TPM_CC_Sign
148 #include "Sign_fp.h"
149 #endif
```

Command Audit

```
150 #ifdef TPM_CC_SetCommandCodeAuditStatus
```

```
151 #include "SetCommandCodeAuditStatus_fp.h"
152 #endif
```

Integrity Collection (PCR)

```
153 #ifdef TPM_CC_PCR_Extend
154 #include "PCR_Extend_fp.h"
155 #endif
156 #ifdef TPM_CC_PCR_Event
157 #include "PCR_Event_fp.h"
158 #endif
159 #ifdef TPM_CC_PCR_Read
160 #include "PCR_Read_fp.h"
161 #endif
162 #ifdef TPM_CC_PCR_Allocate
163 #include "PCR_Allocate_fp.h"
164 #endif
165 #ifdef TPM_CC_PCR_SetAuthPolicy
166 #include "PCR_SetAuthPolicy_fp.h"
167 #endif
168 #ifdef TPM_CC_PCR_SetAuthValue
169 #include "PCR_SetAuthValue_fp.h"
170 #endif
171 #ifdef TPM_CC_PCR_Reset
172 #include "PCR_Reset_fp.h"
173 #endif
```

Enhanced Authorization (EA) Commands

```
174 #ifdef TPM_CC_PolicySigned
175 #include "PolicySigned_fp.h"
176 #endif
177 #ifdef TPM_CC_PolicySecret
178 #include "PolicySecret_fp.h"
179 #endif
180 #ifdef TPM_CC_PolicyTicket
181 #include "PolicyTicket_fp.h"
182 #endif
183 #ifdef TPM_CC_PolicyOR
184 #include "PolicyOR_fp.h"
185 #endif
186 #ifdef TPM_CC_PolicyPCR
187 #include "PolicyPCR_fp.h"
188 #endif
189 #ifdef TPM_CC_PolicyLocality
190 #include "PolicyLocality_fp.h"
191 #endif
192 #ifdef TPM_CC_PolicyNV
193 #include "PolicyNV_fp.h"
194 #endif
195 #ifdef TPM_CC_PolicyCounterTimer
196 #include "PolicyCounterTimer_fp.h"
197 #endif
198 #ifdef TPM_CC_PolicyCommandCode
199 #include "PolicyCommandCode_fp.h"
200 #endif
201 #ifdef TPM_CC_PolicyPhysicalPresence
202 #include "PolicyPhysicalPresence_fp.h"
203 #endif
204 #ifdef TPM_CC_PolicyCpHash
205 #include "PolicyCpHash_fp.h"
206 #endif
207 #ifdef TPM_CC_PolicyNameHash
208 #include "PolicyNameHash_fp.h"
209 #endif
```

```
210 #ifdef TPM_CC_PolicyDuplicationSelect
211 #include "PolicyDuplicationSelect_fp.h"
212 #endif
213 #ifdef TPM_CC_PolicyAuthorize
214 #include "PolicyAuthorize_fp.h"
215 #endif
216 #ifdef TPM_CC_PolicyAuthValue
217 #include "PolicyAuthValue_fp.h"
218 #endif
219 #ifdef TPM_CC_PolicyPassword
220 #include "PolicyPassword_fp.h"
221 #endif
222 #ifdef TPM_CC_PolicyGetDigest
223 #include "PolicyGetDigest_fp.h"
224 #endif
225 #ifdef TPM_CC_PolicyNvWritten
226 #include "PolicyNvWritten_fp.h"
227 #endif
228 #ifdef TPM_CC_PolicyTemplate
229 #include "PolicyTemplate_fp.h"
230 #endif
231 #ifdef TPM_CC_PolicyAuthorizeNV
232 #include "PolicyAuthorizeNV_fp.h"
233 #endif
```

Hierarchy Commands

```
234 #ifdef TPM_CC_CreatePrimary
235 #include "CreatePrimary_fp.h"
236 #endif
237 #ifdef TPM_CC_HierarchyControl
238 #include "HierarchyControl_fp.h"
239 #endif
240 #ifdef TPM_CC_SetPrimaryPolicy
241 #include "SetPrimaryPolicy_fp.h"
242 #endif
243 #ifdef TPM_CC_ChangePPS
244 #include "ChangePPS_fp.h"
245 #endif
246 #ifdef TPM_CC_ChangeEPS
247 #include "ChangeEPS_fp.h"
248 #endif
249 #ifdef TPM_CC_Clear
250 #include "Clear_fp.h"
251 #endif
252 #ifdef TPM_CC_ClearControl
253 #include "ClearControl_fp.h"
254 #endif
255 #ifdef TPM_CC_HierarchyChangeAuth
256 #include "HierarchyChangeAuth_fp.h"
257 #endif
```

Dictionary Attack Functions

```
258 #ifdef TPM_CC_DictionaryAttackLockReset
259 #include "DictionaryAttackLockReset_fp.h"
260 #endif
261 #ifdef TPM_CC_DictionaryAttackParameters
262 #include "DictionaryAttackParameters_fp.h"
263 #endif
```

Miscellaneous Management Functions

```
264 #ifdef TPM_CC_PP_Commands
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
265 #include "PP_Commands_fp.h"
266 #endif
267 #ifdef TPM_CC_SetAlgorithmSet
268 #include "SetAlgorithmSet_fp.h"
269 #endif
```

Field Upgrade

```
270 #ifdef TPM_CC_FieldUpgradeStart
271 #include "FieldUpgradeStart_fp.h"
272 #endif
273 #ifdef TPM_CC_FieldUpgradeData
274 #include "FieldUpgradeData_fp.h"
275 #endif
276 #ifdef TPM_CC_FirmwareRead
277 #include "FirmwareRead_fp.h"
278 #endif
```

Context Management

```
279 #ifdef TPM_CC_ContextSave
280 #include "ContextSave_fp.h"
281 #endif
282 #ifdef TPM_CC_ContextLoad
283 #include "ContextLoad_fp.h"
284 #endif
285 #ifdef TPM_CC_FlushContext
286 #include "FlushContext_fp.h"
287 #endif
288 #ifdef TPM_CC_EvictControl
289 #include "EvictControl_fp.h"
290 #endif
```

Clocks and Timers

```
291 #ifdef TPM_CC_ReadClock
292 #include "ReadClock_fp.h"
293 #endif
294 #ifdef TPM_CC_ClockSet
295 #include "ClockSet_fp.h"
296 #endif
297 #ifdef TPM_CC_ClockRateAdjust
298 #include "ClockRateAdjust_fp.h"
299 #endif
```

Capability Commands

```
300 #ifdef TPM_CC_GetCapability
301 #include "GetCapability_fp.h"
302 #endif
303 #ifdef TPM_CC_TestParms
304 #include "TestParms_fp.h"
305 #endif
```

Non-volatile Storage

```
306 #ifdef TPM_CC_NV_DefineSpace
307 #include "NV_DefineSpace_fp.h"
308 #endif
309 #ifdef TPM_CC_NV_UndefineSpace
310 #include "NV_UndefineSpace_fp.h"
311 #endif
312 #ifdef TPM_CC_NV_UndefineSpaceSpecial
```

 Copyright © TCG 2006-2020

```
313 #include "NV_UndefineSpaceSpecial_fp.h"
314 #endif
315 #ifdef TPM_CC_NV_ReadPublic
316 #include "NV_ReadPublic_fp.h"
317 #endif
318 #ifdef TPM_CC_NV_Write
319 #include "NV_Write_fp.h"
320 #endif
321 #ifdef TPM_CC_NV_Increment
322 #include "NV_Increment_fp.h"
323 #endif
324 #ifdef TPM_CC_NV_Extend
325 #include "NV_Extend_fp.h"
326 #endif
327 #ifdef TPM_CC_NV_SetBits
328 #include "NV_SetBits_fp.h"
329 #endif
330 #ifdef TPM_CC_NV_WriteLock
331 #include "NV_WriteLock_fp.h"
332 #endif
333 #ifdef TPM_CC_NV_GlobalWriteLock
334 #include "NV_GlobalWriteLock_fp.h"
335 #endif
336 #ifdef TPM_CC_NV_Read
337 #include "NV_Read_fp.h"
338 #endif
339 #ifdef TPM_CC_NV_ReadLock
340 #include "NV_ReadLock_fp.h"
341 #endif
342 #ifdef TPM_CC_NV_ChangeAuth
343 #include "NV_ChangeAuth_fp.h"
344 #endif
345 #ifdef TPM_CC_NV_Certify
346 #include "NV_Certify_fp.h"
347 #endif
```

Attached Components

```
348 #ifdef TPM_CC_AC_GetCapability
349 #include "AC_GetCapability_fp.h"
350 #endif
351 #ifdef TPM_CC_AC_Send
352 #include "AC_Send_fp.h"
353 #endif
354 #ifdef TPM_CC_Policy_AC_SendSelect
355 #include "Policy_AC_SendSelect_fp.h"
356 #endif
```

Authenticated Countdown Timer

```
357 #ifdef TPM_CC_ACT_SetTimeout
358 #include "ACT_SetTimeout_fp.h"
359 #endif
```

Vendor Specific

```
360 #ifdef TPM_CC_Vendor_TCG_Test
361 #include "Vendor_TCG_Test_fp.h"
362 #endif
363 #endif
```

5.8 CompilerDependencies.h

This file contains the build switches. This contains switches for multiple versions of the crypto-library so some may not apply to your environment.

```
1  #ifndef _COMPILER_DEPENDENCIES_H_
2  #define _COMPILER_DEPENDENCIES_H_
3  #ifdef GCC
4  #   undef _MSC_VER
5  #   undef WIN32
6  #endif
7  #ifdef _MSC_VER
```

These definitions are for the Microsoft compiler Endian conversion for aligned structures

```
8  #   define REVERSE_ENDIAN_16(_Number) _byteswap_ushort(_Number)
9  #   define REVERSE_ENDIAN_32(_Number) _byteswap_ulong(_Number)
10 #   define REVERSE_ENDIAN_64(_Number) _byteswap_uint64(_Number)
```

Avoid compiler warning for in line of stdio (or not)

```
11 //#define _NO_CRT_STDIO_INLINE
```

This macro is used to handle LIB_EXPORT of function and variable names in lieu of a .def file. Visual Studio requires that functions be explicitly exported and imported.

```
12 #   define LIB_EXPORT __declspec(dllexport) // VS compatible version
13 #   define LIB_IMPORT __declspec(dllimport)
```

This is defined to indicate a function that does not return. Microsoft compilers do not support the _Noretrun function parameter.

```
14 #   define NORETURN  __declspec(noreturn)
15 #   if _MSC_VER >= 1400     // SAL processing when needed
16 #       include <sal.h>
17 #   endif
18 #   ifdef _WIN64
19 #       define _INTPTR 2
20 #    else
21 #       define _INTPTR 1
22 #    endif
23 #define NOT_REFERENCED(x)    (x)
```

Lower the compiler error warning for system include files. They tend not to be that clean and there is no reason to sort through all the spurious errors that they generate when the normal error level is set to /Wall

```
24 #   define _REDUCE_WARNING_LEVEL_(n)                    \
25 __pragma(warning(push, n))
```

Restore the compiler warning level

```
26 #   define _NORMAL_WARNING_LEVEL_                       \
27 __pragma(warning(pop))
28 #   include <stdint.h>
29 #endif
30 #ifndef _MSC_VER
31 #ifndef WINAPI
32 #   define WINAPI
33 #endif
34 #   define __pragma(x)
35 #   define REVERSE_ENDIAN_16(_Number) __builtin_bswap16(_Number)
```

```
36  #   define REVERSE_ENDIAN_32(_Number) __builtin_bswap32(_Number)
37  #   define REVERSE_ENDIAN_64(_Number) __builtin_bswap64(_Number)
38  #endif
39  #if defined(__GNUC__)
40  #   define NORETURN                     __attribute__((noreturn))
41  #   include <stdint.h>
42  #endif
```

Things that are not defined should be defined as NULL

```
43  #ifndef NORETURN
44  #   define NORETURN
45  #endif
46  #ifndef LIB_EXPORT
47  #   define LIB_EXPORT
48  #endif
49  #ifndef LIB_IMPORT
50  #   define LIB_IMPORT
51  #endif
52  #ifndef _REDUCE_WARNING_LEVEL_
53  #   define _REDUCE_WARNING_LEVEL_(n)
54  #endif
55  #ifndef _NORMAL_WARNING_LEVEL_
56  #   define _NORMAL_WARNING_LEVEL_
57  #endif
58  #ifndef NOT_REFERENCED
59  #   define  NOT_REFERENCED(x)  (x = x)
60  #endif
61  #ifdef _POSIX_
62  typedef int SOCKET;
63  #endif
64  #endif // _COMPILER_DEPENDENCIES_H_
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

5.9 Global.h

5.9.1 Description

This file contains internal global type definitions and data declarations that are need between subsystems. The instantiation of global data is in Global.c. The initialization of global data is in the subsystem that is the primary owner of the data.

The first part of this file has the typedefs for structures and other defines used in many portions of the code. After the typedef section, is a section that defines global values that are only present in RAM. The next three sections define the structures for the NV data areas: persistent, orderly, and state save. Additional sections define the data that is used in specific modules. That data is private to the module but is collected here to simplify the management of the instance data. All the data is instanced in Global.c.

```
1    #if !defined _TPM_H_
2    #error "Should only be instanced in TPM.h"
3    #endif
```

5.9.2 Includes

```
4    #ifndef         GLOBAL_H
5    #define         GLOBAL_H
6    _REDUCE_WARNING_LEVEL_(2)
7    #include <string.h>
8    #include <stddef.h>
9    _NORMAL_WARNING_LEVEL_
10
11   #include "Capabilities.h"
12   #include "TpmTypes.h"
13   #include "CommandAttributes.h"
14   #include "CryptTest.h"
15   #include "BnValues.h"
16   #include "CryptHash.h"
17   #include "CryptSym.h"
18   #include "CryptRand.h"
19   #include "CryptEcc.h"
20   #include "CryptRsa.h"
21   #include "CryptTest.h"
22   #include "TpmError.h"
23   #include "NV.h"
24   #include "ACT.h"
25
26   //** Defines and Types
27
28   //*** Size Types
29   // These types are used to differentiate the two different size values used.
30   //
31   // NUMBYTES is used when a size is a number of bytes (usually a TPM2B)
32   typedef UINT16  NUMBYTES;
33
34   //*** Other Types
35   // An AUTH_VALUE is a BYTE array containing a digest (TPMU_HA)
36   typedef BYTE    AUTH_VALUE[sizeof(TPMU_HA)];
```

A TIME_INFO is a BYTE array that can contain a TPMS_TIME_INFO

```
37   typedef BYTE    TIME_INFO[sizeof(TPMS_TIME_INFO)];
```

A NAME is a BYTE array that can contain a TPMU_NAME

```
38   typedef BYTE    NAME[sizeof(TPMU_NAME)];
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

Definition for a PROOF value

```
39 TPM2B_TYPE(PROOF, PROOF_SIZE);
```

Definition for a Primary Seed value

```
40 TPM2B_TYPE(SEED, PRIMARY_SEED_SIZE);
```

A CLOCK_NONCE is used to tag the time value in the authorization session and in the ticket computation so that the ticket expires when there is a time discontinuity. When the clock stops during normal operation, the nonce is 64-bit value kept in RAM but it is a 32-bit counter when the clock only stops during power events.

```
41 #if CLOCK_STOPS
42 typedef UINT64          CLOCK_NONCE;
43 #else
44 typedef UINT32          CLOCK_NONCE;
45 #endif
```

5.9.3 Loaded Object Structures

5.9.3.1 Description

The structures in this section define the object layout as it exists in TPM memory.

Two types of objects are defined: an ordinary object such as a key, and a sequence object that may be a hash, HMAC, or event.

5.9.3.2 OBJECT_ATTRIBUTES

An OBJECT_ATTRIBUTES structure contains the variable attributes of an object. These properties are not part of the public properties but are used by the TPM in managing the object. An OBJECT_ATTRIBUTES is used in the definition of the OBJECT data type.

```
46 typedef struct
47 {
48     unsigned            publicOnly : 1;     //0) SET if only the public portion of
49                                             //   an object is loaded
50     unsigned            epsHierarchy : 1;   //1) SET if the object belongs to EPS
51                                             //   Hierarchy
52     unsigned            ppsHierarchy : 1;   //2) SET if the object belongs to PPS
53                                             //   Hierarchy
54     unsigned            spsHierarchy : 1;   //3) SET f the object belongs to SPS
55                                             //   Hierarchy
56     unsigned            evict : 1;          //4) SET if the object is a platform or
57                                             //   owner evict object.  Platform-
58                                             //   evict object belongs to PPS
59                                             //   hierarchy, owner-evict object
60                                             //   belongs to SPS or EPS hierarchy.
61                                             //   This bit is also used to mark a
62                                             //   completed sequence object so it
63                                             //   will be flush when the
64                                             //   SequenceComplete command succeeds.
65     unsigned            primary : 1;        //5) SET for a primary object
66     unsigned            temporary : 1;      //6) SET for a temporary object
67     unsigned            stClear : 1;        //7) SET for an stClear object
68     unsigned            hmacSeq : 1;        //8) SET for an HMAC or MAC sequence
69                                             //   object
70     unsigned            hashSeq : 1;        //9) SET for a hash sequence object
71     unsigned            eventSeq : 1;       //10) SET for an event sequence object
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
72      unsigned            ticketSafe : 1;      //11) SET if a ticket is safe to create
73                                               //    for hash sequence object
74      unsigned            firstBlock : 1;      //12) SET if the first block of hash
75                                               //    data has been received.  It
76                                               //    works with ticketSafe bit
77      unsigned            isParent : 1;        //13) SET if the key has the proper
78                                               //    attributes to be a parent key
79  //    unsigned           privateExp : 1;     //14) SET when the private exponent
80  //                                           //    of an RSA key has been validated.
81      unsigned            not_used_14 : 1;
82      unsigned            occupied : 1;        //15) SET when the slot is occupied.
83      unsigned            derivation : 1;      //16) SET when the key is a derivation
84                                               //        parent
85      unsigned            external : 1;        //17) SET when the object is loaded with
86                                               //    TPM2_LoadExternal();
87  } OBJECT_ATTRIBUTES;
88  #if ALG_RSA
```

There is an overload of the sensitive.rsa.t.size field of a TPMT_SENSITIVE when an RSA key is loaded. When the sensitive->sensitive contains an RSA key with all of the CRT values, then the MSB of the size field will be set to indicate that the buffer contains all 5 of the CRT private key values.

```
89  #define     RSA_prime_flag      0x8000
90  #endif
```

5.9.3.3 OBJECT Structure

An OBJECT structure holds the object public, sensitive, and meta-data associated. This structure is implementation dependent. For this implementation, the structure is not optimized for space but rather for clarity of the reference implementation. Other implementations may choose to overlap portions of the structure that are not used simultaneously. These changes would necessitate changes to the source code but those changes would be compatible with the reference implementation.

```
91  typedef struct OBJECT
92  {
93      // The attributes field is required to be first followed by the publicArea.
94      // This allows the overlay of the object structure and a sequence structure
95      OBJECT_ATTRIBUTES   attributes;          // object attributes
96      TPMT_PUBLIC         publicArea;          // public area of an object
97      TPMT_SENSITIVE      sensitive;           // sensitive area of an object
98      TPM2B_NAME          qualifiedName;       // object qualified name
99      TPMI_DH_OBJECT      evictHandle;         // if the object is an evict object,
100                                              // the original handle is kept here.
101                                              // The 'working' handle will be the
102                                              // handle of an object slot.
103     TPM2B_NAME          name;                // Name of the object name. Kept here
104                                              // to avoid repeatedly computing it.
105 } OBJECT;
```

5.9.3.4 HASH_OBJECT Structure

This structure holds a hash sequence object or an event sequence object.

The first four components of this structure are manually set to be the same as the first four components of the object structure. This prevents the object from being inadvertently misused as sequence objects occupy the same memory as a regular object. A debug check is present to make sure that the offsets are what they are supposed to be.

NOTE: In a future version, this will probably be renamed as SEQUENCE_OBJECT

```
106 typedef struct HASH_OBJECT
```

 Copyright © TCG 2006-2020

```
107 {
108     OBJECT_ATTRIBUTES   attributes;          // The attributes of the HASH object
109     TPMI_ALG_PUBLIC     type;                // algorithm
110     TPMI_ALG_HASH       nameAlg;             // name algorithm
111     TPMA_OBJECT         objectAttributes;    // object attributes
112
113     // The data below is unique to a sequence object
114     TPM2B_AUTH          auth;                // authorization for use of sequence
115     union
116     {
117         HASH_STATE      hashState[HASH_COUNT];
118         HMAC_STATE      hmacState;
119     }                   state;
120 } HASH_OBJECT;
121 typedef BYTE  HASH_OBJECT_BUFFER[sizeof(HASH_OBJECT)];
```

5.9.3.5 ANY_OBJECT

This is the union for holding either a sequence object or a regular object. for ContextSave() and ContextLoad()

```
122 typedef union ANY_OBJECT
123 {
124     OBJECT              entity;
125     HASH_OBJECT         hash;
126 } ANY_OBJECT;
127 typedef BYTE    ANY_OBJECT_BUFFER[sizeof(ANY_OBJECT)];
```

5.9.4 AUTH_DUP Types

These values are used in the authorization processing.

```
128 typedef UINT32          AUTH_ROLE;
129 #define AUTH_NONE       ((AUTH_ROLE)(0))
130 #define AUTH_USER       ((AUTH_ROLE)(1))
131 #define AUTH_ADMIN      ((AUTH_ROLE)(2))
132 #define AUTH_DUP        ((AUTH_ROLE)(3))
```

5.9.5 Active Session Context

5.9.5.1 Description

The structures in this section define the internal structure of a session context.

5.9.5.2 SESSION_ATTRIBUTES

The attributes in the SESSION_ATTRIBUTES structure track the various properties of the session. It maintains most of the tracking state information for the policy session. It is used within the SESSION structure.

```
133 typedef struct SESSION_ATTRIBUTES
134 {
135     unsigned    isPolicy : 1;           //1) SET if the session may only be used
136                                         //   for policy
137     unsigned    isAudit : 1;            //2) SET if the session is used for audit
138     unsigned    isBound : 1;            //3) SET if the session is bound to with an
139                                         //   entity. This attribute will be CLEAR
140                                         //   if either isPolicy or isAudit is SET.
141     unsigned    isCpHashDefined : 1;    //3) SET if the cpHash has been defined
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
142                                                   //   This attribute is not SET unless
143                                                   //   'isPolicy' is SET.
144     unsigned     isAuthValueNeeded : 1;           //5) SET if the authValue is required for
145                                                   //   computing the session HMAC. This
146                                                   //   attribute is not SET unless 'isPolicy'
147                                                   //   is SET.
148     unsigned     isPasswordNeeded : 1;            //6) SET if a password authValue is required
149                                                   //   for authorization This attribute is not
150                                                   //   SET unless 'isPolicy' is SET.
151     unsigned     isPPRequired : 1;                //7) SET if physical presence is required to
152                                                   //   be asserted when the authorization is
153                                                   //   checked. This attribute is not SET
154                                                   //   unless 'isPolicy' is SET.
155     unsigned     isTrialPolicy : 1;               //8) SET if the policy session is created
156                                                   //   for trial of the policy's policyHash
157                                                   //   generation. This attribute is not SET
158                                                   //   unless 'isPolicy' is SET.
159     unsigned     isDaBound : 1;                   //9) SET if the bind entity had noDA CLEAR.
160                                                   //   If this is SET, then an authorization
161                                                   //   failure using this session will count
162                                                   //   against lockout even if the object
163                                                   //   being authorized is exempt from DA.
164     unsigned     isLockoutBound : 1;              //10) SET if the session is bound to
165                                                   //    lockoutAuth.
166     unsigned     includeAuth : 1;                 //11) This attribute is SET when the
167                                                   //    authValue of an object is to be
168                                                   //    included in the computation of the
169                                                   //    HMAC key for the command and response
170                                                   //    computations. (was 'requestWasBound')
171     unsigned     checkNvWritten : 1;              //12) SET if the TPMA_NV_WRITTEN attribute
172                                                   //    needs to be checked when the policy is
173                                                   //    used for authorization for NV access.
174                                                   //    If this is SET for any other type, the
175                                                   //    policy will fail.
176     unsigned     nvWrittenState : 1;              //13) SET if TPMA_NV_WRITTEN is required to
177                                                   //    be SET. Used when 'checkNvWritten' is
178                                                   //    SET
179     unsigned     isTemplateSet : 1;               //14) SET if the templateHash needs to be
180                                                   //    checked for Create, CreatePrimary, or
181                                                   //    CreateLoaded.
182 } SESSION_ATTRIBUTES;
```

5.9.5.3 SESSION Structure

The SESSION structure contains all the context of a session except for the associated *contextID*.

NOTE: The *contextID* of a session is only relevant when the session context is stored off the TPM.

```
183 typedef struct SESSION
184 {
185     SESSION_ATTRIBUTES   attributes;          // session attributes
186     UINT32               pcrCounter;          // PCR counter value when PCR is
187                                               // included (policy session)
188                                               // If no PCR is included, this
189                                               // value is 0.
190     UINT64               startTime;           // The value in g_time when the session
191                                               // was started (policy session)
192     UINT64               timeout;             // The timeout relative to g_time
193                                               // There is no timeout if this value
194                                               // is 0.
195     CLOCK_NONCE          epoch;               // The g_clockEpoch value when the
196                                               // session was started. If g_clockEpoch
197                                               // does not match this value when the
198                                               // timeout is used, then
```

```
199                                                // then the command will fail.
200     TPM_CC              commandCode;           // command code (policy session)
201     TPM_ALG_ID          authHashAlg;           // session hash algorithm
202     TPMA_LOCALITY       commandLocality;       // command locality (policy session)
203     TPMT_SYM_DEF        symmetric;             // session symmetric algorithm (if any)
204     TPM2B_AUTH          sessionKey;            // session secret value used for
205                                                // this session
206     TPM2B_NONCE         nonceTPM;              // last TPM-generated nonce for
207                                                // generating HMAC and encryption keys
208    union
209     {
210         TPM2B_NAME      boundEntity;           // value used to track the entity to
211                                                // which the session is bound
212
213         TPM2B_DIGEST    cpHash;                // the required cpHash value for the
214                                                // command being authorized
215         TPM2B_DIGEST    nameHash;              // the required nameHash
216         TPM2B_DIGEST    templateHash;          // the required template for creation
217     } u1;
218
219     union
220     {
221         TPM2B_DIGEST    auditDigest;           // audit session digest
222         TPM2B_DIGEST    policyDigest;          // policyHash
223     } u2;                                      // audit log and policyHash may
224                                                // share space to save memory
225 } SESSION;
226 #define     EXPIRES_ON_RESET    INT32_MIN
227 #define     TIMEOUT_ON_RESET    UINT64_MAX
228 #define     EXPIRES_ON_RESTART  (INT32_MIN + 1)
229 #define     TIMEOUT_ON_RESTART  (UINT64_MAX - 1)
230 typedef BYTE        SESSION_BUF[sizeof(SESSION)];
```

5.9.6 PCR

5.9.6.1 PCR_SAVE Structure

The PCR_SAVE structure type contains the PCR data that are saved across power cycles. Only the static PCR are required to be saved across power cycles. The DRTM and resettable PCR are not saved. The number of static and resettable PCR is determined by the platform-specific specification to which the TPM is built.

```
231 typedef struct PCR_SAVE
232 {
233 #if     ALG_SHA1
234     BYTE                sha1[NUM_STATIC_PCR][SHA1_DIGEST_SIZE];
235 #endif
236 #if     ALG_SHA256
237     BYTE                sha256[NUM_STATIC_PCR][SHA256_DIGEST_SIZE];
238 #endif
239 #if     ALG_SHA384
240     BYTE                sha384[NUM_STATIC_PCR][SHA384_DIGEST_SIZE];
241 #endif
242 #if     ALG_SHA512
243     BYTE                sha512[NUM_STATIC_PCR][SHA512_DIGEST_SIZE];
244 #endif
245 #if     ALG_SM3_256
246     BYTE                sm3_256[NUM_STATIC_PCR][SM3_256_DIGEST_SIZE];
247 #endif
248
249     // This counter increments whenever the PCR are updated.
250     // NOTE: A platform-specific specification may designate
251     //       certain PCR changes as not causing this counter
```

 Copyright © TCG 2006-2020

```
252     //        to increment.
253     UINT32                  pcrCounter;
254 } PCR_SAVE;
```

5.9.6.2 PCR_POLICY

```
255 #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
```

This structure holds the PCR policies, one for each group of PCR controlled by policy.

```
256 typedef struct PCR_POLICY
257 {
258     TPMI_ALG_HASH       hashAlg[NUM_POLICY_PCR_GROUP];
259     TPM2B_DIGEST        a;
260     TPM2B_DIGEST        policy[NUM_POLICY_PCR_GROUP];
261 } PCR_POLICY;
262 #endif
```

5.9.6.3 PCR_AUTHVALUE

This structure holds the PCR policies, one for each group of PCR controlled by policy.

```
263 typedef struct PCR_AUTH_VALUE
264 {
265     TPM2B_DIGEST        auth[NUM_AUTHVALUE_PCR_GROUP];
266 } PCR_AUTHVALUE;
```

5.9.7 STARTUP_TYPE

This enumeration is the possible startup types. The type is determined by the combination of TPM2_ShutDown() and TPM2_Startup().

```
267 typedef enum
268 {
269     SU_RESET,
270     SU_RESTART,
271     SU_RESUME
272 } STARTUP_TYPE;
```

5.9.8 NV

5.9.8.1 NV_INDEX

The NV_INDEX structure defines the internal format for an NV index. The *indexData* size varies according to the type of the index. In this implementation, all of the index is manipulated as a unit.

```
273 typedef struct NV_INDEX
274 {
275     TPMS_NV_PUBLIC      publicArea;
276     TPM2B_AUTH          authValue;
277 } NV_INDEX;
```

5.9.8.2 NV_REF

An NV_REF is an opaque value returned by the NV subsystem. It is used to reference and NV Index in a relatively efficient way. Rather than having to continually search for an Index, its reference value may be

used. In this implementation, an NV_REF is a byte pointer that points to the copy of the NV memory that is kept in RAM.

```
278 typedef UINT32           NV_REF;
279 typedef BYTE            *NV_RAM_REF;
```

5.9.8.3 NV_PIN

This structure deals with the possible endianess differences between the canonical form of the TPMS_NV_PIN_COUNTER_PARAMETERS structure and the internal value. The structures allow the data in a PIN index to be read as an 8-octet value using NvReadUINT64Data(). That function will byte swap all the values on a little endian system. This will put the bytes with the 4-octet values in the correct order but will swap the *pinLimit* and *pinCount* values. When written, the PIN index is simply handled as a normal index with the octets in canonical order.

```
280 #if BIG_ENDIAN_TPM
281 typedef struct
282 {
283     UINT32      pinCount;
284     UINT32      pinLimit;
285 } PIN_DATA;
286 #else
287 typedef struct
288 {
289     UINT32      pinLimit;
290     UINT32      pinCount;
291 } PIN_DATA;
292 #endif
293 typedef union
294 {
295     UINT64     intVal;
296     PIN_DATA   pin;
297 } NV_PIN;
```

5.9.9 COMMIT_INDEX_MASK

This is the define for the mask value that is used when manipulating the bits in the commit bit array. The commit counter is a 64-bit value and the low order bits are used to index the *commitArray*. This mask value is applied to the commit counter to extract the bit number in the array.

```
298 #if     ALG_ECC
299 #define COMMIT_INDEX_MASK ((UINT16)((sizeof(gr.commitArray)*8)-1))
300 #endif
```

5.9.10 RAM Global Values

5.9.10.1 Description

The values in this section are only extant in RAM or ROM as constant values.

5.9.10.2 Crypto Self-Test Values

```
301 EXTERN ALGORITHM_VECTOR     g_implementedAlgorithms;
302 EXTERN ALGORITHM_VECTOR     g_toTest;
303
304 //*** g_rcIndex[]
305 // This array is used to contain the array of values that are added to a return
306 // code when it is a parameter-, handle-, or session-related error.
```

 Copyright © TCG 2006-2020

```
307 // This is an implementation choice and the same result can be achieved by using
308 // a macro.
309 #define g_rcIndexInitializer {  TPM_RC_1, TPM_RC_2, TPM_RC_3, TPM_RC_4,           \
310                                 TPM_RC_5, TPM_RC_6, TPM_RC_7, TPM_RC_8,
    \
311                                 TPM_RC_9, TPM_RC_A, TPM_RC_B, TPM_RC_C,
    \
312                                 TPM_RC_D, TPM_RC_E, TPM_RC_F }
313 EXTERN const UINT16     g_rcIndex[15] INITIALIZER(g_rcIndexInitializer);
```

5.9.10.3 g_exclusiveAuditSession

This location holds the session handle for the current exclusive audit session. If there is no exclusive audit session, the location is set to TPM_RH_UNASSIGNED.

```
314 EXTERN TPM_HANDLE       g_exclusiveAuditSession;
315
316 //*** g_time
317 // This is the value in which we keep the current command time. This is initialized
318 // at the start of each command. The time is the accumulated time since the last
319 // time that the TPM's timer was last powered up. Clock is the accumulated time
320 // since the last time that the TPM was cleared. g_time is in mS.
321 EXTERN  UINT64          g_time;
322
323 //*** g_timeEpoch
324 // This value contains the current clock Epoch. It changes when there is a clock
325 // discontinuity. It may be necessary to place this in NV should the timer be able
326 // to run across a power down of the TPM but not in all cases (e.g. dead battery).
327 // If the nonce is placed in NV, it should go in gp because it should be changing
328 // slowly.
329 #if CLOCK_STOPS
330 EXTERN CLOCK_NONCE       g_timeEpoch;
331 #else
332 #define g_timeEpoch      gp.timeEpoch
333 #endif
334
335
336 //*** g_phEnable
337 // This is the platform hierarchy control and determines if the platform hierarchy
338 // is available. This value is SET on each TPM2_Startup(). The default value is
339 // SET.
340 EXTERN BOOL             g_phEnable;
341
342 //*** g_pcrReConfig
343 // This value is SET if a TPM2_PCR_Allocate command successfully executed since
344 // the last TPM2_Startup(). If so, then the next shutdown is required to be
345 // Shutdown(CLEAR).
346 EXTERN BOOL             g_pcrReConfig;
347
348 //*** g_DRTMHandle
349 // This location indicates the sequence object handle that holds the DRTM
350 // sequence data. When not used, it is set to TPM_RH_UNASSIGNED. A sequence
351 // DRTM sequence is started on either _TPM_Init or _TPM_Hash_Start.
352 EXTERN TPMI_DH_OBJECT   g_DRTMHandle;
353
354 //*** g_DrtmPreStartup
355 // This value indicates that an H-CRTM occurred after _TPM_Init but before
356 // TPM2_Startup(). The define for PRE_STARTUP_FLAG is used to add the
357 // g_DrtmPreStartup value to gp_orderlyState at shutdown. This hack is to avoid
358 // adding another NV variable.
359 EXTERN  BOOL            g_DrtmPreStartup;
360
361 //*** g_StartupLocality3
362 // This value indicates that a TPM2_Startup() occurred at locality 3. Otherwise, it
```

 Copyright © TCG 2006-2020

```
363 // at locality 0. The define for STARTUP_LOCALITY_3 is to
364 // indicate that the startup was not at locality 0. This hack is to avoid
365 // adding another NV variable.
366 EXTERN  BOOL            g_StartupLocality3;
367
368 //***TPM_SU_NONE
369 // Part 2 defines the two shutdown/startup types that may be used in
370 // TPM2_Shutdown() and TPM2_Starup(). This additional define is
371 // used by the TPM to indicate that no shutdown was received.
372 // NOTE: This is a reserved value.
373 #define SU_NONE_VALUE           (0xFFFF)
374 #define TPM_SU_NONE             (TPM_SU)(SU_NONE_VALUE)
```

5.9.10.4 TPM_SU_DA_USED

As with TPM_SU_NONE, this value is added to allow indication that the shutdown was not orderly and that a DA=protected object was reference during the previous cycle.

```
375 #define SU_DA_USED_VALUE    (SU_NONE_VALUE - 1)
376 #define TPM_SU_DA_USED      (TPM_SU)(SU_DA_USED_VALUE)
```

5.9.10.5 Startup Flags

These flags are included in *gp.orderlyState*. These are hacks and are being used to avoid having to change the layout of gp. The PRE_STARTUP_FLAG indicates that a _TPM_Hash_Start()/_Data/_End sequence was received after _TPM_Init() but before TPM2_StartUp(). STARTUP_LOCALITY_3 indicates that the last TPM2_Startup() was received at locality 3. These flags are only relevant if after a TPM2_Shutdown(STATE).

```
377 #define PRE_STARTUP_FLAG     0x8000
378 #define STARTUP_LOCALITY_3   0x4000
379 #if USE_DA_USED
```

5.9.10.6 g_daUsed

This location indicates if a DA-protected value is accessed during a boot cycle. If none has, then there is no need to increment *failedTries* on the next non-orderly startup. This bit is merged with *gp.orderlyState* when that gp.orderly is set to SU_NONE_VALUE

```
380 EXTERN  BOOL                g_daUsed;
381 #endif
382
383 //*** g_updateNV
384 // This flag indicates if NV should be updated at the end of a command.
385 // This flag is set to UT_NONE at the beginning of each command in ExecuteCommand().
386 // This flag is checked in ExecuteCommand() after the detailed actions of a command
387 // complete. If the command execution was successful and this flag is not UT_NONE,
388 // any pending NV writes will be committed to NV.
389 // UT_ORDERLY causes any RAM data to be written to the orderly space for staging
390 // the write to NV.
391 typedef BYTE        UPDATE_TYPE;
392 #define UT_NONE     (UPDATE_TYPE)0
393 #define UT_NV       (UPDATE_TYPE)1
394 #define UT_ORDERLY  (UPDATE_TYPE)(UT_NV + 2)
395 EXTERN UPDATE_TYPE          g_updateNV;
396
397 //*** g_powerWasLost
398 // This flag is used to indicate if the power was lost. It is SET in _TPM__Init.
399 // This flag is cleared by TPM2_Startup() after all power-lost activities are
400 // completed.
```

 Copyright © TCG 2006-2020

```
401 // Note: When power is applied, this value can come up as anything. However,
402 // _plat__WasPowerLost() will provide the proper indication in that case. So, when
403 // power is actually lost, we get the correct answer. When power was not lost, but
404 // the power-lost processing has not been completed before the next _TPM_Init(),
405 // then the TPM still does the correct thing.
406 EXTERN BOOL             g_powerWasLost;
407
408 //*** g_clearOrderly
409 // This flag indicates if the execution of a command should cause the orderly
410 // state to be cleared.  This flag is set to FALSE at the beginning of each
411 // command in ExecuteCommand() and is checked in ExecuteCommand() after the
412 // detailed actions of a command complete but before the check of
413 // 'g_updateNV'. If this flag is TRUE, and the orderly state is not
414 // SU_NONE_VALUE, then the orderly state in NV memory will be changed to
415 // SU_NONE_VALUE or SU_DA_USED_VALUE.
416 EXTERN BOOL             g_clearOrderly;
417
418 //*** g_prevOrderlyState
419 // This location indicates how the TPM was shut down before the most recent
420 // TPM2_Startup(). This value, along with the startup type, determines if
421 // the TPM should do a TPM Reset, TPM Restart, or TPM Resume.
422 EXTERN TPM_SU           g_prevOrderlyState;
423
424 //*** g_nvOk
425 // This value indicates if the NV integrity check was successful or not. If not and
426 // the failure was severe, then the TPM would have been put into failure mode after
427 // it had been re-manufactured. If the NV failure was in the area where the state-save
428 // data is kept, then this variable will have a value of FALSE indicating that
429 // a TPM2_Startup(CLEAR) is required.
430 EXTERN BOOL             g_nvOk;
431 // NV availability is sampled as the start of each command and stored here
432 // so that its value remains consistent during the command execution
433 EXTERN TPM_RC           g_NvStatus;
434
435 //*** g_platformUnique
436 // This location contains the unique value(s) used to identify the TPM. It is
437 // loaded on every _TPM2_Startup()
438 // The first value is used to seed the RNG. The second value is used as a vendor
439 // authValue. The value used by the RNG would be the value derived from the
440 // chip unique value (such as fused) with a dependency on the authorities of the
441 // code in the TPM boot path. The second would be derived from the chip unique value
442 // with a dependency on the details of the code in the boot path. That is, the
443 // first value depends on the various signers of the code and the second depends on
444 // what was signed. The TPM vendor should not be able to know the first value but
445 // they are expected to know the second.
446 EXTERN TPM2B_AUTH       g_platformUniqueAuthorities; // Reserved for RNG
447
448 EXTERN TPM2B_AUTH       g_platformUniqueDetails;   // referenced by
    VENDOR_PERMANENT
449
450 //*****************************************************************************
451 //*****************************************************************************
452 //** Persistent Global Values
453 //*****************************************************************************
454 //*****************************************************************************
455 //*** Description
456 // The values in this section are global values that are persistent across power
457 // events. The lifetime of the values determines the structure in which the value
458 // is placed.
459
460 //*****************************************************************************
461 //*** PERSISTENT_DATA
462 //*****************************************************************************
463 // This structure holds the persistent values that only change as a consequence
464 // of a specific Protected Capability and are not affected by TPM power events
```

```
465  // (TPM2_Startup() or TPM2_Shutdown().
466  typedef struct
467  {
468  //*****************************************************************************
469  //          Hierarchy
470  //*****************************************************************************
471  // The values in this section are related to the hierarchies.
472
473      BOOL                disableClear;       // TRUE if TPM2_Clear() using
474                                             // lockoutAuth is disabled
475
476      // Hierarchy authPolicies
477      TPMI_ALG_HASH       ownerAlg;
478      TPMI_ALG_HASH       endorsementAlg;
479      TPMI_ALG_HASH       lockoutAlg;
480      TPM2B_DIGEST        ownerPolicy;
481      TPM2B_DIGEST        endorsementPolicy;
482      TPM2B_DIGEST        lockoutPolicy;
483
484      // Hierarchy authValues
485      TPM2B_AUTH          ownerAuth;
486      TPM2B_AUTH          endorsementAuth;
487      TPM2B_AUTH          lockoutAuth;
488
489      // Primary Seeds
490      TPM2B_SEED          EPSeed;
491      TPM2B_SEED          SPSeed;
492      TPM2B_SEED          PPSeed;
493      // Note there is a nullSeed in the state_reset memory.
494
495      // Hierarchy proofs
496      TPM2B_PROOF          phProof;
497      TPM2B_PROOF          shProof;
498      TPM2B_PROOF          ehProof;
499      // Note there is a nullProof in the state_reset memory.
500
501  //*****************************************************************************
502  //          Reset Events
503  //*****************************************************************************
504  // A count that increments at each TPM reset and never get reset during the life
505  // time of TPM.  The value of this counter is initialized to 1 during TPM
506  // manufacture process. It is used to invalidate all saved contexts after a TPM
507  // Reset.
508      UINT64              totalResetCount;
509
510  // This counter increments on each TPM Reset. The counter is reset by
511  // TPM2_Clear().
512      UINT32              resetCount;
513
514  //*****************************************************************************
515  //          PCR
516  //*****************************************************************************
517  // This structure hold the policies for those PCR that have an update policy.
518  // This implementation only supports a single group of PCR controlled by
519  // policy. If more are required, then this structure would be changed to
520  // an array.
521  #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
522      PCR_POLICY          pcrPolicies;
523  #endif
524
525  // This structure indicates the allocation of PCR. The structure contains a
526  // list of PCR allocations for each implemented algorithm. If no PCR are
527  // allocated for an algorithm, a list entry still exists but the bit map
528  // will contain no SET bits.
529      TPML_PCR_SELECTION  pcrAllocated;
```

```
530
531 //*****************************************************************************
532 //          Physical Presence
533 //*****************************************************************************
534 // The PP_LIST type contains a bit map of the commands that require physical
535 // to be asserted when the authorization is evaluated. Physical presence will be
536 // checked if the corresponding bit in the array is SET and if the authorization
537 // handle is TPM_RH_PLATFORM.
538 //
539 // These bits may be changed with TPM2_PP_Commands().
540     BYTE                 ppList[(COMMAND_COUNT + 7) / 8];
541
542 //*****************************************************************************
543 //          Dictionary attack values
544 //*****************************************************************************
545 // These values are used for dictionary attack tracking and control.
546     UINT32              failedTries;        // the current count of unexpired
547                                             // authorization failures
548
549     UINT32              maxTries;           // number of unexpired authorization
550                                             // failures before the TPM is in
551                                             // lockout
552
553     UINT32              recoveryTime;       // time between authorization failures
554                                             // before failedTries is decremented
555
556     UINT32              lockoutRecovery;    // time that must expire between
557                                             // authorization failures associated
558                                             // with lockoutAuth
559
560     BOOL                lockOutAuthEnabled; // TRUE if use of lockoutAuth is
561                                             // allowed
562
563 //*************************************************************************
564 //            Orderly State
565 //*************************************************************************
566 // The orderly state for current cycle
567     TPM_SU              orderlyState;
568
569 //*************************************************************************
570 //           Command audit values.
571 //*************************************************************************
572     BYTE                auditCommands[((COMMAND_COUNT + 1) + 7) / 8];
573     TPMI_ALG_HASH       auditHashAlg;
574     UINT64              auditCounter;
575
576 //*************************************************************************
577 //           Algorithm selection
578 //*************************************************************************
579 //
580 // The 'algorithmSet' value indicates the collection of algorithms that are
581 // currently in used on the TPM.  The interpretation of value is vendor dependent.
582     UINT32              algorithmSet;
583
584 //*************************************************************************
585 //           Firmware version
586 //*************************************************************************
587 // The firmwareV1 and firmwareV2 values are instanced in TimeStamp.c. This is
588 // a scheme used in development to allow determination of the linker build time
589 // of the TPM. An actual implementation would implement these values in a way that
590 // is consistent with vendor needs. The values are maintained in RAM for simplified
591 // access with a master version in NV.  These values are modified in a
592 // vendor-specific way.
593
594 // g_firmwareV1 contains the more significant 32-bits of the vendor version number.
595 // In the reference implementation, if this value is printed as a hex
```

 Copyright © TCG 2006-2020

```
596 // value, it will have the format of YYYYMMDD
597     UINT32                  firmwareV1;
598
599 // g_firmwareV1 contains the less significant 32-bits of the vendor version number.
600 // In the reference implementation, if this value is printed as a hex
601 // value, it will have the format of 00 HH MM SS
602     UINT32                  firmwareV2;
603 //*********************************************************************************
604 //           Timer Epoch
605 //*********************************************************************************
606 // timeEpoch contains a nonce that has a vendor=specific size (should not be
607 // less than 8 bytes. This nonce changes when the clock epoch changes. The clock
608 // epoch changes when there is a discontinuity in the timing of the TPM.
609 #if !CLOCK_STOPS
610     CLOCK_NONCE             timeEpoch;
611 #endif
612
613 } PERSISTENT_DATA;
614 EXTERN PERSISTENT_DATA  gp;
615
616 //*************************************************************************************
617 //*************************************************************************************
618 //*** ORDERLY_DATA
619 //*************************************************************************************
620 //*************************************************************************************
621 // The data in this structure is saved to NV on each TPM2_Shutdown().
622 typedef struct orderly_data
623 {
624 //*********************************************************************************
625 //           TIME
626 //*********************************************************************************
627
628 // Clock has two parts. One is the state save part and one is the NV part. The
629 // state save version is updated on each command. When the clock rolls over, the
630 // NV version is updated. When the TPM starts up, if the TPM was shutdown in and
631 // orderly way, then the sClock value is used to initialize the clock. If the
632 // TPM shutdown was not orderly, then the persistent value is used and the safe
633 // attribute is clear.
634
635     UINT64                  clock;                  // The orderly version of clock
636     TPMI_YES_NO             clockSafe;              // Indicates if the clock value is
637                                                   // safe.
638
639     // In many implementations, the quality of the entropy available is not that
640     // high. To compensate, the current value of the drbgState can be saved and
641     // restored on each power cycle. This prevents the internal state from reverting
642     // to the initial state on each power cycle and starting with a limited amount
643     // of entropy. By keeping the old state and adding entropy, the entropy will
644     // accumulate.
645     DRBG_STATE              drbgState;
646
647 // These values allow the accumulation of self-healing time across orderly shutdown
648 // of the TPM.
649 #if ACCUMULATE_SELF_HEAL_TIMER
650     UINT64                  selfHealTimer;  // current value of s_selfHealTimer
651     UINT64                  lockoutTimer;   // current value of s_lockoutTimer
652     UINT64                  time;           // current value of g_time at shutdown
653 #endif // ACCUMULATE_SELF_HEAL_TIMER
654
655 // These are the ACT Timeout values. They are saved with the other timers
656 #define DefineActData(N)  ACT_STATE      ACT_##N;
657     FOR_EACH_ACT(DefineActData)
658
659  // this is the 'signaled' attribute data for all the ACT. It is done this way so
660  // that they can be manipulated by ACT number rather than having to access a
```

```
661      // structure.
662         UINT32              signaledACT;
663     } ORDERLY_DATA;
664     #if ACCUMULATE_SELF_HEAL_TIMER
665     #define       s_selfHealTimer     go.selfHealTimer
666     #define       s_lockoutTimer      go.lockoutTimer
667     #endif  // ACCUMULATE_SELF_HEAL_TIMER
668     #  define drbgDefault go.drbgState
669     EXTERN ORDERLY_DATA      go;
670
671     //*****************************************************************************
672     //*****************************************************************************
673     //*** STATE_CLEAR_DATA
674     //*****************************************************************************
675     //*****************************************************************************
676     // This structure contains the data that is saved on Shutdown(STATE)
677     // and restored on Startup(STATE).  The values are set to their default
678     // settings on any Startup(Clear). In other words, the data is only persistent
679     // across TPM Resume.
680     //
681     // If the comments associated with a parameter indicate a default reset value, the
682     // value is applied on each Startup(CLEAR).
683
684     typedef struct state_clear_data
685     {
686     //*************************************************************************
687     //          Hierarchy Control
688     //*************************************************************************
689         BOOL                shEnable;           // default reset is SET
690         BOOL                ehEnable;           // default reset is SET
691         BOOL                phEnableNV;         // default reset is SET
692         TPMI_ALG_HASH       platformAlg;        // default reset is TPM_ALG_NULL
693         TPM2B_DIGEST        platformPolicy;     // default reset is an Empty Buffer
694         TPM2B_AUTH          platformAuth;       // default reset is an Empty Buffer
695
696     //*************************************************************************
697     //          PCR
698     //*************************************************************************
699     // The set of PCR to be saved on Shutdown(STATE)
700         PCR_SAVE            pcrSave;            // default reset is 0...0
701
702     // This structure hold the authorization values for those PCR that have an
703     // update authorization.
704     // This implementation only supports a single group of PCR controlled by
705     // authorization. If more are required, then this structure would be changed to
706     // an array.
707         PCR_AUTHVALUE       pcrAuthValues;
708
709     //*************************************************************************
710     //          ACT
711     //*************************************************************************
712     #define DefineActPolicySpace(N)     TPMT_HA     act_##N;
713         FOR_EACH_ACT(DefineActPolicySpace)
714
715     } STATE_CLEAR_DATA;
716     EXTERN STATE_CLEAR_DATA gc;
717
718     //*****************************************************************************
719     //*****************************************************************************
720     //***  State Reset Data
721     //*****************************************************************************
722     //*****************************************************************************
723     // This structure contains data is that is saved on Shutdown(STATE) and restored on
724     // the subsequent Startup(ANY). That is, the data is preserved across TPM Resume
725     // and TPM Restart.
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
726  //
727  // If a default value is specified in the comments this value is applied on
728  // TPM Reset.
729
730  typedef struct state_reset_data
731  {
732  //*****************************************************************************
733  //          Hierarchy Control
734  //*****************************************************************************
735      TPM2B_PROOF          nullProof;           // The proof value associated with
736                                              // the TPM_RH_NULL hierarchy. The
737                                              // default reset value is from the RNG.
738
739      TPM2B_SEED           nullSeed;            // The seed value for the TPM_RN_NULL
740                                              // hierarchy. The default reset value
741                                              // is from the RNG.
742
743  //*****************************************************************************
744  //           Context
745  //*****************************************************************************
746  // The 'clearCount' counter is incremented each time the TPM successfully executes
747  // a TPM Resume. The counter is included in each saved context that has 'stClear'
748  // SET (including descendants of keys that have 'stClear' SET). This prevents these
749  // objects from being loaded after a TPM Resume.
750  // If 'clearCount' is at its maximum value when the TPM receives a Shutdown(STATE),
751  // the TPM will return TPM_RC_RANGE and the TPM will only accept Shutdown(CLEAR).
752      UINT32               clearCount;          // The default reset value is 0.
753
754      UINT64               objectContextID;     // This is the context ID for a saved
755                                              //  object context. The default reset
756                                              //  value is 0.
757      CONTEXT_SLOT         contextArray[MAX_ACTIVE_SESSIONS];    // This array
     contains
758                                              // contains the values used to track
759                                              // the version numbers of saved
760                                              // contexts (see
761                                              // Session.c in for details). The
762                                              // default reset value is {0}.
763
764      CONTEXT_COUNTER      contextCounter;      // This is the value from which the
765                                              // 'contextID' is derived. The
766                                              // default reset value is {0}.
767
768  //*****************************************************************************
769  //           Command Audit
770  //*****************************************************************************
771  // When an audited command completes, ExecuteCommand() checks the return
772  // value.  If it is TPM_RC_SUCCESS, and the command is an audited command, the
773  // TPM will extend the cpHash and rpHash for the command to this value. If this
774  // digest was the Zero Digest before the cpHash was extended, the audit counter
775  // is incremented.
776
777      TPM2B_DIGEST         commandAuditDigest; // This value is set to an Empty Digest
778                                              // by TPM2_GetCommandAuditDigest() or a
779                                              // TPM Reset.
780
781  //*****************************************************************************
782  //           Boot counter
783  //*****************************************************************************
784
785      UINT32               restartCount;        // This counter counts TPM Restarts.
786                                              // The default reset value is 0.
787
788  //*********************************************************************************
789  //            PCR
```


 Copyright © TCG 2006-2020

```
790 //*****************************************************************************
791 // This counter increments whenever the PCR are updated. This counter is preserved
792 // across TPM Resume even though the PCR are not preserved. This is because
793 // sessions remain active across TPM Restart and the count value in the session
794 // is compared to this counter so this counter must have values that are unique
795 // as long as the sessions are active.
796 // NOTE: A platform-specific specification may designate that certain PCR changes
797 //       do not increment this counter to increment.
798     UINT32              pcrCounter;         // The default reset value is 0.
799
800 #if     ALG_ECC
801
802 //*****************************************************************************
803 //          ECDAA
804 //*****************************************************************************
805     UINT64              commitCounter;      // This counter increments each time
806                                            // TPM2_Commit() returns
807                                            // TPM_RC_SUCCESS. The default reset
808                                            // value is 0.
809
810     TPM2B_NONCE         commitNonce;        // This random value is used to compute
811                                            // the commit values. The default reset
812                                            // value is from the RNG.
813
814 // This implementation relies on the number of bits in g_commitArray being a
815 // power of 2 (8, 16, 32, 64, etc.) and no greater than 64K.
816     BYTE                 commitArray[16];   // The default reset value is {0}.
817
818 #endif // ALG_ECC
819 } STATE_RESET_DATA;
820
821 EXTERN STATE_RESET_DATA gr;
822
823 //** NV Layout
824 // The NV data organization is
825 // 1) a PERSISTENT_DATA structure
826 // 2) a STATE_RESET_DATA structure
827 // 3) a STATE_CLEAR_DATA structure
828 // 4) an ORDERLY_DATA structure
829 // 5) the user defined NV index space
830 #define NV_PERSISTENT_DATA  (0)
831 #define NV_STATE_RESET_DATA (NV_PERSISTENT_DATA + sizeof(PERSISTENT_DATA))
832 #define NV_STATE_CLEAR_DATA (NV_STATE_RESET_DATA + sizeof(STATE_RESET_DATA))
833 #define NV_ORDERLY_DATA     (NV_STATE_CLEAR_DATA + sizeof(STATE_CLEAR_DATA))
834 #define NV_INDEX_RAM_DATA   (NV_ORDERLY_DATA + sizeof(ORDERLY_DATA))
835 #define NV_USER_DYNAMIC     (NV_INDEX_RAM_DATA + sizeof(s_indexOrderlyRam))
836 #define NV_USER_DYNAMIC_END     NV_MEMORY_SIZE
```

5.9.11 Global Macro Definitions

The NV_READ_PERSISTENT and NV_WRITE_PERSISTENT macros are used to access members of the PERSISTENT_DATA structure in NV.

```
837 #define NV_READ_PERSISTENT(to, from)                \
838             NvRead(&to, offsetof(PERSISTENT_DATA, from), sizeof(to))
839 #define NV_WRITE_PERSISTENT(to, from)               \
840             NvWrite(offsetof(PERSISTENT_DATA, to), sizeof(gp.to), &from)
841 #define CLEAR_PERSISTENT(item)                      \
842             NvClearPersistent(offsetof(PERSISTENT_DATA, item), sizeof(gp.item))
843 #define NV_SYNC_PERSISTENT(item) NV_WRITE_PERSISTENT(item, gp.item)
```

At the start of command processing, the index of the command is determined. This index value is used to access the various data tables that contain per-command information. There are multiple options for how the per-command tables can be implemented. This is resolved in GetClosestCommandIndex().

```
844  typedef UINT16      COMMAND_INDEX;
845  #define UNIMPLEMENTED_COMMAND_INDEX     ((COMMAND_INDEX)(~0))
846  typedef struct _COMMAND_FLAGS_
847  {
848      unsigned    trialPolicy : 1;    //1) If SET, one of the handles references a
849                                      //   trial policy and authorization may be
850                                      //   skipped. This is only allowed for a policy
851                                      //   command.
852  } COMMAND_FLAGS;
```

This structure is used to avoid having to manage a large number of parameters being passed through various levels of the command input processing.

```
853  typedef struct _COMMAND_
854  {
855      TPM_ST           tag;                // the parsed command tag
856      TPM_CC           code;               // the parsed command code
857      COMMAND_INDEX    index;              // the computed command index
858      UINT32           handleNum;          // the number of entity handles in the
859                                           //   handle area of the command
860      TPM_HANDLE       handles[MAX_HANDLE_NUM]; // the parsed handle values
861      UINT32           sessionNum;         // the number of sessions found
862      INT32            parameterSize;      // starts out with the parsed command size
863                                           // and is reduced and values are
864                                           // unmarshaled. Just before calling the
865                                           // command actions, this should be zero.
866                                           // After the command actions, this number
867                                           // should grow as values are marshaled
868                                           // in to the response buffer.
869      INT32            authSize;           // this is initialized with the parsed size
870                                           // of authorizationSize field and should
871                                           // be zero when the authorizations are
872                                           // parsed.
873      BYTE            *parameterBuffer;    // input to ExecuteCommand
874      BYTE            *responseBuffer;     // input to ExecuteCommand
875  #if ALG_SHA1
876      TPM2B_SHA1_DIGEST   sha1CpHash;
877      TPM2B_SHA1_DIGEST   sha1RpHash;
878  #endif
879  #if ALG_SHA256
880      TPM2B_SHA256_DIGEST sha256CpHash;
881      TPM2B_SHA256_DIGEST sha256RpHash;
882  #endif
883  #if ALG_SHA384
884      TPM2B_SHA384_DIGEST sha384CpHash;
885      TPM2B_SHA384_DIGEST sha384RpHash;
886  #endif
887  #if ALG_SHA512
888      TPM2B_SHA512_DIGEST sha512CpHash;
889      TPM2B_SHA512_DIGEST sha512RpHash;
890  #endif
891  #if ALG_SM3_256
892      TPM2B_SM3_256_DIGEST sm3_256CpHash;
893      TPM2B_SM3_256_DIGEST sm3_256RpHash;
894  #endif
895  } COMMAND;
```

Global sting constants for consistency in KDF function calls. These string constants are shared across functions to make sure that they are all using consistent sting values.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
896  #define STRING_INITIALIZER(value)    {{sizeof(value), {value}}}
897  #define TPM2B_STRING(name, value)                                              \
898  typedef union name##_ {                                                         \
899          struct  {                                                               \
900              UINT16  size;                                                       \
901              BYTE    buffer[sizeof(value)];                                      \
902          } t;                                                                    \
903          TPM2B   b;                                                              \
904      } TPM2B_##name##_;                                                          \
905  EXTERN  const TPM2B_##name##_      name##_ INITIALIZER(STRING_INITIALIZER(value));  \
906  EXTERN  const TPM2B               *name INITIALIZER(&name##_.b)
907  TPM2B_STRING(PRIMARY_OBJECT_CREATION, "Primary Object Creation");
908  TPM2B_STRING(CFB_KEY, "CFB");
909  TPM2B_STRING(CONTEXT_KEY, "CONTEXT");
910  TPM2B_STRING(INTEGRITY_KEY, "INTEGRITY");
911  TPM2B_STRING(SECRET_KEY, "SECRET");
912  TPM2B_STRING(SESSION_KEY, "ATH");
913  TPM2B_STRING(STORAGE_KEY, "STORAGE");
914  TPM2B_STRING(XOR_KEY, "XOR");
915  TPM2B_STRING(COMMIT_STRING, "ECDAA Commit");
916  TPM2B_STRING(DUPLICATE_STRING, "DUPLICATE");
917  TPM2B_STRING(IDENTITY_STRING, "IDENTITY");
918  TPM2B_STRING(OBFUSCATE_STRING, "OBFUSCATE");
919  #if SELF_TEST
920  TPM2B_STRING(OAEP_TEST_STRING, "OAEP Test Value");
921  #endif // SELF_TEST
```

5.9.12 From CryptTest.c

This structure contains the self-test state values for the cryptographic modules.

```
922  EXTERN CRYPTO_SELF_TEST_STATE    g_cryptoSelfTestState;
923
924  //*****************************************************************************
925  //** From Manufacture.c
926  //*****************************************************************************
927  EXTERN BOOL                g_manufactured INITIALIZER(FALSE);
```

This value indicates if a TPM2_Startup() commands has been receive since the power on event. This flag is maintained in power simulation module because this is the only place that may reliably set this flag to FALSE.

```
928  EXTERN BOOL                g_initialized;
929
930  //** Private data
931
932  //*****************************************************************************
933  //*** From SessionProcess.c
934  //*****************************************************************************
935  #if defined SESSION_PROCESS_C || defined GLOBAL_C || defined MANUFACTURE_C
936  // The following arrays are used to save command sessions information so that the
937  // command handle/session buffer does not have to be preserved for the duration of
938  // the command. These arrays are indexed by the session index in accordance with
939  // the order of sessions in the session area of the command.
940  //
941  // Array of the authorization session handles
942  EXTERN TPM_HANDLE       s_sessionHandles[MAX_SESSION_NUM];
943
944  // Array of authorization session attributes
945  EXTERN TPMA_SESSION     s_attributes[MAX_SESSION_NUM];
946
947  // Array of handles authorized by the corresponding authorization sessions;
948  // and if none, then TPM_RH_UNASSIGNED value is used
```

 Copyright © TCG 2006-2020

```
949 EXTERN TPM_HANDLE       s_associatedHandles[MAX_SESSION_NUM];
950
951 // Array of nonces provided by the caller for the corresponding sessions
952 EXTERN TPM2B_NONCE      s_nonceCaller[MAX_SESSION_NUM];
953
954 // Array of authorization values (HMAC's or passwords) for the corresponding
955 // sessions
956 EXTERN TPM2B_AUTH       s_inputAuthValues[MAX_SESSION_NUM];
957
958 // Array of pointers to the SESSION structures for the sessions in a command
959 EXTERN SESSION         *s_usedSessions[MAX_SESSION_NUM];
960
961 // Special value to indicate an undefined session index
962 #define             UNDEFINED_INDEX     (0xFFFF)
```

Index of the session used for encryption of a response parameter

```
963 EXTERN UINT32           s_encryptSessionIndex;
964
965 // Index of the session used for decryption of a command parameter
966 EXTERN UINT32           s_decryptSessionIndex;
967
968 // Index of a session used for audit
969 EXTERN UINT32           s_auditSessionIndex;
970
971 // The cpHash for command audit
972 #ifdef  TPM_CC_GetCommandAuditDigest
973 EXTERN TPM2B_DIGEST    s_cpHashForCommandAudit;
974 #endif
975
976 // Flag indicating if NV update is pending for the lockOutAuthEnabled or
977 // failedTries DA parameter
978 EXTERN BOOL             s_DAPendingOnNV;
979
980 #endif // SESSION_PROCESS_C
981
982 //*****************************************************************************
983 //*** From DA.c
984 //*****************************************************************************
985 #if defined DA_C || defined GLOBAL_C || defined MANUFACTURE_C
986 // This variable holds the accumulated time since the last time
987 // that 'failedTries' was decremented. This value is in millisecond.
988 #if !ACCUMULATE_SELF_HEAL_TIMER
989 EXTERN UINT64       s_selfHealTimer;
990
991 // This variable holds the accumulated time that the lockoutAuth has been
992 // blocked.
993 EXTERN UINT64       s_lockoutTimer;
994 #endif // ACCUMULATE_SELF_HEAL_TIMER
995
996 #endif // DA_C
997
998 //*****************************************************************************
999 //*** From NV.c
1000 //*****************************************************************************
1001 #if defined NV_C || defined GLOBAL_C
1002 // This marks the end of the NV area. This is a run-time variable as it might
1003 // not be compile-time constant.
1004 EXTERN NV_REF   s_evictNvEnd;
1005
1006 // This space is used to hold the index data for an orderly Index. It also contains
1007 // the attributes for the index.
1008 EXTERN BYTE      s_indexOrderlyRam[RAM_INDEX_SPACE];   // The orderly NV Index data
1009
1010 // This value contains the current max counter value. It is written to the end of
```


 Copyright © TCG 2006-2020

```
1011 // allocatable NV space each time an index is deleted or added. This value is
1012 // initialized on Startup. The indices are searched and the maximum of all the
1013 // current counter indices and this value is the initial value for this.
1014 EXTERN UINT64    s_maxCounter;
1015
1016 // This is space used for the NV Index cache. As with a persistent object, the
1017 // contents of a referenced index are copied into the cache so that the
1018 // NV Index memory scanning and data copying can be reduced.
1019 // Only code that operates on NV Index data should use this cache directly. When
1020 // that action code runs, s_lastNvIndex will contain the index header information.
1021 // It will have been loaded when the handles were verified.
1022 // NOTE: An NV index handle can appear in many commands that do not operate on the
1023 // NV data (e.g. TPM2_StartAuthSession). However, only one NV Index at a time is
1024 // ever directly referenced by any command. If that changes, then the NV Index
1025 // caching needs to be changed to accommodate that. Currently, the code will verify
1026 // that only one NV Index is referenced by the handles of the command.
1027 EXTERN      NV_INDEX         s_cachedNvIndex;
1028 EXTERN      NV_REF           s_cachedNvRef;
1029 EXTERN      BYTE            *s_cachedNvRamRef;
1030
1031 // Initial NV Index/evict object iterator value
1032 #define     NV_REF_INIT     (NV_REF)0xFFFFFFFF
1033 #endif
```

5.9.12.1 From Object.c

```
1034 #if defined OBJECT_C || defined GLOBAL_C
```

This type is the container for an object.

```
1035 EXTERN OBJECT           s_objects[MAX_LOADED_OBJECTS];
1036
1037 #endif // OBJECT_C
1038
1039 //*****************************************************************************
1040 //*** From PCR.c
1041 //*****************************************************************************
1042 #if defined PCR_C || defined GLOBAL_C
1043 typedef struct
1044 {
1045 #if     ALG_SHA1
1046     // SHA1 PCR
1047     BYTE    sha1Pcr[SHA1_DIGEST_SIZE];
1048 #endif
1049 #if     ALG_SHA256
1050     // SHA256 PCR
1051     BYTE    sha256Pcr[SHA256_DIGEST_SIZE];
1052 #endif
1053 #if     ALG_SHA384
1054     // SHA384 PCR
1055     BYTE    sha384Pcr[SHA384_DIGEST_SIZE];
1056 #endif
1057 #if     ALG_SHA512
1058     // SHA512 PCR
1059     BYTE    sha512Pcr[SHA512_DIGEST_SIZE];
1060 #endif
1061 #if     ALG_SM3_256
1062     // SHA256 PCR
1063     BYTE    sm3_256Pcr[SM3_256_DIGEST_SIZE];
1064 #endif
1065 } PCR;
1066
1067 typedef struct
```

```
1068   {
1069       unsigned int    stateSave : 1;              // if the PCR value should be
1070                                                  // saved in state save
1071       unsigned int    resetLocality : 5;          // The locality that the PCR
1072                                                  // can be reset
1073       unsigned int    extendLocality : 5;         // The locality that the PCR
1074                                                  // can be extend
1075   } PCR_Attributes;
1076
1077   EXTERN PCR          s_pcrs[IMPLEMENTATION_PCR];
1078
1079   #endif // PCR_C
1080
1081   //*****************************************************************************
1082   //*** From Session.c
1083   //*****************************************************************************
1084   #if defined SESSION_C || defined GLOBAL_C
1085   // Container for HMAC or policy session tracking information
1086   typedef struct
1087   {
1088       BOOL                occupied;
1089       SESSION             session;        // session structure
1090   } SESSION_SLOT;
1091
1092   EXTERN SESSION_SLOT     s_sessions[MAX_LOADED_SESSIONS];
1093
1094   //  The index in contextArray that has the value of the oldest saved session
1095   //  context. When no context is saved, this will have a value that is greater
1096   //  than or equal to MAX_ACTIVE_SESSIONS.
1097   EXTERN UINT32            s_oldestSavedSession;
1098
1099   // The number of available session slot openings.  When this is 1,
1100   // a session can't be created or loaded if the GAP is maxed out.
1101   // The exception is that the oldest saved session context can always
1102   // be loaded (assuming that there is a space in memory to put it)
1103   EXTERN int               s_freeSessionSlots;
1104
1105   #endif // SESSION_C
1106
1107   //*****************************************************************************
1108   //*** From IoBuffers.c
1109   //*****************************************************************************
1110   #if defined IO_BUFFER_C || defined GLOBAL_C
1111   // Each command function is allowed a structure for the inputs to the function and
1112   // a structure for the outputs. The command dispatch code unmarshals the input butter
1113   // to the command action input structure starting at the first byte of
1114   // s_actionIoBuffer. The value of s_actionIoAllocation is the number of UINT64 values
1115   // allocated. It is used to set the pointer for the response structure. The command
1116   // dispatch code will marshal the response values into the final output buffer.
1117   EXTERN UINT64    s_actionIoBuffer[768];      // action I/O buffer
1118   EXTERN UINT32    s_actionIoAllocation;       // number of UIN64 allocated for the
1119                                               // action input structure
1120   #endif // IO_BUFFER_C
1121
1122   //*****************************************************************************
1123   //*** From TPMFail.c
1124   //*****************************************************************************
1125   // This value holds the address of the string containing the name of the function
1126   // in which the failure occurred. This address value isn't useful for anything
1127   // other than helping the vendor to know in which file the failure  occurred.
1128   EXTERN BOOL      g_inFailureMode;       // Indicates that the TPM is in failure mode
1129   #if SIMULATION
1130   EXTERN BOOL      g_forceFailureMode;    // flag to force failure mode during test
1131   #endif
1132
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1133    typedef void(FailFunction)(const char *function, int line, int code);
1134    #if defined TPM_FAIL_C || defined GLOBAL_C
1135    EXTERN UINT32     s_failFunction;
1136    EXTERN UINT32     s_failLine;             // the line in the file at which
1137                                            // the error was signaled
1138    EXTERN UINT32     s_failCode;             // the error code used
1139
1140    EXTERN FailFunction    *LibFailCallback;
1141
1142    #endif // TPM_FAIL_C
1143
1144    //*****************************************************************************
1145    //*** From ACT_spt.c
1146    //*****************************************************************************
1147    // This value is used to indicate if an ACT has been updated since the last
1148    // TPM2_Startup() (one bit for each ACT). If the ACT is not updated
1149    // (TPM2_ACT_SetTimeout()) after a startup, then on each TPM2_Shutdown() the TPM will
1150    // save 1/2 of the current timer value. This prevents an attack on the ACT by saving
1151    // the counter and then running for a long period of time before doing a TPM Restart.
1152    // A quick TPM2_Shutdown() after each
1153    EXTERN UINT16                        s_ActUpdated;
1154
1155    //*****************************************************************************
1156    //*** From CommandCodeAttributes.c
1157    //*****************************************************************************
1158    // This array is instanced in CommandCodeAttributes.c when it includes
1159    // CommandCodeAttributes.h. Don't change the extern to EXTERN.
1160    extern  const  TPMA_CC               s_ccAttr[];
1161    extern  const  COMMAND_ATTRIBUTES    s_commandAttributes[];
1162
1163    #endif // GLOBAL_H
```

 Copyright © TCG 2006-2020

5.10 GpMacros.h

5.10.1 Introduction

This file is a collection of miscellaneous macros.

```
1  #ifndef GP_MACROS_H
2  #define GP_MACROS_H
3  #ifndef NULL
4  #define NULL 0
5  #endif
6  #include "swap.h"
7  #include "VendorString.h"
```

5.10.2 For Self-test

These macros are used in CryptUtil() to invoke the incremental self test.

```
8  #if SELF_TEST
9  #   define     TEST(alg) if(TEST_BIT(alg, g_toTest)) CryptTestAlgorithm(alg, NULL)
```

Use of TPM_ALG_NULL is reserved for RSAEP/RSADP testing. If someone is wanting to test a hash with that value, don't do it.

```
10 #   define     TEST_HASH(alg)                                                    \
11             if(TEST_BIT(alg, g_toTest)                                            \
12                 &&  (alg != ALG_NULL_VALUE))                                      \
13                 CryptTestAlgorithm(alg, NULL)
14 #else
15 #   define TEST(alg)
16 #   define TEST_HASH(alg)
17 #endif // SELF_TEST
```

5.10.3 For Failures

```
18 #if defined _POSIX_
19 #   define FUNCTION_NAME        0
20 #else
21 #   define FUNCTION_NAME        __FUNCTION__
22 #endif
23 #if !FAIL_TRACE
24 #   define FAIL(errorCode) (TpmFail(errorCode))
25 #   define LOG_FAILURE(errorCode) (TpmLogFailure(errorCode))
26 #else
27 #   define FAIL(errorCode)        TpmFail(FUNCTION_NAME, __LINE__, errorCode)
28 #   define LOG_FAILURE(errorCode) TpmLogFailure(FUNCTION_NAME, __LINE__, errorCode)
29 #endif
```

If implementation is using longjmp, then the call to TpmFail() does not return and the compiler will complain about unreachable code that comes after. To allow for not having longjmp, TpmFail() will return and the subsequent code will be executed. This macro accounts for the difference.

```
30 #ifndef NO_LONGJMP
31 #   define FAIL_RETURN(returnCode)
32 #   define TPM_FAIL_RETURN     NORETURN void
33 #else
34 #   define FAIL_RETURN(returnCode) return (returnCode)
35 #   define TPM_FAIL_RETURN     void
36 #endif
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

This macro tests that a condition is TRUE and puts the TPM into failure mode if it is not. If longjmp is being used, then the FAIL(FATAL_ERROR_) macro makes a call from which there is no return. Otherwise, it returns and the function will exit with the appropriate return code.

```
37  #define REQUIRE(condition, errorCode, returnCode)                              \
38      {                                                                          \
39          if(!!(condition))                                                      \
40          {                                                                      \
41              FAIL(FATAL_ERROR_errorCode);                                       \
42              FAIL_RETURN(returnCode);                                           \
43          }                                                                      \
44      }
45  #define PARAMETER_CHECK(condition, returnCode)                                 \
46      REQUIRE((condition), PARAMETER, returnCode)
47  #if (defined EMPTY_ASSERT) && (EMPTY_ASSERT != NO)
48  #   define pAssert(a)   ((void)0)
49  #else
50  #   define pAssert(a) {if(!(a)) FAIL(FATAL_ERROR_PARAMETER);}
51  #endif
```

5.10.4 Derived from Vendor-specific values

Values derived from vendor specific settings in TpmProfile.h

```
52  #define PCR_SELECT_MIN          ((PLATFORM_PCR+7)/8)
53  #define PCR_SELECT_MAX          ((IMPLEMENTATION_PCR+7)/8)
54  #define MAX_ORDERLY_COUNT       ((1 << ORDERLY_BITS) - 1)
55  #define RSA_MAX_PRIME           (MAX_RSA_KEY_BYTES / 2)
56  #define RSA_PRIVATE_SIZE        (RSA_MAX_PRIME * 5)
```

5.10.5 Compile-time Checks

In some cases, the relationship between two values may be dependent on things that change based on various selections like the chosen cryptographic libraries. It is possible that these selections will result in incompatible settings. These are often detectable by the compiler but it isn't always possible to do the check in the preprocessor code. For example, when the check requires use of **sizeof** then the preprocessor can't do the comparison. For these cases, we include a special macro that, depending on the compiler will generate a warning to indicate if the check always passes or always fails because it involves fixed constants. To run these checks, define COMPILER_CHECKS in TpmBuildSwitches.h

```
57  #if COMPILER_CHECKS
58  #   define  cAssert     pAssert
59  #else
60  #   define cAssert(value)
61  #endif
```

This is used commonly in the **Crypt** code as a way to keep listings from getting too long. This is not to save paper but to allow one to see more useful stuff on the screen at any given time.

```
62  #define     ERROR_RETURN(returnCode)                                           \
63      {                                                                          \
64           retVal = returnCode;                                                  \
65           goto Exit;                                                            \
66      }
67  #ifndef MAX
68  #  define MAX(a, b) ((a) > (b) ? (a) : (b))
69  #endif
70  #ifndef MIN
71  #  define MIN(a, b) ((a) < (b) ? (a) : (b))
72  #endif
73  #ifndef IsOdd
```

```
74    #  define IsOdd(a)         (((a) & 1) != 0)
75    #endif
76    #ifndef BITS_TO_BYTES
77    #  define BITS_TO_BYTES(bits)  (((bits) + 7) >> 3)
78    #endif
```

These are defined for use when the size of the vector being checked is known at compile time.

```
79    #define TEST_BIT(bit, vector)    TestBit((bit), (BYTE *)&(vector), sizeof(vector))
80    #define SET_BIT(bit, vector)     SetBit((bit), (BYTE *)&(vector), sizeof(vector))
81    #define CLEAR_BIT(bit, vector) ClearBit((bit), (BYTE *)&(vector), sizeof(vector))
```

The following definitions are used if they have not already been defined. The defaults for these settings are compatible with ISO/IEC 9899:2011 (E)

```
82    #ifndef LIB_EXPORT
83    #   define LIB_EXPORT
84    #   define LIB_IMPORT
85    #endif
86    #ifndef NORETURN
87    #   define NORETURN _Noreturn
88    #endif
89    #ifndef NOT_REFERENCED
90    #   define NOT_REFERENCED(x = x)    ((void) (x))
91    #endif
92    #define STD_RESPONSE_HEADER (sizeof(TPM_ST) + sizeof(UINT32) + sizeof(TPM_RC))
93    #define JOIN(x, y) x##y
94    #define JOIN3(x, y, z) x##y##z
95    #define CONCAT(x, y) JOIN(x, y)
96    #define CONCAT3(x, y, z) JOIN3(x,y,z)
```

If CONTEXT_INTEGRITY_HASH_ALG is defined, then the vendor is using the old style table. Otherwise, pick the **strongest** implemented hash algorithm as the context hash.

```
97    #ifndef CONTEXT_HASH_ALGORITHM
98    #   if defined ALG_SHA512 && ALG_SHA512 == YES
99    #       define CONTEXT_HASH_ALGORITHM    SHA512
100   #   elif defined ALG_SHA384 && ALG_SHA384 == YES
101   #       define CONTEXT_HASH_ALGORITHM    SHA384
102   #   elif defined ALG_SHA256 && ALG_SHA256 == YES
103   #       define CONTEXT_HASH_ALGORITHM    SHA256
104   #   elif defined ALG_SM3_256 && ALG_SM3_256 == YES
105   #       define CONTEXT_HASH_ALGORITHM    SM3_256
106   #   elif defined ALG_SHA1 && ALG_SHA1 == YES
107   #       define CONTEXT_HASH_ALGORITHM  SHA1
108   #   endif
109   #   define CONTEXT_INTEGRITY_HASH_ALG  CONCAT(TPM_ALG_, CONTEXT_HASH_ALGORITHM)
110   #endif
111   #ifndef CONTEXT_INTEGRITY_HASH_SIZE
112   #define CONTEXT_INTEGRITY_HASH_SIZE CONCAT(CONTEXT_HASH_ALGORITHM, _DIGEST_SIZE)
113   #endif
114   #if     ALG_RSA
115   #define     RSA_SECURITY_STRENGTH (MAX_RSA_KEY_BITS >= 15360 ? 256 :               \
116                                      (MAX_RSA_KEY_BITS >=  7680 ? 192 :              \
117                                      (MAX_RSA_KEY_BITS >=  3072 ? 128 :              \
118                                      (MAX_RSA_KEY_BITS >=  2048 ? 112 :              \
119                                      (MAX_RSA_KEY_BITS >=  1024 ?  80 :   0)))))
120   #else
121   #define     RSA_SECURITY_STRENGTH   0
122   #endif // ALG_RSA
123   #if     ALG_ECC
124   #define     ECC_SECURITY_STRENGTH (MAX_ECC_KEY_BITS >= 521 ? 256 :                 \
125                                      (MAX_ECC_KEY_BITS >= 384 ? 192 :                \
126                                      (MAX_ECC_KEY_BITS >= 256 ? 128 : 0)))
```

```
127 #else
128 #define      ECC_SECURITY_STRENGTH    0
129 #endif // ALG_ECC
130 #define      MAX_ASYM_SECURITY_STRENGTH                                              \
131                          MAX(RSA_SECURITY_STRENGTH, ECC_SECURITY_STRENGTH)
132 #define      MAX_HASH_SECURITY_STRENGTH   ((CONTEXT_INTEGRITY_HASH_SIZE * 8) / 2)
```

Unless some algorithm is broken...

```
133 #define      MAX_SYM_SECURITY_STRENGTH    MAX_SYM_KEY_BITS
134 #define MAX_SECURITY_STRENGTH_BITS                                                   \
135                          MAX(MAX_ASYM_SECURITY_STRENGTH,                             \
136                          MAX(MAX_SYM_SECURITY_STRENGTH,                              \
137                              MAX_HASH_SECURITY_STRENGTH))
```

This is the size that was used before the 1.38 errata requiring that P1.14.4 be followed

```
138 #define PROOF_SIZE         CONTEXT_INTEGRITY_HASH_SIZE
```

As required by P1.14.4

```
139 #define COMPLIANT_PROOF_SIZE                                                         \
140             (MAX(CONTEXT_INTEGRITY_HASH_SIZE, (2 * MAX_SYM_KEY_BYTES)))
```

As required by P1.14.3.1

```
141 #define COMPLIANT_PRIMARY_SEED_SIZE                                                  \
142     BITS_TO_BYTES(MAX_SECURITY_STRENGTH_BITS * 2)
```

This is the pre-errata version

```
143 #ifndef PRIMARY_SEED_SIZE
144 #   define PRIMARY_SEED_SIZE     PROOF_SIZE
145 #endif
146 #if USE_SPEC_COMPLIANT_PROOFS
147 #   undef PROOF_SIZE
148 #   define PROOF_SIZE            COMPLIANT_PROOF_SIZE
149 #   undef PRIMARY_SEED_SIZE
150 #   define PRIMARY_SEED_SIZE     COMPLIANT_PRIMARY_SEED_SIZE
151 #endif   // USE_SPEC_COMPLIANT_PROOFS
152 #if !SKIP_PROOF_ERRORS
153 #   if PROOF_SIZE < COMPLIANT_PROOF_SIZE
154 #       error "PROOF_SIZE is not compliant with TPM specification"
155 #   endif
156 #   if PRIMARY_SEED_SIZE < COMPLIANT_PRIMARY_SEED_SIZE
157 #       error Non-compliant PRIMARY_SEED_SIZE
158 #   endif
159 #endif // !SKIP_PROOF_ERRORS
```

If CONTEXT_ENCRYPT_ALG is defined, then the vendor is using the old style table

```
160 #if defined CONTEXT_ENCRYPT_ALG
161 #   undef CONTEXT_ENCRYPT_ALGORITHM
162 #   if  CONTEXT_ENCRYPT_ALG == ALG_AES_VALUE
163 #       define CONTEXT_ENCRYPT_ALGORITHM    AES
164 #   elif CONTEXT_ENCRYPT_ALG == ALG_SM4_VALUE
165 #       define CONTEXT_ENCRYPT_ALGORITHM    SM4
166 #   elif CONTEXT_ENCRYPT_ALG == ALG_CAMELLIA_VALUE
167 #       define CONTEXT_ENCRYPT_ALGORITHM    CAMELLIA
168 #   elif CONTEXT_ENCRYPT_ALG == ALG_TDES_VALUE
169 #   error Are you kidding?
170 #   else
171 #       error Unknown value for CONTEXT_ENCRYPT_ALG
```

 Copyright © TCG 2006-2020

```
172  #   endif // CONTEXT_ENCRYPT_ALG == ALG_AES_VALUE
173  #else
174  #   define CONTEXT_ENCRYPT_ALG                                          \
175              CONCAT3(ALG_, CONTEXT_ENCRYPT_ALGORITHM, _VALUE)
176  #endif  // CONTEXT_ENCRYPT_ALG
177  #define CONTEXT_ENCRYPT_KEY_BITS                                       \
178                  CONCAT(CONTEXT_ENCRYPT_ALGORITHM, _MAX_KEY_SIZE_BITS)
179  #define CONTEXT_ENCRYPT_KEY_BYTES       ((CONTEXT_ENCRYPT_KEY_BITS+7)/8)
```

This is updated to follow the requirement of P2 that the label not be larger than 32 bytes.

```
180  #ifndef LABEL_MAX_BUFFER
181  #define LABEL_MAX_BUFFER MIN(32, MAX(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE))
182  #endif
```

This bit is used to indicate that an authorization ticket expires on TPM Reset and TPM Restart. It is added to the timeout value returned by TPM2_PoliySigned() and TPM2_PolicySecret() and used by TPM2_PolicyTicket(). The timeout value is relative to Time (g_time). Time is reset whenever the TPM loses power and cannot be moved forward by the user (as can Clock). *g_time* is a 64-bit value expressing time in ms. Stealing the MSb for a flag means that the TPM needs to be reset at least once every 292,471,208 years rather than once every 584,942,417 years.

```
183  #define EXPIRATION_BIT ((UINT64)1 << 63)
```

Check for consistency of the bit ordering of bit fields

```
184  #if BIG_ENDIAN_TPM && MOST_SIGNIFICANT_BIT_0 && USE_BIT_FIELD_STRUCTURES
185  #   error "Settings not consistent"
186  #endif
```

These macros are used to handle the variation in handling of bit fields. If

```
187  #if USE_BIT_FIELD_STRUCTURES // The default, old version, with bit fields
188  #   define IS_ATTRIBUTE(a, type, b)        ((a.b) != 0)
189  #   define SET_ATTRIBUTE(a, type, b)       (a.b = SET)
190  #   define CLEAR_ATTRIBUTE(a, type, b)     (a.b = CLEAR)
191  #   define GET_ATTRIBUTE(a, type, b)       (a.b)
192  #   define TPMA_ZERO_INITIALIZER()         {0}
193  #else
194  #   define IS_ATTRIBUTE(a, type, b)        ((a & type##_##b) != 0)
195  #   define SET_ATTRIBUTE(a, type, b)       (a |= type##_##b)
196  #   define CLEAR_ATTRIBUTE(a, type, b)     (a &= ~type##_##b)
197  #   define GET_ATTRIBUTE(a, type, b)                                    \
198          (type)((a & type##_##b) >> type##_##b##_SHIFT)
199  #   define TPMA_ZERO_INITIALIZER()         (0)
200  #endif
201  #define VERIFY(_X) if(!(_X)) goto Error
```

These macros determine if the values in this file are referenced or instanced. Global.c defines GLOBAL_C so all the values in this file will be instanced in Global.obj. For all other files that include this file, the values will simply be external references. For constants, there can be an initializer.

```
202  #ifdef GLOBAL_C
203  #define EXTERN
204  #define INITIALIZER(_value_)  = _value_
205  #else
206  #define EXTERN  extern
207  #define INITIALIZER(_value_)
208  #endif
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

This macro will create an OID. All OIDs are in DER form with a first octet of 0x06 indicating an OID fallowed by an octet indicating the number of octets in the rest of the OID. This allows a user of this OID to know how much/little to copy.

```
209     #define MAKE_OID(NAME)                                                              \
210             EXTERN  const BYTE OID##NAME[] INITIALIZER({OID##NAME##_VALUE})
```

This definition is moved from TpmProfile.h because it is not actually vendor- specific. It has to be the same size as the *sequence* parameter of a TPMS_CONTEXT and that is a UINT64. So, this is an invariant value

```
211     #define CONTEXT_COUNTER         UINT64
212     #endif // GP_MACROS_H
```

 Copyright © TCG 2006-2020

5.11 InternalRoutines.h

```
1  #ifndef     INTERNAL_ROUTINES_H
2  #define     INTERNAL_ROUTINES_H
3  #if !defined _LIB_SUPPORT_H_ && !defined _TPM_H_
4  #error "Should not be called"
5  #endif
```

DRTM functions

```
6  #include "_TPM_Hash_Start_fp.h"
7  #include "_TPM_Hash_Data_fp.h"
8  #include "_TPM_Hash_End_fp.h"
```

Internal subsystem functions

```
9  #include "Object_fp.h"
10 #include "Context_spt_fp.h"
11 #include "Object_spt_fp.h"
12 #include "Entity_fp.h"
13 #include "Session_fp.h"
14 #include "Hierarchy_fp.h"
15 #include "NvReserved_fp.h"
16 #include "NvDynamic_fp.h"
17 #include "NV_spt_fp.h"
18 #include "ACT_spt_fp.h"
19 #include "PCR_fp.h"
20 #include "DA_fp.h"
21 #include "TpmFail_fp.h"
22 #include "SessionProcess_fp.h"
```

Internal support functions

```
23 #include "CommandCodeAttributes_fp.h"
24 #include "Marshal.h"
25 #include "Time_fp.h"
26 #include "Locality_fp.h"
27 #include "PP_fp.h"
28 #include "CommandAudit_fp.h"
29 #include "Manufacture_fp.h"
30 #include "Handle_fp.h"
31 #include "Power_fp.h"
32 #include "Response_fp.h"
33 #include "CommandDispatcher_fp.h"
34 #ifdef CC_AC_Send
35 #   include "AC_spt_fp.h"
36 #endif // CC_AC_Send
```

Miscellaneous

```
37 #include "Bits_fp.h"
38 #include "AlgorithmCap_fp.h"
39 #include "PropertyCap_fp.h"
40 #include "IoBuffers_fp.h"
41 #include "Memory_fp.h"
42 #include "ResponseCodeProcessing_fp.h"
```

Internal cryptographic functions

```
43 #include "BnConvert_fp.h"
44 #include "BnMath_fp.h"
45 #include "BnMemory_fp.h"
```

 Copyright © TCG 2006-2020

```
46 #include "Ticket_fp.h"
47 #include "CryptUtil_fp.h"
48 #include "CryptHash_fp.h"
49 #include "CryptSym_fp.h"
50 #include "CryptDes_fp.h"
51 #include "CryptPrime_fp.h"
52 #include "CryptRand_fp.h"
53 #include "CryptSelfTest_fp.h"
54 #include "MathOnByteBuffers_fp.h"
55 #include "CryptSym_fp.h"
56 #include "AlgorithmTests_fp.h"
57 #if ALG_RSA
58 #include "CryptRsa_fp.h"
59 #include "CryptPrimeSieve_fp.h"
60 #endif
61 #if ALG_ECC
62 #include "CryptEccMain_fp.h"
63 #include "CryptEccSignature_fp.h"
64 #include "CryptEccKeyExchange_fp.h"
65 #endif
66 #if CC_MAC || CC_MAC_Start
67 #   include "CryptSmac_fp.h"
68 #   if ALG_CMAC
69 #       include "CryptCmac_fp.h"
70 #   endif
71 #endif
```

Support library

```
72 #include "SupportLibraryFunctionPrototypes_fp.h"
```

Linkage to platform functions

```
73 #include "Platform_fp.h"
74 #endif
```

5.12 LibSupport.h

This header file is used to select the library code that gets included in the TPM build.

```
1   #ifndef _LIB_SUPPORT_H_
2   #define _LIB_SUPPORT_H_
3   #ifndef RADIX_BITS
4   #   if defined(__x86_64__) || defined(__x86_64)
    \
5           || defined(__amd64__) || defined(__amd64) || defined(_WIN64) ||
    defined(_M_X64) \
6           || defined(_M_ARM64) || defined(__aarch64__)
7   #       define RADIX_BITS                      64
8   #   elif defined(__i386__) || defined(__i386) || defined(i386)
    \
9           || defined(_WIN32) || defined(_M_IX86)
    \
10          || defined(_M_ARM) || defined(__arm__) || defined(__thumb__)
11  #       define RADIX_BITS                      32
12  #   else
13  #       error Unable to determine RADIX_BITS from compiler environment
14  #   endif
15  #endif // RADIX_BITS
```

These macros use the selected libraries to the proper include files.

```
16  #define LIB_QUOTE(_STRING_) #_STRING_
17  #define LIB_INCLUDE2(_LIB_, _TYPE_) LIB_QUOTE(_LIB_/TpmTo##_LIB_##_TYPE_.h)
18  #define LIB_INCLUDE(_LIB_, _TYPE_) LIB_INCLUDE2(_LIB_, _TYPE_)
```

Include the options for hashing and symmetric. Defer the load of the math package Until the bignum parameters are defined.

```
19  #include LIB_INCLUDE(SYM_LIB, Sym)
20  #include LIB_INCLUDE(HASH_LIB, Hash)
21  #undef MIN
22  #undef MAX
23  #endif // _LIB_SUPPORT_H_
```

5.13 MinMax.h

```
1   #ifndef _MIN_MAX_H_
2   #define _MIN_MAX_H_
3   #ifndef MAX
4   #define MAX(a, b) ((a) > (b) ? (a) : (b))
5   #endif
6   #ifndef MIN
7   #define MIN(a, b) ((a) < (b) ? (a) : (b))
8   #endif
9   #endif // _MIN_MAX_H_
```

5.14 NV.h

5.14.1 Index Type Definitions

These definitions allow the same code to be used pre and post 1.21. The main action is to redefine the index type values from the bit values. Use TPM_NT_ORDINARY to indicate if the TPM_NT type is defined

```
1   #ifndef     _NV_H_
2   #define     _NV_H_
3   #ifdef      TPM_NT_ORDINARY
```

If TPM_NT_ORDINARY is defined, then the TPM_NT field is present in a TPMA_NV

```
4   #   define GET_TPM_NT(attributes) GET_ATTRIBUTE(attributes, TPMA_NV, TPM_NT)
5   #else
```

If TPM_NT_ORDINARY is not defined, then need to synthesize it from the attributes

```
6   #   define GetNv_TPM_NV(attributes)                              \
7           (   IS_ATTRIBUTE(attributes, TPMA_NV, COUNTER)           \
8           +   (IS_ATTRIBUTE(attributes, TPMA_NV, BITS) << 1)       \
9           +   (IS_ATTRIBUTE(attributes, TPMA_NV, EXTEND) << 2)     \
10          )
11  #   define TPM_NT_ORDINARY (0)
12  #   define TPM_NT_COUNTER  (1)
13  #   define TPM_NT_BITS     (2)
14  #   define TPM_NT_EXTEND   (4)
15  #endif
```

5.14.2 Attribute Macros

These macros are used to isolate the differences in the way that the index type changed in version 1.21 of the specification

```
16  #   define IsNvOrdinaryIndex(attributes)                         \
17                  (GET_TPM_NT(attributes) == TPM_NT_ORDINARY)
18  #   define  IsNvCounterIndex(attributes)                         \
19                  (GET_TPM_NT(attributes) == TPM_NT_COUNTER)
20  #   define  IsNvBitsIndex(attributes)                            \
21                 (GET_TPM_NT(attributes) == TPM_NT_BITS)
22  #   define  IsNvExtendIndex(attributes)                          \
23                  (GET_TPM_NT(attributes) == TPM_NT_EXTEND)
24  #ifdef TPM_NT_PIN_PASS
25  #   define  IsNvPinPassIndex(attributes)                         \
26                  (GET_TPM_NT(attributes) == TPM_NT_PIN_PASS)
27  #endif
28  #ifdef TPM_NT_PIN_FAIL
29  #   define  IsNvPinFailIndex(attributes)                         \
30                  (GET_TPM_NT(attributes) == TPM_NT_PIN_FAIL)
31  #endif
32  typedef struct {
33      UINT32      size;
34      TPM_HANDLE  handle;
35  } NV_ENTRY_HEADER;
36  #define NV_EVICT_OBJECT_SIZE        \
37       (sizeof(UINT32)   + sizeof(TPM_HANDLE) + sizeof(OBJECT))
38  #define NV_INDEX_COUNTER_SIZE       \
39      (sizeof(UINT32) + sizeof(NV_INDEX) + sizeof(UINT64))
40  #define NV_RAM_INDEX_COUNTER_SIZE   \
41      (sizeof(NV_RAM_HEADER) + sizeof(UINT64))
```

```
42  typedef struct {
43      UINT32          size;
44      TPM_HANDLE      handle;
45      TPMA_NV         attributes;
46  } NV_RAM_HEADER;
```

Defines the end-of-list marker for NV. The list terminator is a UINT32 of zero, followed by the current value of *s_maxCounter* which is a 64-bit value. The structure is defined as an array of 3 UINT32 values so that there is no padding between the UINT32 list end marker and the UINT64 *maxCounter* value.

```
47  typedef UINT32 NV_LIST_TERMINATOR[3];
```

5.14.3 Orderly RAM Values

The following defines are for accessing orderly RAM values. This is the initialize for the RAM reference iterator.

```
48  #define     NV_RAM_REF_INIT         0
```

This is the starting address of the RAM space used for orderly data

```
49  #define     RAM_ORDERLY_START                 \
50                  (&s_indexOrderlyRam[0])
```

This is the offset within NV that is used to save the orderly data on an orderly shutdown.

```
51  #define     NV_ORDERLY_START                  \
52                  (NV_INDEX_RAM_DATA)
```

This is the end of the orderly RAM space. It is actually the first byte after the last byte of orderly RAM data

```
53  #define     RAM_ORDERLY_END                   \
54                  (RAM_ORDERLY_START + sizeof(s_indexOrderlyRam))
```

This is the end of the orderly space in NV memory. As with RAM_ORDERLY_END, it is actually the offset of the first byte after the end of the NV orderly data.

```
55  #define     NV_ORDERLY_END                    \
56                  (NV_ORDERLY_START + sizeof(s_indexOrderlyRam))
```

Macro to check that an orderly RAM address is with range.

```
57  #define ORDERLY_RAM_ADDRESS_OK(start, offset)       \
58          ((start >= RAM_ORDERLY_START) && ((start + offset - 1) < RAM_ORDERLY_END))
59  #define RETURN_IF_NV_IS_NOT_AVAILABLE               \
60  {                                                   \
61      if(g_NvStatus != TPM_RC_SUCCESS)                \
62          return g_NvStatus;                          \
63  }
```

Routinely have to clear the orderly flag and fail if the NV is not available so that it can be cleared.

```
64  #define RETURN_IF_ORDERLY                           \
65  {                                                   \
66      if(NvClearOrderly() != TPM_RC_SUCCESS)          \
67          return g_NvStatus;                          \
68  }
69  #define NV_IS_AVAILABLE      (g_NvStatus == TPM_RC_SUCCESS)
70  #define IS_ORDERLY(value)    (value < SU_DA_USED_VALUE)
71  #define NV_IS_ORDERLY        (IS_ORDERLY(gp.orderlyState))
```

 Copyright © TCG 2006-2020

Macro to set the NV UPDATE_TYPE. This deals with the fact that the update is possibly a combination of UT_NV and UT_ORDERLY.

```
72    #define SET_NV_UPDATE(type)     g_updateNV |= (type)
73    #endif  // _NV_H_
```

 Copyright © TCG 2006-2020

5.15 TPMB.h

This file contains extra TPM2B structures

```
1    #ifndef _TPMB_H
2    #define _TPMB_H
```

TPM2B Types

```
3    typedef struct {
4        UINT16          size;
5        BYTE            buffer[1];
6    } TPM2B, *P2B;
7    typedef const TPM2B     *PC2B;
```

This macro helps avoid having to type in the structure in order to create a new TPM2B type that is used in a function.

```
8    #define TPM2B_TYPE(name, bytes)                 \
9        typedef union {                             \
10           struct  {                               \
11               UINT16  size;                       \
12               BYTE    buffer[(bytes)];            \
13           } t;                                    \
14           TPM2B   b;                              \
15       } TPM2B_##name
```

This macro defines a TPM2B with a constant character value. This macro sets the size of the string to the size minus the terminating zero byte. This lets the user of the label add their terminating 0. This method is chosen so that existing code that provides a label will continue to work correctly. Macro to instance and initialize a TPM2B value

```
16   #define TPM2B_INIT(TYPE, name)  \
17       TPM2B_##TYPE    name = {sizeof(name.t.buffer), {0}}
18   #define TPM2B_BYTE_VALUE(bytes) TPM2B_TYPE(bytes##_BYTE_VALUE, bytes)
19   #endif
```

5.16 Tpm.h

Root header file for building any TPM.lib code

```
1  #ifndef     _TPM_H_
2  #define     _TPM_H_
3  #include "TpmBuildSwitches.h"
4  #include "BaseTypes.h"
5  #include "TPMB.h"
6  #include "MinMax.h"
7  #include "TpmProfile.h"
8  #include "TpmAlgorithmDefines.h"
9  #include "LibSupport.h"         // Types from the library. These need to come before
10                                 // Global.h because some of the structures in
11                                 // that file depend on the structures used by the
12                                 // cryptographic libraries.
13 #include "GpMacros.h"           // Define additional macros
14 #include "Global.h"             // Define other TPM types
15 #include "InternalRoutines.h"   // Function prototypes
16 #endif // _TPM_H_
```

 Copyright © TCG 2006-2020

5.17 TpmBuildSwitches.h

This file contains the build switches. This contains switches for multiple versions of the crypto-library so some may not apply to your environment.

The switches are guarded so that they can either be set on the command line or set here. If the switch is listed on the command line (-DSOME_SWITCH) with NO setting, then the switch will be set to YES. If the switch setting is not on the command line or if the setting is other than YES or NO, then the switch will be set to the default value. The default can either be YES or NO as indicated on each line where the default is selected.

A caution. Do not try to test these macros by inserting #defines in this file. For some curious reason, a variable set on the command line with no setting will have a value of 1. An #if SOME_VARIABLE will work if the variable is not defined or is defined on the command line with no initial setting. However, a "#define SOME_VARIABLE" is a null string and when used in "#if SOME_VARIABLE" will not be a proper expression If you want to test various switches, either use the command line or change the default.

```
1   #ifndef _TPM_BUILD_SWITCHES_H_
2   #define _TPM_BUILD_SWITCHES_H_
3   #undef YES
4   #define YES 1
5   #undef NO
6   #define NO 0
```

Allow the command line to specify a **profile** file

```
7   #ifdef PROFILE
8   #   define PROFILE_QUOTE(a) #a
9   #   define PROFILE_INCLUDE(a) PROFILE_QUOTE(a)
10  #   include PROFILE_INCLUDE(PROFILE)
11  #endif
```

Need an unambiguous definition for DEBUG. Don't change this

```
12  #ifndef DEBUG
13  #   ifdef NDEBUG
14  #       define  DEBUG   NO
15  #   else
16  #       define  DEBUG   YES
17  #   endif
18  #elif (DEBUG != NO) && (DEBUG != YES)
19  #   undef   DEBUG
20  #   define  DEBUG                        YES     // Default: Either YES or NO
21  #endif
22  #include "CompilerDependencies.h"
```

This definition is required for the re-factored code

```
23  #if (!defined USE_BN_ECC_DATA)                                               \
24                  || ((USE_BN_ECC_DATA != NO) && (USE_BN_ECC_DATA != YES))
25  #   undef   USE_BN_ECC_DATA
26  #   define  USE_BN_ECC_DATA              YES     // Default: Either YES or NO
27  #endif
```

The SIMULATION switch allows certain other macros to be enabled. The things that can be enabled in a simulation include key caching, reproducible **random** sequences, instrumentation of the RSA key generation process, and certain other debug code. SIMULATION Needs to be defined as either YES or NO. This grouping of macros will make sure that it is set correctly. A simulated TPM would include a Virtual TPM. The interfaces for a Virtual TPM should be modified from the standard ones in the Simulator project.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

If SIMULATION is in the compile parameters without modifiers, make SIMULATION == YES

```
28  #if !(defined SIMULATION) || ((SIMULATION != NO) && (SIMULATION != YES))
29  #   undef   SIMULATION
30  #   define  SIMULATION                 YES     // Default: Either YES or NO
31  #endif
```

Define this to run the function that checks the compatibility between the chosen big number math library and the TPM code. Not all ports use this.

```
32  #if !(defined LIBRARY_COMPATIBILITY_CHECK)                                  \
33          || (( LIBRARY_COMPATIBILITY_CHECK != NO)                            \
34              && (LIBRARY_COMPATIBILITY_CHECK != YES))
35  #   undef   LIBRARY_COMPATIBILITY_CHECK
36  #   define  LIBRARY_COMPATIBILITY_CHECK YES     // Default: Either YES or NO
37  #endif
38  #if !(defined FIPS_COMPLIANT) || ((FIPS_COMPLIANT != NO) && (FIPS_COMPLIANT != YES))
39  #   undef   FIPS_COMPLIANT
40  #   define  FIPS_COMPLIANT             YES     // Default: Either YES or NO
41  #endif
```

Definition to allow alternate behavior for non-orderly startup. If there is a chance that the TPM could not update *failedTries*

```
42  #if !(defined USE_DA_USED) || ((USE_DA_USED != NO) && (USE_DA_USED != YES))
43  #   undef   USE_DA_USED
44  #   define  USE_DA_USED                YES     // Default: Either YES or NO
45  #endif
```

Define TABLE_DRIVEN_DISPATCH to use tables rather than case statements for command dispatch and handle unmarshaling

```
46  #if !(defined TABLE_DRIVEN_DISPATCH)                                        \
47          || ((TABLE_DRIVEN_DISPATCH != NO) && (TABLE_DRIVEN_DISPATCH != YES))
48  #   undef   TABLE_DRIVEN_DISPATCH
49  #   define  TABLE_DRIVEN_DISPATCH      YES     // Default: Either YES or NO
50  #endif
```

This switch is used to enable the self-test capability in AlgorithmTests.c

```
51  #if !(defined SELF_TEST) || ((SELF_TEST != NO) && (SELF_TEST != YES))
52  #   undef   SELF_TEST
53  #   define  SELF_TEST                  YES     // Default: Either YES or NO
54  #endif
```

Enable the generation of RSA primes using a sieve.

```
55  #if !(defined RSA_KEY_SIEVE) || ((RSA_KEY_SIEVE != NO) && (RSA_KEY_SIEVE != YES))
56  #   undef   RSA_KEY_SIEVE
57  #   define  RSA_KEY_SIEVE              YES     // Default: Either YES or NO
58  #endif
```

Enable the instrumentation of the sieve process. This is used to tune the sieve variables.

```
59  #if RSA_KEY_SIEVE && SIMULATION
60  #   if !(defined RSA_INSTRUMENT)                                            \
61              || ((RSA_INSTRUMENT != NO) && (RSA_INSTRUMENT != YES))
62  #       undef   RSA_INSTRUMENT
63  #       define  RSA_INSTRUMENT         NO      // Default: Either YES or NO
64  #   endif
65  #endif
```

This switch enables the RNG state save and restore

```
66  #if !(defined _DRBG_STATE_SAVE)                                                    \
67      || ((_DRBG_STATE_SAVE != NO) && (_DRBG_STATE_SAVE != YES))
68  #   undef    _DRBG_STATE_SAVE
69  #   define   _DRBG_STATE_SAVE           YES     // Default: Either YES or NO
70  #endif
```

Switch added to support packed lists that leave out space associated with unimplemented commands. Comment this out to use linear lists.

NOTE: if vendor specific commands are present, the associated list is always in compressed form.

```
71  #if !(defined COMPRESSED_LISTS)                                                    \
72      || ((COMPRESSED_LISTS != NO) && (COMPRESSED_LISTS != YES))
73  #   undef    COMPRESSED_LISTS
74  #   define   COMPRESSED_LISTS           YES     // Default: Either YES or NO
75  #endif
```

This switch indicates where clock epoch value should be stored. If this value defined, then it is assumed that the timer will change at any time so the nonce should be a random number kept in RAM. When it is not defined, then the timer only stops during power outages.

```
76  #if !(defined CLOCK_STOPS) || ((CLOCK_STOPS != NO) && (CLOCK_STOPS != YES))
77  #   undef    CLOCK_STOPS
78  #   define   CLOCK_STOPS                NO      // Default: Either YES or NO
79  #endif
```

This switch allows use of #defines in place of pass-through marshaling or unmarshaling code. A pass-through function just calls another function to do the required function and does no parameter checking of its own. The table-driven dispatcher calls directly to the lowest level marshaling/unmarshaling code and by-passes any pass-through functions.

```
80  #if (defined USE_MARSHALING_DEFINES) && (USE_MARSHALING_DEFINES != NO)
81  #   undef    USE_MARSHALING_DEFINES
82  #   define   USE_MARSHALING_DEFINES     YES
83  #else
84  #   define USE_MARSHALING_DEFINES      YES     // Default: Either YES or NO
85  #endif
```

The switches in this group can only be enabled when doing debug during simulation

```
86  #if SIMULATION && DEBUG
```

This forces the use of a smaller context slot size. This reduction reduces the range of the epoch allowing the tester to force the epoch to occur faster than the normal defined in TpmProfile.h

```
87  #   if !(defined CONTEXT_SLOT)
88  #       define CONTEXT_SLOT             UINT8
89  #   endif
```

Enables use of the key cache. Default is YES

```
90  #   if !(defined USE_RSA_KEY_CACHE)                                                \
91      || ((USE_RSA_KEY_CACHE != NO) && (USE_RSA_KEY_CACHE != YES))
92  #       undef    USE_RSA_KEY_CACHE
93  #       define   USE_RSA_KEY_CACHE      YES     // Default: Either YES or NO
94  #   endif
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

Enables use of a file to store the key cache values so that the TPM will start faster during debug. Default for this is YES

```
 95     #   if USE_RSA_KEY_CACHE
 96     #       if !(defined USE_KEY_CACHE_FILE)                                        \
 97                 || ((USE_KEY_CACHE_FILE != NO) && (USE_KEY_CACHE_FILE != YES))
 98     #           undef    USE_KEY_CACHE_FILE
 99     #           define   USE_KEY_CACHE_FILE  YES     // Default: Either YES or NO
100     #       endif
101     #   else
102     #       undef   USE_KEY_CACHE_FILE
103     #       define  USE_KEY_CACHE_FILE      NO
104     #   endif   // USE_RSA_KEY_CACHE
```

This provides fixed seeding of the RNG when doing debug on a simulator. This should allow consistent results on test runs as long as the input parameters to the functions remains the same. There is no default value.

```
105     #   if !(defined USE_DEBUG_RNG) || ((USE_DEBUG_RNG != NO) && (USE_DEBUG_RNG != YES))
106     #       undef   USE_DEBUG_RNG
107     #       define  USE_DEBUG_RNG           YES     // Default: Either YES or NO
108     #   endif
```

Don't change these. They are the settings needed when not doing a simulation and not doing debug. Can't use the key cache except during debug. Otherwise, all of the key values end up being the same

```
109     #else
110     #   define USE_RSA_KEY_CACHE           NO
111     #   define USE_RSA_KEY_CACHE_FILE      NO
112     #   define USE_DEBUG_RNG               NO
113     #endif  // DEBUG && SIMULATION
114     #if DEBUG
```

In some cases, the relationship between two values may be dependent on things that change based on various selections like the chosen cryptographic libraries. It is possible that these selections will result in incompatible settings. These are often detectable by the compiler but it isn't always possible to do the check in the preprocessor code. For example, when the check requires use of 'sizeof()' then the preprocessor can't do the comparison. For these cases, we include a special macro that, depending on the compiler will generate a warning to indicate if the check always passes or always fails because it involves fixed constants. To run these checks, define COMPILER_CHECKS.

```
115     #   if !(defined COMPILER_CHECKS)                                           \
116             || ((COMPILER_CHECKS != NO) && (COMPILER_CHECKS != YES))
117     #       undef   COMPILER_CHECKS
118     #       define  COMPILER_CHECKS         NO      // Default: Either YES or NO
119     #   endif
```

Some of the values (such as sizes) are the result of different options set in TpmProfile.h. The combination might not be consistent. A function is defined (TpmSizeChecks()) that is used to verify the sizes at run time. To enable the function, define this parameter.

```
120     #   if !(defined RUNTIME_SIZE_CHECKS)                                       \
121         || ((RUNTIME_SIZE_CHECKS != NO) && (RUNTIME_SIZE_CHECKS != YES))
122     #       undef RUNTIME_SIZE_CHECKS
123     #       define RUNTIME_SIZE_CHECKS      YES      // Default: Either YES or NO
124     #   endif
```

If doing debug, can set the DRBG to print out the intermediate test values. Before enabling this, make sure that the dbgDumpMemBlock() function has been added someplace (preferably, somewhere in CryptRand.c)

```
125  #   if !(defined DRBG_DEBUG_PRINT)                                                    \
126      || ((DRBG_DEBUG_PRINT != NO) && (DRBG_DEBUG_PRINT != YES))
127  #       undef   DRBG_DEBUG_PRINT
128  #       define  DRBG_DEBUG_PRINT        NO      // Default: Either YES or NO
129  #   endif
```

If an assertion event it not going to produce any trace information (function and line number) then make FAIL_TRACE == NO

```
130  #   if !(defined FAIL_TRACE) || ((FAIL_TRACE != NO) && (FAIL_TRACE != YES))
131  #       undef   FAIL_TRACE
132  #       define  FAIL_TRACE              YES     // Default: Either YES or NO
133  #   endif
134  #endif // DEBUG
```

Indicate if the implementation is going to give lockout time credit for time up to the last orderly shutdown.

```
135  #if !(defined ACCUMULATE_SELF_HEAL_TIMER)                                                 \
136      || ((ACCUMULATE_SELF_HEAL_TIMER != NO) && (ACCUMULATE_SELF_HEAL_TIMER != YES))
137  #   undef   ACCUMULATE_SELF_HEAL_TIMER
138  #   define  ACCUMULATE_SELF_HEAL_TIMER  YES     // Default: Either YES or NO
139  #endif
```

Indicates if the implementation is to compute the sizes of the proof and primary seed size values based on the implemented algorithms.

```
140  #if !(defined USE_SPEC_COMPLIANT_PROOFS)                                                  \
141      || ((USE_SPEC_COMPLIANT_PROOFS != NO) && (USE_SPEC_COMPLIANT_PROOFS != YES))
142  #   undef   USE_SPEC_COMPLIANT_PROOFS
143  #   define  USE_SPEC_COMPLIANT_PROOFS   YES     // Default: Either YES or NO
144  #endif
```

Comment this out to allow compile to continue even though the chosen proof values do not match the compliant values. This is written so that someone would have to proactively ignore errors.

```
145  #if !(defined SKIP_PROOF_ERRORS)                                                          \
146      || ((SKIP_PROOF_ERRORS != NO) && (SKIP_PROOF_ERRORS != YES))
147  #   undef   SKIP_PROOF_ERRORS
148  #   define  SKIP_PROOF_ERRORS           NO      // Default: Either YES or NO
149  #endif
```

This define is used to eliminate the use of bit-fields. It can be enabled for big- or little-endian machines. For big-endian architectures that numbers bits in registers from left to right (MSb0) this must be enabled. Little-endian machines number from right to left with the least significant bit having assigned a bit number of 0. These are LSb0 machines (they are also little-endian so they are also least-significant byte 0 (LSB0) machines. Big-endian (MSB0) machines may number in either direction (MSb0 or LSb0). For an MSB0+MSb0 machine this value is required to be *NO*

```
150  #if !(defined USE_BIT_FIELD_STRUCTURES)                                                   \
151      || ((USE_BIT_FIELD_STRUCTURES != NO) && (USE_BIT_FIELD_STRUCTURES != YES))
152  #   undef   USE_BIT_FIELD_STRUCTURES
153  #   define  USE_BIT_FIELD_STRUCTURES    DEBUG       // Default: Either YES or NO
154  #endif
```

This define is used to control the debug for the CertifyX509() command.

```
155  #if !(defined CERTIFYX509_DEBUG)                                                          \
156      || ((CERTIFYX509_DEBUG != NO) && (CERTIFYX509_DEBUG != YES))
157  #   undef   CERTIFYX509_DEBUG
158  #   define  CERTIFYX509_DEBUG YES               // Default: Either YES or NO
159  #endif
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

This define is used to enable the new table-driven marshaling code.

```
160  #if !(defined TABLE_DRIVEN_MARSHAL)                                              \
161      || ((TABLE_DRIVEN_MARSHAL != NO) && (TABLE_DRIVEN_MARSHAL != YES))
162  #   undef   TABLE_DRIVEN_MARSHAL
163  #   define  TABLE_DRIVEN_MARSHAL YES    // Default: Either YES or NO
164  #endif
```

Change these definitions to turn all algorithms or commands ON or OFF. That is, to turn all algorithms on, set ALG_NO to YES. This is mostly useful as a debug feature.

```
165  #define      ALG_YES      YES
166  #define      ALG_NO       NO
167  #define      CC_YES       YES
168  #define      CC_NO        NO
169  #endif // _TPM_BUILD_SWITCHES_H_
```

 Copyright © TCG 2006-2020

5.18 TpmError.h

```
1  #ifndef _TPM_ERROR_H
2  #define _TPM_ERROR_H
3  #define     FATAL_ERROR_ALLOCATION                (1)
4  #define     FATAL_ERROR_DIVIDE_ZERO               (2)
5  #define     FATAL_ERROR_INTERNAL                  (3)
6  #define     FATAL_ERROR_PARAMETER                 (4)
7  #define     FATAL_ERROR_ENTROPY                   (5)
8  #define     FATAL_ERROR_SELF_TEST                 (6)
9  #define     FATAL_ERROR_CRYPTO                    (7)
10 #define     FATAL_ERROR_NV_UNRECOVERABLE          (8)
11 #define     FATAL_ERROR_REMANUFACTURED            (9) // indicates that the TPM has
12                                                       // been re-manufactured after an
13                                                       // unrecoverable NV error
14 #define     FATAL_ERROR_DRBG                      (10)
15 #define     FATAL_ERROR_MOVE_SIZE                 (11)
16 #define     FATAL_ERROR_COUNTER_OVERFLOW          (12)
17 #define     FATAL_ERROR_SUBTRACT                  (13)
18 #define     FATAL_ERROR_MATHLIBRARY               (14)
19 #define     FATAL_ERROR_FORCED                    (666)
20 #endif // _TPM_ERROR_H
```


 Copyright © TCG 2006-2020

5.19 TpmTypes.h

```
1   #ifndef _TPM_TYPES_H_
2   #define _TPM_TYPES_H_
```

Table 1:2 - Definition of TPM_ALG_ID Constants

```
3   typedef UINT16                          TPM_ALG_ID;
4   #define TYPE_OF_TPM_ALG_ID              UINT16
5   #define     ALG_ERROR_VALUE             0x0000
6   #define TPM_ALG_ERROR                   (TPM_ALG_ID)(ALG_ERROR_VALUE)
7   #define     ALG_RSA_VALUE               0x0001
8   #define TPM_ALG_RSA                     (TPM_ALG_ID)(ALG_RSA_VALUE)
9   #define     ALG_TDES_VALUE              0x0003
10  #define TPM_ALG_TDES                    (TPM_ALG_ID)(ALG_TDES_VALUE)
11  #define     ALG_SHA_VALUE               0x0004
12  #define TPM_ALG_SHA                     (TPM_ALG_ID)(ALG_SHA_VALUE)
13  #define     ALG_SHA1_VALUE              0x0004
14  #define TPM_ALG_SHA1                    (TPM_ALG_ID)(ALG_SHA1_VALUE)
15  #define     ALG_HMAC_VALUE              0x0005
16  #define TPM_ALG_HMAC                    (TPM_ALG_ID)(ALG_HMAC_VALUE)
17  #define     ALG_AES_VALUE               0x0006
18  #define TPM_ALG_AES                     (TPM_ALG_ID)(ALG_AES_VALUE)
19  #define     ALG_MGF1_VALUE              0x0007
20  #define TPM_ALG_MGF1                    (TPM_ALG_ID)(ALG_MGF1_VALUE)
21  #define     ALG_KEYEDHASH_VALUE         0x0008
22  #define TPM_ALG_KEYEDHASH               (TPM_ALG_ID)(ALG_KEYEDHASH_VALUE)
23  #define     ALG_XOR_VALUE               0x000A
24  #define TPM_ALG_XOR                     (TPM_ALG_ID)(ALG_XOR_VALUE)
25  #define     ALG_SHA256_VALUE            0x000B
26  #define TPM_ALG_SHA256                  (TPM_ALG_ID)(ALG_SHA256_VALUE)
27  #define     ALG_SHA384_VALUE            0x000C
28  #define TPM_ALG_SHA384                  (TPM_ALG_ID)(ALG_SHA384_VALUE)
29  #define     ALG_SHA512_VALUE            0x000D
30  #define TPM_ALG_SHA512                  (TPM_ALG_ID)(ALG_SHA512_VALUE)
31  #define     ALG_NULL_VALUE              0x0010
32  #define TPM_ALG_NULL                    (TPM_ALG_ID)(ALG_NULL_VALUE)
33  #define     ALG_SM3_256_VALUE           0x0012
34  #define TPM_ALG_SM3_256                 (TPM_ALG_ID)(ALG_SM3_256_VALUE)
35  #define     ALG_SM4_VALUE               0x0013
36  #define TPM_ALG_SM4                     (TPM_ALG_ID)(ALG_SM4_VALUE)
37  #define     ALG_RSASSA_VALUE            0x0014
38  #define TPM_ALG_RSASSA                  (TPM_ALG_ID)(ALG_RSASSA_VALUE)
39  #define     ALG_RSAES_VALUE             0x0015
40  #define TPM_ALG_RSAES                   (TPM_ALG_ID)(ALG_RSAES_VALUE)
41  #define     ALG_RSAPSS_VALUE            0x0016
42  #define TPM_ALG_RSAPSS                  (TPM_ALG_ID)(ALG_RSAPSS_VALUE)
43  #define     ALG_OAEP_VALUE              0x0017
44  #define TPM_ALG_OAEP                    (TPM_ALG_ID)(ALG_OAEP_VALUE)
45  #define     ALG_ECDSA_VALUE             0x0018
46  #define TPM_ALG_ECDSA                   (TPM_ALG_ID)(ALG_ECDSA_VALUE)
47  #define     ALG_ECDH_VALUE              0x0019
48  #define TPM_ALG_ECDH                    (TPM_ALG_ID)(ALG_ECDH_VALUE)
49  #define     ALG_ECDAA_VALUE             0x001A
50  #define TPM_ALG_ECDAA                   (TPM_ALG_ID)(ALG_ECDAA_VALUE)
51  #define     ALG_SM2_VALUE               0x001B
52  #define TPM_ALG_SM2                     (TPM_ALG_ID)(ALG_SM2_VALUE)
53  #define     ALG_ECSCHNORR_VALUE         0x001C
54  #define TPM_ALG_ECSCHNORR               (TPM_ALG_ID)(ALG_ECSCHNORR_VALUE)
55  #define     ALG_ECMQV_VALUE             0x001D
56  #define TPM_ALG_ECMQV                   (TPM_ALG_ID)(ALG_ECMQV_VALUE)
57  #define     ALG_KDF1_SP800_56A_VALUE    0x0020
58  #define TPM_ALG_KDF1_SP800_56A          (TPM_ALG_ID)(ALG_KDF1_SP800_56A_VALUE)
59  #define     ALG_KDF2_VALUE              0x0021
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
60   #define TPM_ALG_KDF2                        (TPM_ALG_ID)(ALG_KDF2_VALUE)
61   #define     ALG_KDF1_SP800_108_VALUE        0x0022
62   #define TPM_ALG_KDF1_SP800_108              (TPM_ALG_ID)(ALG_KDF1_SP800_108_VALUE)
63   #define     ALG_ECC_VALUE                   0x0023
64   #define TPM_ALG_ECC                         (TPM_ALG_ID)(ALG_ECC_VALUE)
65   #define     ALG_SYMCIPHER_VALUE             0x0025
66   #define TPM_ALG_SYMCIPHER                   (TPM_ALG_ID)(ALG_SYMCIPHER_VALUE)
67   #define     ALG_CAMELLIA_VALUE              0x0026
68   #define TPM_ALG_CAMELLIA                    (TPM_ALG_ID)(ALG_CAMELLIA_VALUE)
69   #define     ALG_SHA3_256_VALUE              0x0027
70   #define TPM_ALG_SHA3_256                    (TPM_ALG_ID)(ALG_SHA3_256_VALUE)
71   #define     ALG_SHA3_384_VALUE              0x0028
72   #define TPM_ALG_SHA3_384                    (TPM_ALG_ID)(ALG_SHA3_384_VALUE)
73   #define     ALG_SHA3_512_VALUE              0x0029
74   #define TPM_ALG_SHA3_512                    (TPM_ALG_ID)(ALG_SHA3_512_VALUE)
75   #define     ALG_CMAC_VALUE                  0x003F
76   #define TPM_ALG_CMAC                        (TPM_ALG_ID)(ALG_CMAC_VALUE)
77   #define     ALG_CTR_VALUE                   0x0040
78   #define TPM_ALG_CTR                         (TPM_ALG_ID)(ALG_CTR_VALUE)
79   #define     ALG_OFB_VALUE                   0x0041
80   #define TPM_ALG_OFB                         (TPM_ALG_ID)(ALG_OFB_VALUE)
81   #define     ALG_CBC_VALUE                   0x0042
82   #define TPM_ALG_CBC                         (TPM_ALG_ID)(ALG_CBC_VALUE)
83   #define     ALG_CFB_VALUE                   0x0043
84   #define TPM_ALG_CFB                         (TPM_ALG_ID)(ALG_CFB_VALUE)
85   #define     ALG_ECB_VALUE                   0x0044
86   #define TPM_ALG_ECB                         (TPM_ALG_ID)(ALG_ECB_VALUE)
```

Values derived from Table 1:2

```
87   #define     ALG_FIRST_VALUE                 0x0001
88   #define TPM_ALG_FIRST                       (TPM_ALG_ID)(ALG_FIRST_VALUE)
89   #define     ALG_LAST_VALUE                  0x0044
90   #define TPM_ALG_LAST                        (TPM_ALG_ID)(ALG_LAST_VALUE)
```

Table 1:4 - Definition of TPM_ECC_CURVE Constants

```
91   typedef UINT16                  TPM_ECC_CURVE;
92   #define TYPE_OF_TPM_ECC_CURVE       UINT16
93   #define TPM_ECC_NONE            (TPM_ECC_CURVE)(0x0000)
94   #define TPM_ECC_NIST_P192       (TPM_ECC_CURVE)(0x0001)
95   #define TPM_ECC_NIST_P224       (TPM_ECC_CURVE)(0x0002)
96   #define TPM_ECC_NIST_P256       (TPM_ECC_CURVE)(0x0003)
97   #define TPM_ECC_NIST_P384       (TPM_ECC_CURVE)(0x0004)
98   #define TPM_ECC_NIST_P521       (TPM_ECC_CURVE)(0x0005)
99   #define TPM_ECC_BN_P256         (TPM_ECC_CURVE)(0x0010)
100  #define TPM_ECC_BN_P638         (TPM_ECC_CURVE)(0x0011)
101  #define TPM_ECC_SM2_P256        (TPM_ECC_CURVE)(0x0020)
```

Table 2:12 - Definition of TPM_CC Constants

```
102  typedef UINT32                              TPM_CC;
103  #define TYPE_OF_TPM_CC                      UINT32
104  #define TPM_CC_NV_UndefineSpaceSpecial      (TPM_CC)(0x0000011F)
105  #define TPM_CC_EvictControl                 (TPM_CC)(0x00000120)
106  #define TPM_CC_HierarchyControl             (TPM_CC)(0x00000121)
107  #define TPM_CC_NV_UndefineSpace             (TPM_CC)(0x00000122)
108  #define TPM_CC_ChangeEPS                    (TPM_CC)(0x00000124)
109  #define TPM_CC_ChangePPS                    (TPM_CC)(0x00000125)
110  #define TPM_CC_Clear                        (TPM_CC)(0x00000126)
111  #define TPM_CC_ClearControl                 (TPM_CC)(0x00000127)
112  #define TPM_CC_ClockSet                     (TPM_CC)(0x00000128)
113  #define TPM_CC_HierarchyChangeAuth          (TPM_CC)(0x00000129)
114  #define TPM_CC_NV_DefineSpace               (TPM_CC)(0x0000012A)
```

 Copyright © TCG 2006-2020

```
115 #define TPM_CC_PCR_Allocate                 (TPM_CC)(0x0000012B)
116 #define TPM_CC_PCR_SetAuthPolicy            (TPM_CC)(0x0000012C)
117 #define TPM_CC_PP_Commands                  (TPM_CC)(0x0000012D)
118 #define TPM_CC_SetPrimaryPolicy             (TPM_CC)(0x0000012E)
119 #define TPM_CC_FieldUpgradeStart            (TPM_CC)(0x0000012F)
120 #define TPM_CC_ClockRateAdjust              (TPM_CC)(0x00000130)
121 #define TPM_CC_CreatePrimary                (TPM_CC)(0x00000131)
122 #define TPM_CC_NV_GlobalWriteLock           (TPM_CC)(0x00000132)
123 #define TPM_CC_GetCommandAuditDigest        (TPM_CC)(0x00000133)
124 #define TPM_CC_NV_Increment                 (TPM_CC)(0x00000134)
125 #define TPM_CC_NV_SetBits                   (TPM_CC)(0x00000135)
126 #define TPM_CC_NV_Extend                    (TPM_CC)(0x00000136)
127 #define TPM_CC_NV_Write                     (TPM_CC)(0x00000137)
128 #define TPM_CC_NV_WriteLock                 (TPM_CC)(0x00000138)
129 #define TPM_CC_DictionaryAttackLockReset    (TPM_CC)(0x00000139)
130 #define TPM_CC_DictionaryAttackParameters   (TPM_CC)(0x0000013A)
131 #define TPM_CC_NV_ChangeAuth                (TPM_CC)(0x0000013B)
132 #define TPM_CC_PCR_Event                    (TPM_CC)(0x0000013C)
133 #define TPM_CC_PCR_Reset                    (TPM_CC)(0x0000013D)
134 #define TPM_CC_SequenceComplete             (TPM_CC)(0x0000013E)
135 #define TPM_CC_SetAlgorithmSet              (TPM_CC)(0x0000013F)
136 #define TPM_CC_SetCommandCodeAuditStatus    (TPM_CC)(0x00000140)
137 #define TPM_CC_FieldUpgradeData             (TPM_CC)(0x00000141)
138 #define TPM_CC_IncrementalSelfTest          (TPM_CC)(0x00000142)
139 #define TPM_CC_SelfTest                     (TPM_CC)(0x00000143)
140 #define TPM_CC_Startup                      (TPM_CC)(0x00000144)
141 #define TPM_CC_Shutdown                     (TPM_CC)(0x00000145)
142 #define TPM_CC_StirRandom                   (TPM_CC)(0x00000146)
143 #define TPM_CC_ActivateCredential           (TPM_CC)(0x00000147)
144 #define TPM_CC_Certify                      (TPM_CC)(0x00000148)
145 #define TPM_CC_PolicyNV                     (TPM_CC)(0x00000149)
146 #define TPM_CC_CertifyCreation              (TPM_CC)(0x0000014A)
147 #define TPM_CC_Duplicate                    (TPM_CC)(0x0000014B)
148 #define TPM_CC_GetTime                      (TPM_CC)(0x0000014C)
149 #define TPM_CC_GetSessionAuditDigest        (TPM_CC)(0x0000014D)
150 #define TPM_CC_NV_Read                      (TPM_CC)(0x0000014E)
151 #define TPM_CC_NV_ReadLock                  (TPM_CC)(0x0000014F)
152 #define TPM_CC_ObjectChangeAuth             (TPM_CC)(0x00000150)
153 #define TPM_CC_PolicySecret                 (TPM_CC)(0x00000151)
154 #define TPM_CC_Rewrap                       (TPM_CC)(0x00000152)
155 #define TPM_CC_Create                       (TPM_CC)(0x00000153)
156 #define TPM_CC_ECDH_ZGen                    (TPM_CC)(0x00000154)
157 #define TPM_CC_HMAC                         (TPM_CC)(0x00000155)
158 #define TPM_CC_MAC                          (TPM_CC)(0x00000155)
159 #define TPM_CC_Import                       (TPM_CC)(0x00000156)
160 #define TPM_CC_Load                         (TPM_CC)(0x00000157)
161 #define TPM_CC_Quote                        (TPM_CC)(0x00000158)
162 #define TPM_CC_RSA_Decrypt                  (TPM_CC)(0x00000159)
163 #define TPM_CC_HMAC_Start                   (TPM_CC)(0x0000015B)
164 #define TPM_CC_MAC_Start                    (TPM_CC)(0x0000015B)
165 #define TPM_CC_SequenceUpdate               (TPM_CC)(0x0000015C)
166 #define TPM_CC_Sign                         (TPM_CC)(0x0000015D)
167 #define TPM_CC_Unseal                       (TPM_CC)(0x0000015E)
168 #define TPM_CC_PolicySigned                 (TPM_CC)(0x00000160)
169 #define TPM_CC_ContextLoad                  (TPM_CC)(0x00000161)
170 #define TPM_CC_ContextSave                  (TPM_CC)(0x00000162)
171 #define TPM_CC_ECDH_KeyGen                  (TPM_CC)(0x00000163)
172 #define TPM_CC_EncryptDecrypt               (TPM_CC)(0x00000164)
173 #define TPM_CC_FlushContext                 (TPM_CC)(0x00000165)
174 #define TPM_CC_LoadExternal                 (TPM_CC)(0x00000167)
175 #define TPM_CC_MakeCredential               (TPM_CC)(0x00000168)
176 #define TPM_CC_NV_ReadPublic                (TPM_CC)(0x00000169)
177 #define TPM_CC_PolicyAuthorize              (TPM_CC)(0x0000016A)
178 #define TPM_CC_PolicyAuthValue              (TPM_CC)(0x0000016B)
179 #define TPM_CC_PolicyCommandCode            (TPM_CC)(0x0000016C)
180 #define TPM_CC_PolicyCounterTimer           (TPM_CC)(0x0000016D)
```

 Copyright © TCG 2006-2020

```
181   #define TPM_CC_PolicyCpHash                   (TPM_CC)(0x0000016E)
182   #define TPM_CC_PolicyLocality                 (TPM_CC)(0x0000016F)
183   #define TPM_CC_PolicyNameHash                 (TPM_CC)(0x00000170)
184   #define TPM_CC_PolicyOR                       (TPM_CC)(0x00000171)
185   #define TPM_CC_PolicyTicket                   (TPM_CC)(0x00000172)
186   #define TPM_CC_ReadPublic                     (TPM_CC)(0x00000173)
187   #define TPM_CC_RSA_Encrypt                    (TPM_CC)(0x00000174)
188   #define TPM_CC_StartAuthSession               (TPM_CC)(0x00000176)
189   #define TPM_CC_VerifySignature                (TPM_CC)(0x00000177)
190   #define TPM_CC_ECC_Parameters                 (TPM_CC)(0x00000178)
191   #define TPM_CC_FirmwareRead                   (TPM_CC)(0x00000179)
192   #define TPM_CC_GetCapability                  (TPM_CC)(0x0000017A)
193   #define TPM_CC_GetRandom                      (TPM_CC)(0x0000017B)
194   #define TPM_CC_GetTestResult                  (TPM_CC)(0x0000017C)
195   #define TPM_CC_Hash                           (TPM_CC)(0x0000017D)
196   #define TPM_CC_PCR_Read                       (TPM_CC)(0x0000017E)
197   #define TPM_CC_PolicyPCR                      (TPM_CC)(0x0000017F)
198   #define TPM_CC_PolicyRestart                  (TPM_CC)(0x00000180)
199   #define TPM_CC_ReadClock                      (TPM_CC)(0x00000181)
200   #define TPM_CC_PCR_Extend                     (TPM_CC)(0x00000182)
201   #define TPM_CC_PCR_SetAuthValue               (TPM_CC)(0x00000183)
202   #define TPM_CC_NV_Certify                     (TPM_CC)(0x00000184)
203   #define TPM_CC_EventSequenceComplete          (TPM_CC)(0x00000185)
204   #define TPM_CC_HashSequenceStart              (TPM_CC)(0x00000186)
205   #define TPM_CC_PolicyPhysicalPresence         (TPM_CC)(0x00000187)
206   #define TPM_CC_PolicyDuplicationSelect        (TPM_CC)(0x00000188)
207   #define TPM_CC_PolicyGetDigest                (TPM_CC)(0x00000189)
208   #define TPM_CC_TestParms                      (TPM_CC)(0x0000018A)
209   #define TPM_CC_Commit                         (TPM_CC)(0x0000018B)
210   #define TPM_CC_PolicyPassword                 (TPM_CC)(0x0000018C)
211   #define TPM_CC_ZGen_2Phase                    (TPM_CC)(0x0000018D)
212   #define TPM_CC_EC_Ephemeral                   (TPM_CC)(0x0000018E)
213   #define TPM_CC_PolicyNvWritten                (TPM_CC)(0x0000018F)
214   #define TPM_CC_PolicyTemplate                 (TPM_CC)(0x00000190)
215   #define TPM_CC_CreateLoaded                   (TPM_CC)(0x00000191)
216   #define TPM_CC_PolicyAuthorizeNV              (TPM_CC)(0x00000192)
217   #define TPM_CC_EncryptDecrypt2                (TPM_CC)(0x00000193)
218   #define TPM_CC_AC_GetCapability               (TPM_CC)(0x00000194)
219   #define TPM_CC_AC_Send                        (TPM_CC)(0x00000195)
220   #define TPM_CC_Policy_AC_SendSelect           (TPM_CC)(0x00000196)
221   #define TPM_CC_CertifyX509                    (TPM_CC)(0x00000197)
222   #define TPM_CC_ACT_SetTimeout                 (TPM_CC)(0x00000198)
223   #define CC_VEND                               0x20000000
224   #define TPM_CC_Vendor_TCG_Test                (TPM_CC)(0x20000000)
```

Table 2:5 - Definition of Types for Documentation Clarity

```
225   typedef UINT32              TPM_ALGORITHM_ID;
226   #define TYPE_OF_TPM_ALGORITHM_ID    UINT32
227   typedef UINT32              TPM_MODIFIER_INDICATOR;
228   #define TYPE_OF_TPM_MODIFIER_INDICATOR  UINT32
229   typedef UINT32              TPM_AUTHORIZATION_SIZE;
230   #define TYPE_OF_TPM_AUTHORIZATION_SIZE  UINT32
231   typedef UINT32              TPM_PARAMETER_SIZE;
232   #define TYPE_OF_TPM_PARAMETER_SIZE  UINT32
233   typedef UINT16              TPM_KEY_SIZE;
234   #define TYPE_OF_TPM_KEY_SIZE    UINT16
235   typedef UINT16              TPM_KEY_BITS;
236   #define TYPE_OF_TPM_KEY_BITS    UINT16
```

Table 2:6 - Definition of TPM_SPEC Constants

```
237   typedef UINT32                  TPM_SPEC;
238   #define TYPE_OF_TPM_SPEC        UINT32
239   #define SPEC_FAMILY             0x322E3000
```

```
240     #define TPM_SPEC_FAMILY            (TPM_SPEC)(SPEC_FAMILY)
241     #define SPEC_LEVEL                 00
242     #define TPM_SPEC_LEVEL             (TPM_SPEC)(SPEC_LEVEL)
243     #define SPEC_VERSION               159
244     #define TPM_SPEC_VERSION           (TPM_SPEC)(SPEC_VERSION)
245     #define SPEC_YEAR                  2019
246     #define TPM_SPEC_YEAR              (TPM_SPEC)(SPEC_YEAR)
247     #define SPEC_DAY_OF_YEAR           312
248     #define TPM_SPEC_DAY_OF_YEAR       (TPM_SPEC)(SPEC_DAY_OF_YEAR)
```

Table 2:7 - Definition of TPM_GENERATED Constants

```
249     typedef UINT32                     TPM_GENERATED;
250     #define TYPE_OF_TPM_GENERATED      UINT32
251     #define TPM_GENERATED_VALUE        (TPM_GENERATED)(0xFF544347)
```

Table 2:16 - Definition of TPM_RC Constants

```
252     typedef UINT32                         TPM_RC;
253     #define TYPE_OF_TPM_RC                 UINT32
254     #define TPM_RC_SUCCESS                 (TPM_RC)(0x000)
255     #define TPM_RC_BAD_TAG                 (TPM_RC)(0x01E)
256     #define RC_VER1                        (TPM_RC)(0x100)
257     #define TPM_RC_INITIALIZE              (TPM_RC)(RC_VER1+0x000)
258     #define TPM_RC_FAILURE                 (TPM_RC)(RC_VER1+0x001)
259     #define TPM_RC_SEQUENCE                (TPM_RC)(RC_VER1+0x003)
260     #define TPM_RC_PRIVATE                 (TPM_RC)(RC_VER1+0x00B)
261     #define TPM_RC_HMAC                    (TPM_RC)(RC_VER1+0x019)
262     #define TPM_RC_DISABLED                (TPM_RC)(RC_VER1+0x020)
263     #define TPM_RC_EXCLUSIVE               (TPM_RC)(RC_VER1+0x021)
264     #define TPM_RC_AUTH_TYPE               (TPM_RC)(RC_VER1+0x024)
265     #define TPM_RC_AUTH_MISSING            (TPM_RC)(RC_VER1+0x025)
266     #define TPM_RC_POLICY                  (TPM_RC)(RC_VER1+0x026)
267     #define TPM_RC_PCR                     (TPM_RC)(RC_VER1+0x027)
268     #define TPM_RC_PCR_CHANGED             (TPM_RC)(RC_VER1+0x028)
269     #define TPM_RC_UPGRADE                 (TPM_RC)(RC_VER1+0x02D)
270     #define TPM_RC_TOO_MANY_CONTEXTS       (TPM_RC)(RC_VER1+0x02E)
271     #define TPM_RC_AUTH_UNAVAILABLE        (TPM_RC)(RC_VER1+0x02F)
272     #define TPM_RC_REBOOT                  (TPM_RC)(RC_VER1+0x030)
273     #define TPM_RC_UNBALANCED              (TPM_RC)(RC_VER1+0x031)
274     #define TPM_RC_COMMAND_SIZE            (TPM_RC)(RC_VER1+0x042)
275     #define TPM_RC_COMMAND_CODE            (TPM_RC)(RC_VER1+0x043)
276     #define TPM_RC_AUTHSIZE                (TPM_RC)(RC_VER1+0x044)
277     #define TPM_RC_AUTH_CONTEXT            (TPM_RC)(RC_VER1+0x045)
278     #define TPM_RC_NV_RANGE                (TPM_RC)(RC_VER1+0x046)
279     #define TPM_RC_NV_SIZE                 (TPM_RC)(RC_VER1+0x047)
280     #define TPM_RC_NV_LOCKED               (TPM_RC)(RC_VER1+0x048)
281     #define TPM_RC_NV_AUTHORIZATION        (TPM_RC)(RC_VER1+0x049)
282     #define TPM_RC_NV_UNINITIALIZED        (TPM_RC)(RC_VER1+0x04A)
283     #define TPM_RC_NV_SPACE                (TPM_RC)(RC_VER1+0x04B)
284     #define TPM_RC_NV_DEFINED              (TPM_RC)(RC_VER1+0x04C)
285     #define TPM_RC_BAD_CONTEXT             (TPM_RC)(RC_VER1+0x050)
286     #define TPM_RC_CPHASH                  (TPM_RC)(RC_VER1+0x051)
287     #define TPM_RC_PARENT                  (TPM_RC)(RC_VER1+0x052)
288     #define TPM_RC_NEEDS_TEST              (TPM_RC)(RC_VER1+0x053)
289     #define TPM_RC_NO_RESULT               (TPM_RC)(RC_VER1+0x054)
290     #define TPM_RC_SENSITIVE               (TPM_RC)(RC_VER1+0x055)
291     #define RC_MAX_FM0                     (TPM_RC)(RC_VER1+0x07F)
292     #define RC_FMT1                        (TPM_RC)(0x080)
293     #define TPM_RC_ASYMMETRIC              (TPM_RC)(RC_FMT1+0x001)
294     #define TPM_RCS_ASYMMETRIC             (TPM_RC)(RC_FMT1+0x001)
295     #define TPM_RC_ATTRIBUTES              (TPM_RC)(RC_FMT1+0x002)
296     #define TPM_RCS_ATTRIBUTES             (TPM_RC)(RC_FMT1+0x002)
297     #define TPM_RC_HASH                    (TPM_RC)(RC_FMT1+0x003)
298     #define TPM_RCS_HASH                   (TPM_RC)(RC_FMT1+0x003)
```

 Copyright © TCG 2006-2020

```
299 #define TPM_RC_VALUE                (TPM_RC)(RC_FMT1+0x004)
300 #define TPM_RCS_VALUE               (TPM_RC)(RC_FMT1+0x004)
301 #define TPM_RC_HIERARCHY            (TPM_RC)(RC_FMT1+0x005)
302 #define TPM_RCS_HIERARCHY           (TPM_RC)(RC_FMT1+0x005)
303 #define TPM_RC_KEY_SIZE             (TPM_RC)(RC_FMT1+0x007)
304 #define TPM_RCS_KEY_SIZE            (TPM_RC)(RC_FMT1+0x007)
305 #define TPM_RC_MGF                  (TPM_RC)(RC_FMT1+0x008)
306 #define TPM_RCS_MGF                 (TPM_RC)(RC_FMT1+0x008)
307 #define TPM_RC_MODE                 (TPM_RC)(RC_FMT1+0x009)
308 #define TPM_RCS_MODE                (TPM_RC)(RC_FMT1+0x009)
309 #define TPM_RC_TYPE                 (TPM_RC)(RC_FMT1+0x00A)
310 #define TPM_RCS_TYPE                (TPM_RC)(RC_FMT1+0x00A)
311 #define TPM_RC_HANDLE               (TPM_RC)(RC_FMT1+0x00B)
312 #define TPM_RCS_HANDLE              (TPM_RC)(RC_FMT1+0x00B)
313 #define TPM_RC_KDF                  (TPM_RC)(RC_FMT1+0x00C)
314 #define TPM_RCS_KDF                 (TPM_RC)(RC_FMT1+0x00C)
315 #define TPM_RC_RANGE                (TPM_RC)(RC_FMT1+0x00D)
316 #define TPM_RCS_RANGE               (TPM_RC)(RC_FMT1+0x00D)
317 #define TPM_RC_AUTH_FAIL            (TPM_RC)(RC_FMT1+0x00E)
318 #define TPM_RCS_AUTH_FAIL           (TPM_RC)(RC_FMT1+0x00E)
319 #define TPM_RC_NONCE                (TPM_RC)(RC_FMT1+0x00F)
320 #define TPM_RCS_NONCE               (TPM_RC)(RC_FMT1+0x00F)
321 #define TPM_RC_PP                   (TPM_RC)(RC_FMT1+0x010)
322 #define TPM_RCS_PP                  (TPM_RC)(RC_FMT1+0x010)
323 #define TPM_RC_SCHEME               (TPM_RC)(RC_FMT1+0x012)
324 #define TPM_RCS_SCHEME              (TPM_RC)(RC_FMT1+0x012)
325 #define TPM_RC_SIZE                 (TPM_RC)(RC_FMT1+0x015)
326 #define TPM_RCS_SIZE                (TPM_RC)(RC_FMT1+0x015)
327 #define TPM_RC_SYMMETRIC            (TPM_RC)(RC_FMT1+0x016)
328 #define TPM_RCS_SYMMETRIC           (TPM_RC)(RC_FMT1+0x016)
329 #define TPM_RC_TAG                  (TPM_RC)(RC_FMT1+0x017)
330 #define TPM_RCS_TAG                 (TPM_RC)(RC_FMT1+0x017)
331 #define TPM_RC_SELECTOR             (TPM_RC)(RC_FMT1+0x018)
332 #define TPM_RCS_SELECTOR            (TPM_RC)(RC_FMT1+0x018)
333 #define TPM_RC_INSUFFICIENT         (TPM_RC)(RC_FMT1+0x01A)
334 #define TPM_RCS_INSUFFICIENT        (TPM_RC)(RC_FMT1+0x01A)
335 #define TPM_RC_SIGNATURE            (TPM_RC)(RC_FMT1+0x01B)
336 #define TPM_RCS_SIGNATURE           (TPM_RC)(RC_FMT1+0x01B)
337 #define TPM_RC_KEY                  (TPM_RC)(RC_FMT1+0x01C)
338 #define TPM_RCS_KEY                 (TPM_RC)(RC_FMT1+0x01C)
339 #define TPM_RC_POLICY_FAIL          (TPM_RC)(RC_FMT1+0x01D)
340 #define TPM_RCS_POLICY_FAIL         (TPM_RC)(RC_FMT1+0x01D)
341 #define TPM_RC_INTEGRITY            (TPM_RC)(RC_FMT1+0x01F)
342 #define TPM_RCS_INTEGRITY           (TPM_RC)(RC_FMT1+0x01F)
343 #define TPM_RC_TICKET               (TPM_RC)(RC_FMT1+0x020)
344 #define TPM_RCS_TICKET              (TPM_RC)(RC_FMT1+0x020)
345 #define TPM_RC_RESERVED_BITS        (TPM_RC)(RC_FMT1+0x021)
346 #define TPM_RCS_RESERVED_BITS       (TPM_RC)(RC_FMT1+0x021)
347 #define TPM_RC_BAD_AUTH             (TPM_RC)(RC_FMT1+0x022)
348 #define TPM_RCS_BAD_AUTH            (TPM_RC)(RC_FMT1+0x022)
349 #define TPM_RC_EXPIRED              (TPM_RC)(RC_FMT1+0x023)
350 #define TPM_RCS_EXPIRED             (TPM_RC)(RC_FMT1+0x023)
351 #define TPM_RC_POLICY_CC            (TPM_RC)(RC_FMT1+0x024)
352 #define TPM_RCS_POLICY_CC           (TPM_RC)(RC_FMT1+0x024)
353 #define TPM_RC_BINDING              (TPM_RC)(RC_FMT1+0x025)
354 #define TPM_RCS_BINDING             (TPM_RC)(RC_FMT1+0x025)
355 #define TPM_RC_CURVE                (TPM_RC)(RC_FMT1+0x026)
356 #define TPM_RCS_CURVE               (TPM_RC)(RC_FMT1+0x026)
357 #define TPM_RC_ECC_POINT            (TPM_RC)(RC_FMT1+0x027)
358 #define TPM_RCS_ECC_POINT           (TPM_RC)(RC_FMT1+0x027)
359 #define RC_WARN                     (TPM_RC)(0x900)
360 #define TPM_RC_CONTEXT_GAP          (TPM_RC)(RC_WARN+0x001)
361 #define TPM_RC_OBJECT_MEMORY        (TPM_RC)(RC_WARN+0x002)
362 #define TPM_RC_SESSION_MEMORY       (TPM_RC)(RC_WARN+0x003)
363 #define TPM_RC_MEMORY               (TPM_RC)(RC_WARN+0x004)
364 #define TPM_RC_SESSION_HANDLES      (TPM_RC)(RC_WARN+0x005)
```

```
365     #define TPM_RC_OBJECT_HANDLES       (TPM_RC)(RC_WARN+0x006)
366     #define TPM_RC_LOCALITY             (TPM_RC)(RC_WARN+0x007)
367     #define TPM_RC_YIELDED              (TPM_RC)(RC_WARN+0x008)
368     #define TPM_RC_CANCELED             (TPM_RC)(RC_WARN+0x009)
369     #define TPM_RC_TESTING              (TPM_RC)(RC_WARN+0x00A)
370     #define TPM_RC_REFERENCE_H0         (TPM_RC)(RC_WARN+0x010)
371     #define TPM_RC_REFERENCE_H1         (TPM_RC)(RC_WARN+0x011)
372     #define TPM_RC_REFERENCE_H2         (TPM_RC)(RC_WARN+0x012)
373     #define TPM_RC_REFERENCE_H3         (TPM_RC)(RC_WARN+0x013)
374     #define TPM_RC_REFERENCE_H4         (TPM_RC)(RC_WARN+0x014)
375     #define TPM_RC_REFERENCE_H5         (TPM_RC)(RC_WARN+0x015)
376     #define TPM_RC_REFERENCE_H6         (TPM_RC)(RC_WARN+0x016)
377     #define TPM_RC_REFERENCE_S0         (TPM_RC)(RC_WARN+0x018)
378     #define TPM_RC_REFERENCE_S1         (TPM_RC)(RC_WARN+0x019)
379     #define TPM_RC_REFERENCE_S2         (TPM_RC)(RC_WARN+0x01A)
380     #define TPM_RC_REFERENCE_S3         (TPM_RC)(RC_WARN+0x01B)
381     #define TPM_RC_REFERENCE_S4         (TPM_RC)(RC_WARN+0x01C)
382     #define TPM_RC_REFERENCE_S5         (TPM_RC)(RC_WARN+0x01D)
383     #define TPM_RC_REFERENCE_S6         (TPM_RC)(RC_WARN+0x01E)
384     #define TPM_RC_NV_RATE              (TPM_RC)(RC_WARN+0x020)
385     #define TPM_RC_LOCKOUT              (TPM_RC)(RC_WARN+0x021)
386     #define TPM_RC_RETRY                (TPM_RC)(RC_WARN+0x022)
387     #define TPM_RC_NV_UNAVAILABLE       (TPM_RC)(RC_WARN+0x023)
388     #define TPM_RC_NOT_USED             (TPM_RC)(RC_WARN+0x7F)
389     #define TPM_RC_H                    (TPM_RC)(0x000)
390     #define TPM_RC_P                    (TPM_RC)(0x040)
391     #define TPM_RC_S                    (TPM_RC)(0x800)
392     #define TPM_RC_1                    (TPM_RC)(0x100)
393     #define TPM_RC_2                    (TPM_RC)(0x200)
394     #define TPM_RC_3                    (TPM_RC)(0x300)
395     #define TPM_RC_4                    (TPM_RC)(0x400)
396     #define TPM_RC_5                    (TPM_RC)(0x500)
397     #define TPM_RC_6                    (TPM_RC)(0x600)
398     #define TPM_RC_7                    (TPM_RC)(0x700)
399     #define TPM_RC_8                    (TPM_RC)(0x800)
400     #define TPM_RC_9                    (TPM_RC)(0x900)
401     #define TPM_RC_A                    (TPM_RC)(0xA00)
402     #define TPM_RC_B                    (TPM_RC)(0xB00)
403     #define TPM_RC_C                    (TPM_RC)(0xC00)
404     #define TPM_RC_D                    (TPM_RC)(0xD00)
405     #define TPM_RC_E                    (TPM_RC)(0xE00)
406     #define TPM_RC_F                    (TPM_RC)(0xF00)
407     #define TPM_RC_N_MASK               (TPM_RC)(0xF00)
```

Table 2:17 - Definition of TPM_CLOCK_ADJUST Constants

```
408     typedef INT8                        TPM_CLOCK_ADJUST;
409     #define TYPE_OF_TPM_CLOCK_ADJUST    UINT8
410     #define TPM_CLOCK_COARSE_SLOWER     (TPM_CLOCK_ADJUST)(-3)
411     #define TPM_CLOCK_MEDIUM_SLOWER     (TPM_CLOCK_ADJUST)(-2)
412     #define TPM_CLOCK_FINE_SLOWER       (TPM_CLOCK_ADJUST)(-1)
413     #define TPM_CLOCK_NO_CHANGE         (TPM_CLOCK_ADJUST)(0)
414     #define TPM_CLOCK_FINE_FASTER       (TPM_CLOCK_ADJUST)(1)
415     #define TPM_CLOCK_MEDIUM_FASTER     (TPM_CLOCK_ADJUST)(2)
416     #define TPM_CLOCK_COARSE_FASTER     (TPM_CLOCK_ADJUST)(3)
```

Table 2:18 - Definition of TPM_EO Constants

```
417     typedef UINT16              TPM_EO;
418     #define TYPE_OF_TPM_EO      UINT16
419     #define TPM_EO_EQ           (TPM_EO)(0x0000)
420     #define TPM_EO_NEQ          (TPM_EO)(0x0001)
421     #define TPM_EO_SIGNED_GT    (TPM_EO)(0x0002)
422     #define TPM_EO_UNSIGNED_GT  (TPM_EO)(0x0003)
423     #define TPM_EO_SIGNED_LT    (TPM_EO)(0x0004)
```

 Copyright © TCG 2006-2020

```
424     #define TPM_EO_UNSIGNED_LT    (TPM_EO)(0x0005)
425     #define TPM_EO_SIGNED_GE      (TPM_EO)(0x0006)
426     #define TPM_EO_UNSIGNED_GE    (TPM_EO)(0x0007)
427     #define TPM_EO_SIGNED_LE      (TPM_EO)(0x0008)
428     #define TPM_EO_UNSIGNED_LE    (TPM_EO)(0x0009)
429     #define TPM_EO_BITSET         (TPM_EO)(0x000A)
430     #define TPM_EO_BITCLEAR       (TPM_EO)(0x000B)
```

Table 2:19 - Definition of TPM_ST Constants

```
431     typedef UINT16                              TPM_ST;
432     #define TYPE_OF_TPM_ST                      UINT16
433     #define TPM_ST_RSP_COMMAND                  (TPM_ST)(0x00C4)
434     #define TPM_ST_NULL                         (TPM_ST)(0x8000)
435     #define TPM_ST_NO_SESSIONS                  (TPM_ST)(0x8001)
436     #define TPM_ST_SESSIONS                     (TPM_ST)(0x8002)
437     #define TPM_ST_ATTEST_NV                    (TPM_ST)(0x8014)
438     #define TPM_ST_ATTEST_COMMAND_AUDIT         (TPM_ST)(0x8015)
439     #define TPM_ST_ATTEST_SESSION_AUDIT         (TPM_ST)(0x8016)
440     #define TPM_ST_ATTEST_CERTIFY               (TPM_ST)(0x8017)
441     #define TPM_ST_ATTEST_QUOTE                 (TPM_ST)(0x8018)
442     #define TPM_ST_ATTEST_TIME                  (TPM_ST)(0x8019)
443     #define TPM_ST_ATTEST_CREATION              (TPM_ST)(0x801A)
444     #define TPM_ST_ATTEST_NV_DIGEST             (TPM_ST)(0x801C)
445     #define TPM_ST_CREATION                     (TPM_ST)(0x8021)
446     #define TPM_ST_VERIFIED                     (TPM_ST)(0x8022)
447     #define TPM_ST_AUTH_SECRET                  (TPM_ST)(0x8023)
448     #define TPM_ST_HASHCHECK                    (TPM_ST)(0x8024)
449     #define TPM_ST_AUTH_SIGNED                  (TPM_ST)(0x8025)
450     #define TPM_ST_FU_MANIFEST                  (TPM_ST)(0x8029)
```

Table 2:20 - Definition of TPM_SU Constants

```
451     typedef UINT16              TPM_SU;
452     #define TYPE_OF_TPM_SU      UINT16
453     #define TPM_SU_CLEAR        (TPM_SU)(0x0000)
454     #define TPM_SU_STATE        (TPM_SU)(0x0001)
```

Table 2:21 - Definition of TPM_SE Constants

```
455     typedef UINT8               TPM_SE;
456     #define TYPE_OF_TPM_SE      UINT8
457     #define TPM_SE_HMAC         (TPM_SE)(0x00)
458     #define TPM_SE_POLICY       (TPM_SE)(0x01)
459     #define TPM_SE_TRIAL        (TPM_SE)(0x03)
```

Table 2:22 - Definition of TPM_CAP Constants

```
460     typedef UINT32                       TPM_CAP;
461     #define TYPE_OF_TPM_CAP              UINT32
462     #define TPM_CAP_FIRST                (TPM_CAP)(0x00000000)
463     #define TPM_CAP_ALGS                 (TPM_CAP)(0x00000000)
464     #define TPM_CAP_HANDLES              (TPM_CAP)(0x00000001)
465     #define TPM_CAP_COMMANDS             (TPM_CAP)(0x00000002)
466     #define TPM_CAP_PP_COMMANDS          (TPM_CAP)(0x00000003)
467     #define TPM_CAP_AUDIT_COMMANDS       (TPM_CAP)(0x00000004)
468     #define TPM_CAP_PCRS                 (TPM_CAP)(0x00000005)
469     #define TPM_CAP_TPM_PROPERTIES       (TPM_CAP)(0x00000006)
470     #define TPM_CAP_PCR_PROPERTIES       (TPM_CAP)(0x00000007)
471     #define TPM_CAP_ECC_CURVES           (TPM_CAP)(0x00000008)
472     #define TPM_CAP_AUTH_POLICIES        (TPM_CAP)(0x00000009)
473     #define TPM_CAP_ACT                  (TPM_CAP)(0x0000000A)
474     #define TPM_CAP_LAST                 (TPM_CAP)(0x0000000A)
475     #define TPM_CAP_VENDOR_PROPERTY      (TPM_CAP)(0x00000100)
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

Table 2:23 - Definition of TPM_PT Constants

```
476 typedef UINT32                  TPM_PT;
477 #define TYPE_OF_TPM_PT            UINT32
478 #define TPM_PT_NONE               (TPM_PT)(0x00000000)
479 #define PT_GROUP                  (TPM_PT)(0x00000100)
480 #define PT_FIXED                  (TPM_PT)(PT_GROUP*1)
481 #define TPM_PT_FAMILY_INDICATOR   (TPM_PT)(PT_FIXED+0)
482 #define TPM_PT_LEVEL              (TPM_PT)(PT_FIXED+1)
483 #define TPM_PT_REVISION           (TPM_PT)(PT_FIXED+2)
484 #define TPM_PT_DAY_OF_YEAR        (TPM_PT)(PT_FIXED+3)
485 #define TPM_PT_YEAR               (TPM_PT)(PT_FIXED+4)
486 #define TPM_PT_MANUFACTURER       (TPM_PT)(PT_FIXED+5)
487 #define TPM_PT_VENDOR_STRING_1    (TPM_PT)(PT_FIXED+6)
488 #define TPM_PT_VENDOR_STRING_2    (TPM_PT)(PT_FIXED+7)
489 #define TPM_PT_VENDOR_STRING_3    (TPM_PT)(PT_FIXED+8)
490 #define TPM_PT_VENDOR_STRING_4    (TPM_PT)(PT_FIXED+9)
491 #define TPM_PT_VENDOR_TPM_TYPE    (TPM_PT)(PT_FIXED+10)
492 #define TPM_PT_FIRMWARE_VERSION_1 (TPM_PT)(PT_FIXED+11)
493 #define TPM_PT_FIRMWARE_VERSION_2 (TPM_PT)(PT_FIXED+12)
494 #define TPM_PT_INPUT_BUFFER       (TPM_PT)(PT_FIXED+13)
495 #define TPM_PT_HR_TRANSIENT_MIN   (TPM_PT)(PT_FIXED+14)
496 #define TPM_PT_HR_PERSISTENT_MIN  (TPM_PT)(PT_FIXED+15)
497 #define TPM_PT_HR_LOADED_MIN      (TPM_PT)(PT_FIXED+16)
498 #define TPM_PT_ACTIVE_SESSIONS_MAX (TPM_PT)(PT_FIXED+17)
499 #define TPM_PT_PCR_COUNT          (TPM_PT)(PT_FIXED+18)
500 #define TPM_PT_PCR_SELECT_MIN     (TPM_PT)(PT_FIXED+19)
501 #define TPM_PT_CONTEXT_GAP_MAX    (TPM_PT)(PT_FIXED+20)
502 #define TPM_PT_NV_COUNTERS_MAX    (TPM_PT)(PT_FIXED+22)
503 #define TPM_PT_NV_INDEX_MAX       (TPM_PT)(PT_FIXED+23)
504 #define TPM_PT_MEMORY             (TPM_PT)(PT_FIXED+24)
505 #define TPM_PT_CLOCK_UPDATE       (TPM_PT)(PT_FIXED+25)
506 #define TPM_PT_CONTEXT_HASH       (TPM_PT)(PT_FIXED+26)
507 #define TPM_PT_CONTEXT_SYM        (TPM_PT)(PT_FIXED+27)
508 #define TPM_PT_CONTEXT_SYM_SIZE   (TPM_PT)(PT_FIXED+28)
509 #define TPM_PT_ORDERLY_COUNT      (TPM_PT)(PT_FIXED+29)
510 #define TPM_PT_MAX_COMMAND_SIZE   (TPM_PT)(PT_FIXED+30)
511 #define TPM_PT_MAX_RESPONSE_SIZE  (TPM_PT)(PT_FIXED+31)
512 #define TPM_PT_MAX_DIGEST         (TPM_PT)(PT_FIXED+32)
513 #define TPM_PT_MAX_OBJECT_CONTEXT (TPM_PT)(PT_FIXED+33)
514 #define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT)(PT_FIXED+34)
515 #define TPM_PT_PS_FAMILY_INDICATOR (TPM_PT)(PT_FIXED+35)
516 #define TPM_PT_PS_LEVEL           (TPM_PT)(PT_FIXED+36)
517 #define TPM_PT_PS_REVISION        (TPM_PT)(PT_FIXED+37)
518 #define TPM_PT_PS_DAY_OF_YEAR     (TPM_PT)(PT_FIXED+38)
519 #define TPM_PT_PS_YEAR            (TPM_PT)(PT_FIXED+39)
520 #define TPM_PT_SPLIT_MAX          (TPM_PT)(PT_FIXED+40)
521 #define TPM_PT_TOTAL_COMMANDS     (TPM_PT)(PT_FIXED+41)
522 #define TPM_PT_LIBRARY_COMMANDS   (TPM_PT)(PT_FIXED+42)
523 #define TPM_PT_VENDOR_COMMANDS    (TPM_PT)(PT_FIXED+43)
524 #define TPM_PT_NV_BUFFER_MAX      (TPM_PT)(PT_FIXED+44)
525 #define TPM_PT_MODES              (TPM_PT)(PT_FIXED+45)
526 #define TPM_PT_MAX_CAP_BUFFER     (TPM_PT)(PT_FIXED+46)
527 #define PT_VAR                    (TPM_PT)(PT_GROUP*2)
528 #define TPM_PT_PERMANENT          (TPM_PT)(PT_VAR+0)
529 #define TPM_PT_STARTUP_CLEAR      (TPM_PT)(PT_VAR+1)
530 #define TPM_PT_HR_NV_INDEX        (TPM_PT)(PT_VAR+2)
531 #define TPM_PT_HR_LOADED          (TPM_PT)(PT_VAR+3)
532 #define TPM_PT_HR_LOADED_AVAIL    (TPM_PT)(PT_VAR+4)
533 #define TPM_PT_HR_ACTIVE          (TPM_PT)(PT_VAR+5)
534 #define TPM_PT_HR_ACTIVE_AVAIL    (TPM_PT)(PT_VAR+6)
535 #define TPM_PT_HR_TRANSIENT_AVAIL (TPM_PT)(PT_VAR+7)
536 #define TPM_PT_HR_PERSISTENT      (TPM_PT)(PT_VAR+8)
537 #define TPM_PT_HR_PERSISTENT_AVAIL (TPM_PT)(PT_VAR+9)
538 #define TPM_PT_NV_COUNTERS        (TPM_PT)(PT_VAR+10)
539 #define TPM_PT_NV_COUNTERS_AVAIL  (TPM_PT)(PT_VAR+11)
```

 Copyright © TCG 2006-2020

```
540     #define TPM_PT_ALGORITHM_SET          (TPM_PT)(PT_VAR+12)
541     #define TPM_PT_LOADED_CURVES          (TPM_PT)(PT_VAR+13)
542     #define TPM_PT_LOCKOUT_COUNTER        (TPM_PT)(PT_VAR+14)
543     #define TPM_PT_MAX_AUTH_FAIL          (TPM_PT)(PT_VAR+15)
544     #define TPM_PT_LOCKOUT_INTERVAL       (TPM_PT)(PT_VAR+16)
545     #define TPM_PT_LOCKOUT_RECOVERY       (TPM_PT)(PT_VAR+17)
546     #define TPM_PT_NV_WRITE_RECOVERY      (TPM_PT)(PT_VAR+18)
547     #define TPM_PT_AUDIT_COUNTER_0        (TPM_PT)(PT_VAR+19)
548     #define TPM_PT_AUDIT_COUNTER_1        (TPM_PT)(PT_VAR+20)
```

Table 2:24 - Definition of TPM_PT_PCR Constants

```
549     typedef UINT32                        TPM_PT_PCR;
550     #define TYPE_OF_TPM_PT_PCR            UINT32
551     #define TPM_PT_PCR_FIRST              (TPM_PT_PCR)(0x00000000)
552     #define TPM_PT_PCR_SAVE               (TPM_PT_PCR)(0x00000000)
553     #define TPM_PT_PCR_EXTEND_L0          (TPM_PT_PCR)(0x00000001)
554     #define TPM_PT_PCR_RESET_L0           (TPM_PT_PCR)(0x00000002)
555     #define TPM_PT_PCR_EXTEND_L1          (TPM_PT_PCR)(0x00000003)
556     #define TPM_PT_PCR_RESET_L1           (TPM_PT_PCR)(0x00000004)
557     #define TPM_PT_PCR_EXTEND_L2          (TPM_PT_PCR)(0x00000005)
558     #define TPM_PT_PCR_RESET_L2           (TPM_PT_PCR)(0x00000006)
559     #define TPM_PT_PCR_EXTEND_L3          (TPM_PT_PCR)(0x00000007)
560     #define TPM_PT_PCR_RESET_L3           (TPM_PT_PCR)(0x00000008)
561     #define TPM_PT_PCR_EXTEND_L4          (TPM_PT_PCR)(0x00000009)
562     #define TPM_PT_PCR_RESET_L4           (TPM_PT_PCR)(0x0000000A)
563     #define TPM_PT_PCR_NO_INCREMENT       (TPM_PT_PCR)(0x00000011)
564     #define TPM_PT_PCR_DRTM_RESET         (TPM_PT_PCR)(0x00000012)
565     #define TPM_PT_PCR_POLICY             (TPM_PT_PCR)(0x00000013)
566     #define TPM_PT_PCR_AUTH               (TPM_PT_PCR)(0x00000014)
567     #define TPM_PT_PCR_LAST               (TPM_PT_PCR)(0x00000014)
```

Table 2:25 - Definition of TPM_PS Constants

```
568     typedef UINT32                    TPM_PS;
569     #define TYPE_OF_TPM_PS            UINT32
570     #define TPM_PS_MAIN               (TPM_PS)(0x00000000)
571     #define TPM_PS_PC                 (TPM_PS)(0x00000001)
572     #define TPM_PS_PDA                (TPM_PS)(0x00000002)
573     #define TPM_PS_CELL_PHONE         (TPM_PS)(0x00000003)
574     #define TPM_PS_SERVER             (TPM_PS)(0x00000004)
575     #define TPM_PS_PERIPHERAL         (TPM_PS)(0x00000005)
576     #define TPM_PS_TSS                (TPM_PS)(0x00000006)
577     #define TPM_PS_STORAGE            (TPM_PS)(0x00000007)
578     #define TPM_PS_AUTHENTICATION     (TPM_PS)(0x00000008)
579     #define TPM_PS_EMBEDDED           (TPM_PS)(0x00000009)
580     #define TPM_PS_HARDCOPY           (TPM_PS)(0x0000000A)
581     #define TPM_PS_INFRASTRUCTURE     (TPM_PS)(0x0000000B)
582     #define TPM_PS_VIRTUALIZATION     (TPM_PS)(0x0000000C)
583     #define TPM_PS_TNC                (TPM_PS)(0x0000000D)
584     #define TPM_PS_MULTI_TENANT       (TPM_PS)(0x0000000E)
585     #define TPM_PS_TC                 (TPM_PS)(0x0000000F)
```

Table 2:26 - Definition of Types for Handles

```
586     typedef UINT32                TPM_HANDLE;
587     #define TYPE_OF_TPM_HANDLE    UINT32
```

Table 2:27 - Definition of TPM_HT Constants

```
588     typedef UINT8                     TPM_HT;
589     #define TYPE_OF_TPM_HT            UINT8
590     #define TPM_HT_PCR                (TPM_HT)(0x00)
591     #define TPM_HT_NV_INDEX           (TPM_HT)(0x01)
```

 Copyright © TCG 2006-2020

```
592 #define TPM_HT_HMAC_SESSION      (TPM_HT)(0x02)
593 #define TPM_HT_LOADED_SESSION    (TPM_HT)(0x02)
594 #define TPM_HT_POLICY_SESSION    (TPM_HT)(0x03)
595 #define TPM_HT_SAVED_SESSION     (TPM_HT)(0x03)
596 #define TPM_HT_PERMANENT         (TPM_HT)(0x40)
597 #define TPM_HT_TRANSIENT         (TPM_HT)(0x80)
598 #define TPM_HT_PERSISTENT        (TPM_HT)(0x81)
599 #define TPM_HT_AC                (TPM_HT)(0x90)
```

Table 2:28 - Definition of TPM_RH Constants

```
600 typedef TPM_HANDLE           TPM_RH;
601 #define TPM_RH_FIRST         (TPM_RH)(0x40000000)
602 #define TPM_RH_SRK           (TPM_RH)(0x40000000)
603 #define TPM_RH_OWNER         (TPM_RH)(0x40000001)
604 #define TPM_RH_REVOKE        (TPM_RH)(0x40000002)
605 #define TPM_RH_TRANSPORT     (TPM_RH)(0x40000003)
606 #define TPM_RH_OPERATOR      (TPM_RH)(0x40000004)
607 #define TPM_RH_ADMIN         (TPM_RH)(0x40000005)
608 #define TPM_RH_EK            (TPM_RH)(0x40000006)
609 #define TPM_RH_NULL          (TPM_RH)(0x40000007)
610 #define TPM_RH_UNASSIGNED    (TPM_RH)(0x40000008)
611 #define TPM_RS_PW            (TPM_RH)(0x40000009)
612 #define TPM_RH_LOCKOUT       (TPM_RH)(0x4000000A)
613 #define TPM_RH_ENDORSEMENT   (TPM_RH)(0x4000000B)
614 #define TPM_RH_PLATFORM      (TPM_RH)(0x4000000C)
615 #define TPM_RH_PLATFORM_NV   (TPM_RH)(0x4000000D)
616 #define TPM_RH_AUTH_00       (TPM_RH)(0x40000010)
617 #define TPM_RH_AUTH_FF       (TPM_RH)(0x4000010F)
618 #define TPM_RH_ACT_0         (TPM_RH)(0x40000110)
619 #define TPM_RH_ACT_F         (TPM_RH)(0x4000011F)
620 #define TPM_RH_LAST          (TPM_RH)(0x4000011F)
```

Table 2:29 - Definition of TPM_HC Constants

```
621 typedef TPM_HANDLE               TPM_HC;
622 #define HR_HANDLE_MASK           (TPM_HC)(0x00FFFFFF)
623 #define HR_RANGE_MASK            (TPM_HC)(0xFF000000)
624 #define HR_SHIFT                 (TPM_HC)(24)
625 #define HR_PCR                   (TPM_HC)((TPM_HT_PCR<<HR_SHIFT))
626 #define HR_HMAC_SESSION          (TPM_HC)((TPM_HT_HMAC_SESSION<<HR_SHIFT))
627 #define HR_POLICY_SESSION        (TPM_HC)((TPM_HT_POLICY_SESSION<<HR_SHIFT))
628 #define HR_TRANSIENT             (TPM_HC)((TPM_HT_TRANSIENT<<HR_SHIFT))
629 #define HR_PERSISTENT            (TPM_HC)((TPM_HT_PERSISTENT<<HR_SHIFT))
630 #define HR_NV_INDEX              (TPM_HC)((TPM_HT_NV_INDEX<<HR_SHIFT))
631 #define HR_PERMANENT             (TPM_HC)((TPM_HT_PERMANENT<<HR_SHIFT))
632 #define PCR_FIRST                (TPM_HC)((HR_PCR+0))
633 #define PCR_LAST                 (TPM_HC)((PCR_FIRST+IMPLEMENTATION_PCR-1))
634 #define HMAC_SESSION_FIRST       (TPM_HC)((HR_HMAC_SESSION+0))
635 #define HMAC_SESSION_LAST        (TPM_HC)((HMAC_SESSION_FIRST+MAX_ACTIVE_SESSIONS-1))
636 #define LOADED_SESSION_FIRST     (TPM_HC)(HMAC_SESSION_FIRST)
637 #define LOADED_SESSION_LAST      (TPM_HC)(HMAC_SESSION_LAST)
638 #define POLICY_SESSION_FIRST     (TPM_HC)((HR_POLICY_SESSION+0))
639 #define POLICY_SESSION_LAST      \
640             (TPM_HC)((POLICY_SESSION_FIRST+MAX_ACTIVE_SESSIONS-1))
641 #define TRANSIENT_FIRST          (TPM_HC)((HR_TRANSIENT+0))
642 #define ACTIVE_SESSION_FIRST     (TPM_HC)(POLICY_SESSION_FIRST)
643 #define ACTIVE_SESSION_LAST      (TPM_HC)(POLICY_SESSION_LAST)
644 #define TRANSIENT_LAST           (TPM_HC)((TRANSIENT_FIRST+MAX_LOADED_OBJECTS-1))
645 #define PERSISTENT_FIRST         (TPM_HC)((HR_PERSISTENT+0))
646 #define PERSISTENT_LAST          (TPM_HC)((PERSISTENT_FIRST+0x00FFFFFF))
647 #define PLATFORM_PERSISTENT      (TPM_HC)((PERSISTENT_FIRST+0x00800000))
648 #define NV_INDEX_FIRST           (TPM_HC)((HR_NV_INDEX+0))
649 #define NV_INDEX_LAST            (TPM_HC)((NV_INDEX_FIRST+0x00FFFFFF))
650 #define PERMANENT_FIRST          (TPM_HC)(TPM_RH_FIRST)
```


 Copyright © TCG 2006-2020

```
651     #define PERMANENT_LAST           (TPM_HC)(TPM_RH_LAST)
652     #define HR_NV_AC                 (TPM_HC)(((TPM_HT_NV_INDEX<<HR_SHIFT)+0xD00000))
653     #define NV_AC_FIRST              (TPM_HC)((HR_NV_AC+0))
654     #define NV_AC_LAST               (TPM_HC)((HR_NV_AC+0x0000FFFF))
655     #define HR_AC                    (TPM_HC)((TPM_HT_AC<<HR_SHIFT))
656     #define AC_FIRST                 (TPM_HC)((HR_AC+0))
657     #define AC_LAST                  (TPM_HC)((HR_AC+0x0000FFFF))
658     #define TYPE_OF_TPMA_ALGORITHM  UINT32
659     #define TPMA_ALGORITHM_TO_UINT32(a)   (*((UINT32 *)&(a)))
660     #define UINT32_TO_TPMA_ALGORITHM(a)   (*((TPMA_ALGORITHM *)&(a)))
661     #define TPMA_ALGORITHM_TO_BYTE_ARRAY(i, a)                                   \
662                 UINT32_TO_BYTE_ARRAY((TPMA_ALGORITHM_TO_UINT32(i)), (a))
663     #define BYTE_ARRAY_TO_TPMA_ALGORITHM(i, a)                                   \
664                 {UINT32 x = BYTE_ARRAY_TO_UINT32(a);                             \
665                  i = UINT32_TO_TPMA_ALGORITHM(x);                                \
666                  }
667     #if USE_BIT_FIELD_STRUCTURES
668     typedef struct TPMA_ALGORITHM {                        // Table 2:30
669         unsigned    asymmetric           : 1;
670         unsigned    symmetric            : 1;
671         unsigned    hash                 : 1;
672         unsigned    object               : 1;
673         unsigned    Reserved_bits_at_4   : 4;
674         unsigned    signing              : 1;
675         unsigned    encrypting           : 1;
676         unsigned    method               : 1;
677         unsigned    Reserved_bits_at_11  : 21;
678     } TPMA_ALGORITHM;                                      /* Bits */
```

This is the initializer for a TPMA_ALGORITHM structure

```
679     #define TPMA_ALGORITHM_INITIALIZER(                                          \
680                  asymmetric, symmetric,  hash,       object,     bits_at_4,      \
681                  signing,    encrypting, method,     bits_at_11)                 \
682                 {asymmetric, symmetric,  hash,       object,     bits_at_4,      \
683                  signing,    encrypting, method,     bits_at_11}
684     #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:30 TPMA_ALGORITHM using bit masking

```
685     typedef UINT32                          TPMA_ALGORITHM;
686     #define TYPE_OF_TPMA_ALGORITHM          UINT32
687     #define TPMA_ALGORITHM_asymmetric       ((TPMA_ALGORITHM)1 << 0)
688     #define TPMA_ALGORITHM_symmetric        ((TPMA_ALGORITHM)1 << 1)
689     #define TPMA_ALGORITHM_hash             ((TPMA_ALGORITHM)1 << 2)
690     #define TPMA_ALGORITHM_object           ((TPMA_ALGORITHM)1 << 3)
691     #define TPMA_ALGORITHM_signing          ((TPMA_ALGORITHM)1 << 8)
692     #define TPMA_ALGORITHM_encrypting       ((TPMA_ALGORITHM)1 << 9)
693     #define TPMA_ALGORITHM_method           ((TPMA_ALGORITHM)1 << 10)
```

This is the initializer for a TPMA_ALGORITHM bit array.

```
694     #define TPMA_ALGORITHM_INITIALIZER(                                          \
695                  asymmetric, symmetric,  hash,       object,     bits_at_4,      \
696                  signing,    encrypting, method,     bits_at_11)                 \
697                 {(asymmetric << 0) + (symmetric << 1)  + (hash << 2)        +    \
698                  (object << 3)     + (signing << 8)     + (encrypting << 9) +    \
699                  (method << 10)}
700     #endif // USE_BIT_FIELD_STRUCTURES
701     #define TYPE_OF_TPMA_OBJECT UINT32
702     #define TPMA_OBJECT_TO_UINT32(a)      (*((UINT32 *)&(a)))
703     #define UINT32_TO_TPMA_OBJECT(a)      (*((TPMA_OBJECT *)&(a)))
704     #define TPMA_OBJECT_TO_BYTE_ARRAY(i, a)                                      \
705                 UINT32_TO_BYTE_ARRAY((TPMA_OBJECT_TO_UINT32(i)), (a))
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
706     #define BYTE_ARRAY_TO_TPMA_OBJECT(i, a)                                          \
707                 { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_OBJECT(x); }
708     #if USE_BIT_FIELD_STRUCTURES
709     typedef struct TPMA_OBJECT {                            // Table 2:31
710         unsigned    Reserved_bit_at_0        : 1;
711         unsigned    fixedTPM                 : 1;
712         unsigned    stClear                  : 1;
713         unsigned    Reserved_bit_at_3        : 1;
714         unsigned    fixedParent              : 1;
715         unsigned    sensitiveDataOrigin      : 1;
716         unsigned    userWithAuth             : 1;
717         unsigned    adminWithPolicy          : 1;
718         unsigned    Reserved_bits_at_8       : 2;
719         unsigned    noDA                     : 1;
720         unsigned    encryptedDuplication     : 1;
721         unsigned    Reserved_bits_at_12      : 4;
722         unsigned    restricted               : 1;
723         unsigned    decrypt                  : 1;
724         unsigned    sign                     : 1;
725         unsigned    x509sign                 : 1;
726         unsigned    Reserved_bits_at_20      : 12;
727     } TPMA_OBJECT;                                          /* Bits */
```

This is the initializer for a TPMA_OBJECT structure

```
728     #define TPMA_OBJECT_INITIALIZER(                                                 \
729                  bit_at_0,              fixedtpm,              stclear,               \
730                  bit_at_3,              fixedparent,           sensitivedataorigin,   \
731                  userwithauth,          adminwithpolicy,       bits_at_8,             \
732                  noda,                  encryptedduplication,  bits_at_12,            \
733                  restricted,            decrypt,               sign,                  \
734                  x509sign,              bits_at_20)                                   \
735                 {bit_at_0,              fixedtpm,              stclear,               \
736                  bit_at_3,              fixedparent,           sensitivedataorigin,   \
737                  userwithauth,          adminwithpolicy,       bits_at_8,             \
738                  noda,                  encryptedduplication,  bits_at_12,            \
739                  restricted,            decrypt,               sign,                  \
740                  x509sign,              bits_at_20}
741     #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:31 TPMA_OBJECT using bit masking

```
742     typedef UINT32                                  TPMA_OBJECT;
743     #define TYPE_OF_TPMA_OBJECT                     UINT32
744     #define TPMA_OBJECT_fixedTPM                    ((TPMA_OBJECT)1 << 1)
745     #define TPMA_OBJECT_stClear                     ((TPMA_OBJECT)1 << 2)
746     #define TPMA_OBJECT_fixedParent                 ((TPMA_OBJECT)1 << 4)
747     #define TPMA_OBJECT_sensitiveDataOrigin         ((TPMA_OBJECT)1 << 5)
748     #define TPMA_OBJECT_userWithAuth                ((TPMA_OBJECT)1 << 6)
749     #define TPMA_OBJECT_adminWithPolicy             ((TPMA_OBJECT)1 << 7)
750     #define TPMA_OBJECT_noDA                        ((TPMA_OBJECT)1 << 10)
751     #define TPMA_OBJECT_encryptedDuplication        ((TPMA_OBJECT)1 << 11)
752     #define TPMA_OBJECT_restricted                  ((TPMA_OBJECT)1 << 16)
753     #define TPMA_OBJECT_decrypt                     ((TPMA_OBJECT)1 << 17)
754     #define TPMA_OBJECT_sign                        ((TPMA_OBJECT)1 << 18)
755     #define TPMA_OBJECT_x509sign                    ((TPMA_OBJECT)1 << 19)
```

This is the initializer for a TPMA_OBJECT bit array.

```
756     #define TPMA_OBJECT_INITIALIZER(                                                 \
757                  bit_at_0,              fixedtpm,              stclear,               \
758                  bit_at_3,              fixedparent,           sensitivedataorigin,   \
759                  userwithauth,          adminwithpolicy,       bits_at_8,             \
760                  noda,                  encryptedduplication,  bits_at_12,            \
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
761                 restricted,           decrypt,              sign,                  \
762                 x509sign,             bits_at_20)                                  \
763                {(fixedtpm << 1)              + (stclear << 2)              +       \
764                 (fixedparent << 4)           + (sensitivedataorigin << 5)  +       \
765                 (userwithauth << 6)          + (adminwithpolicy << 7)      +       \
766                 (noda << 10)                 + (encryptedduplication << 11) +      \
767                 (restricted << 16)           + (decrypt << 17)             +       \
768                 (sign << 18)                 + (x509sign << 19)}
769    #endif // USE_BIT_FIELD_STRUCTURES
770    #define TYPE_OF_TPMA_SESSION    UINT8
771    #define TPMA_SESSION_TO_UINT8(a)      (*((UINT8 *)&(a)))
772    #define UINT8_TO_TPMA_SESSION(a)      (*((TPMA_SESSION *)&(a)))
773    #define TPMA_SESSION_TO_BYTE_ARRAY(i, a)                                         \
774                UINT8_TO_BYTE_ARRAY((TPMA_SESSION_TO_UINT8(i)), (a))
775    #define BYTE_ARRAY_TO_TPMA_SESSION(i, a)                                         \
776                { UINT8 x = BYTE_ARRAY_TO_UINT8(a); i = UINT8_TO_TPMA_SESSION(x); }
777    #if USE_BIT_FIELD_STRUCTURES
778    typedef struct TPMA_SESSION {                          // Table 2:32
779        unsigned    continueSession      : 1;
780        unsigned    auditExclusive       : 1;
781        unsigned    auditReset           : 1;
782        unsigned    Reserved_bits_at_3   : 2;
783        unsigned    decrypt              : 1;
784        unsigned    encrypt              : 1;
785        unsigned    audit                : 1;
786    } TPMA_SESSION;                                        /* Bits */
```

This is the initializer for a TPMA_SESSION structure

```
787    #define TPMA_SESSION_INITIALIZER(                                                \
788                 continuesession, auditexclusive,  auditreset,      bits_at_3,        \
789                 decrypt,         encrypt,         audit)                             \
790                {continuesession, auditexclusive,  auditreset,      bits_at_3,        \
791                 decrypt,         encrypt,         audit}
792    #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:32 TPMA_SESSION using bit masking

```
793    typedef UINT8                               TPMA_SESSION;
794    #define TYPE_OF_TPMA_SESSION                UINT8
795    #define TPMA_SESSION_continueSession        ((TPMA_SESSION)1 << 0)
796    #define TPMA_SESSION_auditExclusive         ((TPMA_SESSION)1 << 1)
797    #define TPMA_SESSION_auditReset             ((TPMA_SESSION)1 << 2)
798    #define TPMA_SESSION_decrypt                ((TPMA_SESSION)1 << 5)
799    #define TPMA_SESSION_encrypt                ((TPMA_SESSION)1 << 6)
800    #define TPMA_SESSION_audit                  ((TPMA_SESSION)1 << 7)
```

This is the initializer for a TPMA_SESSION bit array.

```
801    #define TPMA_SESSION_INITIALIZER(                                                \
802                 continuesession, auditexclusive,  auditreset,      bits_at_3,        \
803                 decrypt,         encrypt,         audit)                             \
804                {(continuesession << 0) + (auditexclusive << 1)  +                    \
805                 (auditreset << 2)      + (decrypt << 5)         +                    \
806                 (encrypt << 6)         + (audit << 7)}
807    #endif // USE_BIT_FIELD_STRUCTURES
808    #define TYPE_OF_TPMA_LOCALITY   UINT8
809    #define TPMA_LOCALITY_TO_UINT8(a)     (*((UINT8 *)&(a)))
810    #define UINT8_TO_TPMA_LOCALITY(a)     (*((TPMA_LOCALITY *)&(a)))
811    #define TPMA_LOCALITY_TO_BYTE_ARRAY(i, a)                                        \
812                UINT8_TO_BYTE_ARRAY((TPMA_LOCALITY_TO_UINT8(i)), (a))
813    #define BYTE_ARRAY_TO_TPMA_LOCALITY(i, a)                                        \
814                { UINT8 x = BYTE_ARRAY_TO_UINT8(a); i = UINT8_TO_TPMA_LOCALITY(x); }
815    #if USE_BIT_FIELD_STRUCTURES
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
816 typedef struct TPMA_LOCALITY {                          // Table 2:33
817     unsigned    TPM_LOC_ZERO          : 1;
818     unsigned    TPM_LOC_ONE           : 1;
819     unsigned    TPM_LOC_TWO           : 1;
820     unsigned    TPM_LOC_THREE         : 1;
821     unsigned    TPM_LOC_FOUR          : 1;
822     unsigned    Extended              : 3;
823 } TPMA_LOCALITY;                                        /* Bits */
```

This is the initializer for a TPMA_LOCALITY structure

```
824 #define TPMA_LOCALITY_INITIALIZER(                                          \
825              tpm_loc_zero,  tpm_loc_one,   tpm_loc_two,   tpm_loc_three,     \
826              tpm_loc_four,  extended)                                       \
827             {tpm_loc_zero,  tpm_loc_one,   tpm_loc_two,   tpm_loc_three,     \
828              tpm_loc_four,  extended}
829 #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:33 TPMA_LOCALITY using bit masking

```
830 typedef UINT8                               TPMA_LOCALITY;
831 #define TYPE_OF_TPMA_LOCALITY               UINT8
832 #define TPMA_LOCALITY_TPM_LOC_ZERO          ((TPMA_LOCALITY)1 << 0)
833 #define TPMA_LOCALITY_TPM_LOC_ONE           ((TPMA_LOCALITY)1 << 1)
834 #define TPMA_LOCALITY_TPM_LOC_TWO           ((TPMA_LOCALITY)1 << 2)
835 #define TPMA_LOCALITY_TPM_LOC_THREE         ((TPMA_LOCALITY)1 << 3)
836 #define TPMA_LOCALITY_TPM_LOC_FOUR          ((TPMA_LOCALITY)1 << 4)
837 #define TPMA_LOCALITY_Extended_SHIFT        5
838 #define TPMA_LOCALITY_Extended              ((TPMA_LOCALITY)0x7 << 5)
```

This is the initializer for a TPMA_LOCALITY bit array.

```
839 #define TPMA_LOCALITY_INITIALIZER(                                          \
840              tpm_loc_zero,  tpm_loc_one,   tpm_loc_two,   tpm_loc_three,     \
841              tpm_loc_four,  extended)                                       \
842             {(tpm_loc_zero << 0)  + (tpm_loc_one << 1)   + (tpm_loc_two << 2)   +  \
843              (tpm_loc_three << 3) + (tpm_loc_four << 4)  + (extended << 5)}
844 #endif // USE_BIT_FIELD_STRUCTURES
845 #define TYPE_OF_TPMA_PERMANENT  UINT32
846 #define TPMA_PERMANENT_TO_UINT32(a)   (*((UINT32 *)&(a)))
847 #define UINT32_TO_TPMA_PERMANENT(a)   (*((TPMA_PERMANENT *)&(a)))
848 #define TPMA_PERMANENT_TO_BYTE_ARRAY(i, a)                                  \
849             UINT32_TO_BYTE_ARRAY((TPMA_PERMANENT_TO_UINT32(i)), (a))
850 #define BYTE_ARRAY_TO_TPMA_PERMANENT(i, a)                                  \
851             {UINT32 x = BYTE_ARRAY_TO_UINT32(a);                            \
852              i = UINT32_TO_TPMA_PERMANENT(x);                               \
853              }
854 #if USE_BIT_FIELD_STRUCTURES
855 typedef struct TPMA_PERMANENT {                         // Table 2:34
856     unsigned    ownerAuthSet          : 1;
857     unsigned    endorsementAuthSet    : 1;
858     unsigned    lockoutAuthSet        : 1;
859     unsigned    Reserved_bits_at_3    : 5;
860     unsigned    disableClear          : 1;
861     unsigned    inLockout             : 1;
862     unsigned    tpmGeneratedEPS       : 1;
863     unsigned    Reserved_bits_at_11   : 21;
864 } TPMA_PERMANENT;                                       /* Bits */
```

This is the initializer for a TPMA_PERMANENT structure

```
865 #define TPMA_PERMANENT_INITIALIZER(                                         \
866              ownerauthset,       endorsementauthset, lockoutauthset,        \
867              bits_at_3,          disableclear,       inlockout,             \
```

```
868                tpmgeneratedeps,    bits_at_11)                                        \
869               {ownerauthset,       endorsementauthset, lockoutauthset,                \
870                bits_at_3,          disableclear,       inlockout,                     \
871                tpmgeneratedeps,    bits_at_11}
872    #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:34 TPMA_PERMANENT using bit masking

```
873    typedef UINT32                              TPMA_PERMANENT;
874    #define TYPE_OF_TPMA_PERMANENT              UINT32
875    #define TPMA_PERMANENT_ownerAuthSet         ((TPMA_PERMANENT)1 << 0)
876    #define TPMA_PERMANENT_endorsementAuthSet   ((TPMA_PERMANENT)1 << 1)
877    #define TPMA_PERMANENT_lockoutAuthSet       ((TPMA_PERMANENT)1 << 2)
878    #define TPMA_PERMANENT_disableClear         ((TPMA_PERMANENT)1 << 8)
879    #define TPMA_PERMANENT_inLockout            ((TPMA_PERMANENT)1 << 9)
880    #define TPMA_PERMANENT_tpmGeneratedEPS      ((TPMA_PERMANENT)1 << 10)
```

This is the initializer for a TPMA_PERMANENT bit array.

```
881    #define TPMA_PERMANENT_INITIALIZER(                                                    \
882                ownerauthset,       endorsementauthset, lockoutauthset,                \
883                bits_at_3,          disableclear,       inlockout,                     \
884                tpmgeneratedeps,    bits_at_11)                                        \
885               {(ownerauthset << 0)       + (endorsementauthset << 1) +                \
886                (lockoutauthset << 2)     + (disableclear << 8)       +                \
887                (inlockout << 9)          + (tpmgeneratedeps << 10)}
888    #endif // USE_BIT_FIELD_STRUCTURES
889    #define TYPE_OF_TPMA_STARTUP_CLEAR  UINT32
890    #define TPMA_STARTUP_CLEAR_TO_UINT32(a)   (*((UINT32 *)&(a)))
891    #define UINT32_TO_TPMA_STARTUP_CLEAR(a)   (*((TPMA_STARTUP_CLEAR *)&(a)))
892    #define TPMA_STARTUP_CLEAR_TO_BYTE_ARRAY(i, a)                                         \
893               UINT32_TO_BYTE_ARRAY((TPMA_STARTUP_CLEAR_TO_UINT32(i)), (a))
894    #define BYTE_ARRAY_TO_TPMA_STARTUP_CLEAR(i, a)                                         \
895               {UINT32 x = BYTE_ARRAY_TO_UINT32(a);                                     \
896                i = UINT32_TO_TPMA_STARTUP_CLEAR(x);                                    \
897                }
898    #if USE_BIT_FIELD_STRUCTURES
899    typedef struct TPMA_STARTUP_CLEAR {                      // Table 2:35
900        unsigned    phEnable             : 1;
901        unsigned    shEnable             : 1;
902        unsigned    ehEnable             : 1;
903        unsigned    phEnableNV           : 1;
904        unsigned    Reserved_bits_at_4   : 27;
905        unsigned    orderly              : 1;
906    } TPMA_STARTUP_CLEAR;                                    /* Bits */
```

This is the initializer for a TPMA_STARTUP_CLEAR structure

```
907    #define TPMA_STARTUP_CLEAR_INITIALIZER(                                                \
908                phenable, shenable, ehenable, phenablenv, bits_at_4, orderly)         \
909               {phenable, shenable, ehenable, phenablenv, bits_at_4, orderly}
910    #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:35 TPMA_STARTUP_CLEAR using bit masking

```
911    typedef UINT32                          TPMA_STARTUP_CLEAR;
912    #define TYPE_OF_TPMA_STARTUP_CLEAR      UINT32
913    #define TPMA_STARTUP_CLEAR_phEnable     ((TPMA_STARTUP_CLEAR)1 << 0)
914    #define TPMA_STARTUP_CLEAR_shEnable     ((TPMA_STARTUP_CLEAR)1 << 1)
915    #define TPMA_STARTUP_CLEAR_ehEnable     ((TPMA_STARTUP_CLEAR)1 << 2)
916    #define TPMA_STARTUP_CLEAR_phEnableNV   ((TPMA_STARTUP_CLEAR)1 << 3)
917    #define TPMA_STARTUP_CLEAR_orderly      ((TPMA_STARTUP_CLEAR)1 << 31)
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

This is the initializer for a TPMA_STARTUP_CLEAR bit array.

```
918 #define TPMA_STARTUP_CLEAR_INITIALIZER(                                              \
919              phenable, shenable, ehenable, phenablenv, bits_at_4, orderly)         \
920             {(phenable << 0)   + (shenable << 1)   + (ehenable << 2)   +           \
921              (phenablenv << 3) + (orderly << 31)}
922 #endif // USE_BIT_FIELD_STRUCTURES
923 #define TYPE_OF_TPMA_MEMORY UINT32
924 #define TPMA_MEMORY_TO_UINT32(a)     (*((UINT32 *)&(a)))
925 #define UINT32_TO_TPMA_MEMORY(a)     (*((TPMA_MEMORY *)&(a)))
926 #define TPMA_MEMORY_TO_BYTE_ARRAY(i, a)                                              \
927             UINT32_TO_BYTE_ARRAY((TPMA_MEMORY_TO_UINT32(i)), (a))
928 #define BYTE_ARRAY_TO_TPMA_MEMORY(i, a)                                              \
929             { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_MEMORY(x); }
930 #if USE_BIT_FIELD_STRUCTURES
931 typedef struct TPMA_MEMORY {                         // Table 2:36
932     unsigned    sharedRAM            : 1;
933     unsigned    sharedNV             : 1;
934     unsigned    objectCopiedToRam    : 1;
935     unsigned    Reserved_bits_at_3   : 29;
936 } TPMA_MEMORY;                                       /* Bits */
```

This is the initializer for a TPMA_MEMORY structure

```
937 #define TPMA_MEMORY_INITIALIZER(                                                     \
938              sharedram, sharednv, objectcopiedtoram, bits_at_3)                     \
939             {sharedram, sharednv, objectcopiedtoram, bits_at_3}
940 #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:36 TPMA_MEMORY using bit masking

```
941 typedef UINT32                              TPMA_MEMORY;
942 #define TYPE_OF_TPMA_MEMORY                 UINT32
943 #define TPMA_MEMORY_sharedRAM               ((TPMA_MEMORY)1 << 0)
944 #define TPMA_MEMORY_sharedNV                ((TPMA_MEMORY)1 << 1)
945 #define TPMA_MEMORY_objectCopiedToRam       ((TPMA_MEMORY)1 << 2)
```

This is the initializer for a TPMA_MEMORY bit array.

```
946 #define TPMA_MEMORY_INITIALIZER(                                                     \
947              sharedram, sharednv, objectcopiedtoram, bits_at_3)                     \
948             {(sharedram << 0) + (sharednv << 1) + (objectcopiedtoram << 2)}
949 #endif // USE_BIT_FIELD_STRUCTURES
950 #define TYPE_OF_TPMA_CC     UINT32
951 #define TPMA_CC_TO_UINT32(a)     (*((UINT32 *)&(a)))
952 #define UINT32_TO_TPMA_CC(a)     (*((TPMA_CC *)&(a)))
953 #define TPMA_CC_TO_BYTE_ARRAY(i, a)                                                  \
954             UINT32_TO_BYTE_ARRAY((TPMA_CC_TO_UINT32(i)), (a))
955 #define BYTE_ARRAY_TO_TPMA_CC(i, a)                                                  \
956             { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_CC(x); }
957 #if USE_BIT_FIELD_STRUCTURES
958 typedef struct TPMA_CC {                             // Table 2:37
959     unsigned    commandIndex         : 16;
960     unsigned    Reserved_bits_at_16  : 6;
961     unsigned    nv                   : 1;
962     unsigned    extensive            : 1;
963     unsigned    flushed              : 1;
964     unsigned    cHandles             : 3;
965     unsigned    rHandle              : 1;
966     unsigned    V                    : 1;
967     unsigned    Reserved_bits_at_30  : 2;
968 } TPMA_CC;                                           /* Bits */
```

This is the initializer for a TPMA_CC structure

```
969   #define TPMA_CC_INITIALIZER(                                                    \
970                commandindex, bits_at_16,   nv,           extensive,    flushed,     \
971                chandles,     rhandle,      v,            bits_at_30)                \
972               {commandindex, bits_at_16,   nv,           extensive,    flushed,     \
973                chandles,     rhandle,      v,            bits_at_30}
974   #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:37 TPMA_CC using bit masking

```
975   typedef UINT32                        TPMA_CC;
976   #define TYPE_OF_TPMA_CC               UINT32
977   #define TPMA_CC_commandIndex_SHIFT    0
978   #define TPMA_CC_commandIndex          ((TPMA_CC)0xffff << 0)
979   #define TPMA_CC_nv                    ((TPMA_CC)1 << 22)
980   #define TPMA_CC_extensive             ((TPMA_CC)1 << 23)
981   #define TPMA_CC_flushed               ((TPMA_CC)1 << 24)
982   #define TPMA_CC_cHandles_SHIFT        25
983   #define TPMA_CC_cHandles              ((TPMA_CC)0x7 << 25)
984   #define TPMA_CC_rHandle               ((TPMA_CC)1 << 28)
985   #define TPMA_CC_V                     ((TPMA_CC)1 << 29)
```

This is the initializer for a TPMA_CC bit array.

```
986   #define TPMA_CC_INITIALIZER(                                                    \
987                commandindex, bits_at_16,   nv,           extensive,    flushed,     \
988                chandles,     rhandle,      v,            bits_at_30)                \
989               {(commandindex << 0) + (nv << 22)          + (extensive << 23)   +    \
990                (flushed << 24)     + (chandles << 25)    + (rhandle << 28)     +    \
991                (v << 29)}
992   #endif // USE_BIT_FIELD_STRUCTURES
993   #define TYPE_OF_TPMA_MODES  UINT32
994   #define TPMA_MODES_TO_UINT32(a)  (*((UINT32 *)&(a)))
995   #define UINT32_TO_TPMA_MODES(a)  (*((TPMA_MODES *)&(a)))
996   #define TPMA_MODES_TO_BYTE_ARRAY(i, a)                                           \
997               UINT32_TO_BYTE_ARRAY((TPMA_MODES_TO_UINT32(i)), (a))
998   #define BYTE_ARRAY_TO_TPMA_MODES(i, a)                                           \
999               { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_MODES(x); }
1000  #if USE_BIT_FIELD_STRUCTURES
1001  typedef struct TPMA_MODES {                        // Table 2:38
1002      unsigned     FIPS_140_2          : 1;
1003      unsigned     Reserved_bits_at_1   : 31;
1004  } TPMA_MODES;                                      /* Bits */
```

This is the initializer for a TPMA_MODES structure

```
1005  #define TPMA_MODES_INITIALIZER(fips_140_2, bits_at_1) {fips_140_2, bits_at_1}
1006  #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:38 TPMA_MODES using bit masking

```
1007  typedef UINT32                   TPMA_MODES;
1008  #define TYPE_OF_TPMA_MODES       UINT32
1009  #define TPMA_MODES_FIPS_140_2    ((TPMA_MODES)1 << 0)
```

This is the initializer for a TPMA_MODES bit array.

```
1010  #define TPMA_MODES_INITIALIZER(fips_140_2, bits_at_1) {(fips_140_2 << 0)}
1011  #endif // USE_BIT_FIELD_STRUCTURES
1012  #define TYPE_OF_TPMA_X509_KEY_USAGE UINT32
1013  #define TPMA_X509_KEY_USAGE_TO_UINT32(a)      (*((UINT32 *)&(a)))
1014  #define UINT32_TO_TPMA_X509_KEY_USAGE(a)      (*((TPMA_X509_KEY_USAGE *)&(a)))
1015  #define TPMA_X509_KEY_USAGE_TO_BYTE_ARRAY(i, a)                                  \
1016              UINT32_TO_BYTE_ARRAY((TPMA_X509_KEY_USAGE_TO_UINT32(i)), (a))
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1017  #define BYTE_ARRAY_TO_TPMA_X509_KEY_USAGE(i, a)                              \
1018              {UINT32 x = BYTE_ARRAY_TO_UINT32(a);                             \
1019               i = UINT32_TO_TPMA_X509_KEY_USAGE(x);                           \
1020              }
1021  #if USE_BIT_FIELD_STRUCTURES
1022  typedef struct TPMA_X509_KEY_USAGE {                    // Table 2:39
1023      unsigned    Reserved_bits_at_0    : 23;
1024      unsigned    decipherOnly          : 1;
1025      unsigned    encipherOnly          : 1;
1026      unsigned    cRLSign               : 1;
1027      unsigned    keyCertSign           : 1;
1028      unsigned    keyAgreement          : 1;
1029      unsigned    dataEncipherment      : 1;
1030      unsigned    keyEncipherment       : 1;
1031      unsigned    nonrepudiation        : 1;
1032      unsigned    digitalSignature      : 1;
1033  } TPMA_X509_KEY_USAGE;                                  /* Bits */
```

This is the initializer for a TPMA_X509_KEY_USAGE structure

```
1034  #define TPMA_X509_KEY_USAGE_INITIALIZER(                                     \
1035               bits_at_0,        decipheronly,     encipheronly,             \
1036               crlsign,          keycertsign,      keyagreement,             \
1037               dataencipherment, keyencipherment,  nonrepudiation,           \
1038               digitalsignature)                                             \
1039              {bits_at_0,        decipheronly,     encipheronly,             \
1040               crlsign,          keycertsign,      keyagreement,             \
1041               dataencipherment, keyencipherment,  nonrepudiation,           \
1042               digitalsignature}
1043  #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:39 TPMA_X509_KEY_USAGE using bit masking

```
1044  typedef UINT32                                 TPMA_X509_KEY_USAGE;
1045  #define TYPE_OF_TPMA_X509_KEY_USAGE            UINT32
1046  #define TPMA_X509_KEY_USAGE_decipherOnly       ((TPMA_X509_KEY_USAGE)1 << 23)
1047  #define TPMA_X509_KEY_USAGE_encipherOnly       ((TPMA_X509_KEY_USAGE)1 << 24)
1048  #define TPMA_X509_KEY_USAGE_cRLSign            ((TPMA_X509_KEY_USAGE)1 << 25)
1049  #define TPMA_X509_KEY_USAGE_keyCertSign        ((TPMA_X509_KEY_USAGE)1 << 26)
1050  #define TPMA_X509_KEY_USAGE_keyAgreement       ((TPMA_X509_KEY_USAGE)1 << 27)
1051  #define TPMA_X509_KEY_USAGE_dataEncipherment   ((TPMA_X509_KEY_USAGE)1 << 28)
1052  #define TPMA_X509_KEY_USAGE_keyEncipherment    ((TPMA_X509_KEY_USAGE)1 << 29)
1053  #define TPMA_X509_KEY_USAGE_nonrepudiation     ((TPMA_X509_KEY_USAGE)1 << 30)
1054  #define TPMA_X509_KEY_USAGE_digitalSignature   ((TPMA_X509_KEY_USAGE)1 << 31)
```

This is the initializer for a TPMA_X509_KEY_USAGE bit array.

```
1055  #define TPMA_X509_KEY_USAGE_INITIALIZER(                                     \
1056               bits_at_0,        decipheronly,     encipheronly,             \
1057               crlsign,          keycertsign,      keyagreement,             \
1058               dataencipherment, keyencipherment,  nonrepudiation,           \
1059               digitalsignature)                                             \
1060              {(decipheronly << 23)    + (encipheronly << 24)     +          \
1061               (crlsign << 25)         + (keycertsign << 26)      +          \
1062               (keyagreement << 27)    + (dataencipherment << 28) +          \
1063               (keyencipherment << 29) + (nonrepudiation << 30)   +          \
1064               (digitalsignature << 31)}
1065  #endif // USE_BIT_FIELD_STRUCTURES
1066  #define TYPE_OF_TPMA_ACT    UINT32
1067  #define TPMA_ACT_TO_UINT32(a)     (*((UINT32 *)&(a)))
1068  #define UINT32_TO_TPMA_ACT(a)     (*((TPMA_ACT *)&(a)))
1069  #define TPMA_ACT_TO_BYTE_ARRAY(i, a)                                         \
1070              UINT32_TO_BYTE_ARRAY((TPMA_ACT_TO_UINT32(i)), (a))
1071  #define BYTE_ARRAY_TO_TPMA_ACT(i, a)                                         \
```

 Copyright © TCG 2006-2020

```
1072              { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_ACT(x); }
1073  #if USE_BIT_FIELD_STRUCTURES
1074  typedef struct TPMA_ACT {                               // Table 2:40
1075      unsigned    signaled              : 1;
1076      unsigned    preserveSignaled      : 1;
1077      unsigned    Reserved_bits_at_2    : 30;
1078  } TPMA_ACT;                                             /* Bits */
```

This is the initializer for a TPMA_ACT structure

```
1079  #define TPMA_ACT_INITIALIZER(signaled, preservesignaled, bits_at_2)               \
1080              {signaled, preservesignaled, bits_at_2}
1081  #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:40 TPMA_ACT using bit masking

```
1082  typedef UINT32                          TPMA_ACT;
1083  #define TYPE_OF_TPMA_ACT                UINT32
1084  #define TPMA_ACT_signaled               ((TPMA_ACT)1 << 0)
1085  #define TPMA_ACT_preserveSignaled       ((TPMA_ACT)1 << 1)
```

This is the initializer for a TPMA_ACT bit array.

```
1086  #define TPMA_ACT_INITIALIZER(signaled, preservesignaled, bits_at_2)               \
1087              {(signaled << 0) + (preservesignaled << 1)}
1088  #endif // USE_BIT_FIELD_STRUCTURES
1089  typedef BYTE                TPMI_YES_NO;            // Table 2:41   /* Interface */
1090  typedef TPM_HANDLE          TPMI_DH_OBJECT;         // Table 2:42   /* Interface */
1091  typedef TPM_HANDLE          TPMI_DH_PARENT;         // Table 2:43   /* Interface */
1092  typedef TPM_HANDLE          TPMI_DH_PERSISTENT;     // Table 2:44   /* Interface */
1093  typedef TPM_HANDLE          TPMI_DH_ENTITY;         // Table 2:45   /* Interface */
1094  typedef TPM_HANDLE          TPMI_DH_PCR;            // Table 2:46   /* Interface */
1095  typedef TPM_HANDLE          TPMI_SH_AUTH_SESSION;   // Table 2:47   /* Interface */
1096  typedef TPM_HANDLE          TPMI_SH_HMAC;           // Table 2:48   /* Interface */
1097  typedef TPM_HANDLE          TPMI_SH_POLICY;         // Table 2:49   /* Interface */
1098  typedef TPM_HANDLE          TPMI_DH_CONTEXT;        // Table 2:50   /* Interface */
1099  typedef TPM_HANDLE          TPMI_DH_SAVED;          // Table 2:51   /* Interface */
1100  typedef TPM_HANDLE          TPMI_RH_HIERARCHY;      // Table 2:52   /* Interface */
1101  typedef TPM_HANDLE          TPMI_RH_ENABLES;        // Table 2:53   /* Interface */
1102  typedef TPM_HANDLE          TPMI_RH_HIERARCHY_AUTH; // Table 2:54   /* Interface */
1103  typedef TPM_HANDLE          TPMI_RH_HIERARCHY_POLICY;
1104  typedef TPM_HANDLE          TPMI_RH_PLATFORM;       // Table 2:56   /* Interface */
1105  typedef TPM_HANDLE          TPMI_RH_OWNER;          // Table 2:57   /* Interface */
1106  typedef TPM_HANDLE          TPMI_RH_ENDORSEMENT;    // Table 2:58   /* Interface */
1107  typedef TPM_HANDLE          TPMI_RH_PROVISION;      // Table 2:59   /* Interface */
1108  typedef TPM_HANDLE          TPMI_RH_CLEAR;          // Table 2:60   /* Interface */
1109  typedef TPM_HANDLE          TPMI_RH_NV_AUTH;        // Table 2:61   /* Interface */
1110  typedef TPM_HANDLE          TPMI_RH_LOCKOUT;        // Table 2:62   /* Interface */
1111  typedef TPM_HANDLE          TPMI_RH_NV_INDEX;       // Table 2:63   /* Interface */
1112  typedef TPM_HANDLE          TPMI_RH_AC;             // Table 2:64   /* Interface */
1113  typedef TPM_HANDLE          TPMI_RH_ACT;            // Table 2:65   /* Interface */
1114  typedef TPM_ALG_ID          TPMI_ALG_HASH;          // Table 2:66   /* Interface */
1115  typedef TPM_ALG_ID          TPMI_ALG_ASYM;          // Table 2:67   /* Interface */
1116  typedef TPM_ALG_ID          TPMI_ALG_SYM;           // Table 2:68   /* Interface */
1117  typedef TPM_ALG_ID          TPMI_ALG_SYM_OBJECT;    // Table 2:69   /* Interface */
1118  typedef TPM_ALG_ID          TPMI_ALG_SYM_MODE;      // Table 2:70   /* Interface */
1119  typedef TPM_ALG_ID          TPMI_ALG_KDF;           // Table 2:71   /* Interface */
1120  typedef TPM_ALG_ID          TPMI_ALG_SIG_SCHEME;    // Table 2:72   /* Interface */
1121  typedef TPM_ALG_ID          TPMI_ECC_KEY_EXCHANGE;  // Table 2:73   /* Interface */
1122  typedef TPM_ST              TPMI_ST_COMMAND_TAG;    // Table 2:74   /* Interface */
1123  typedef TPM_ALG_ID          TPMI_ALG_MAC_SCHEME;    // Table 2:75   /* Interface */
1124  typedef TPM_ALG_ID          TPMI_ALG_CIPHER_MODE;   // Table 2:76   /* Interface */
1125  typedef BYTE                TPMS_EMPTY;             // Table 2:77
1126  typedef struct {                                    // Table 2:78
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1127        TPM_ALG_ID                  alg;
1128        TPMA_ALGORITHM              attributes;
1129    } TPMS_ALGORITHM_DESCRIPTION;                           /* Structure */
1130    typedef union {                                         // Table 2:79
1131    #if ALG_SHA1
1132        BYTE                        sha1[SHA1_DIGEST_SIZE];
1133    #endif // ALG_SHA1
1134    #if ALG_SHA256
1135        BYTE                        sha256[SHA256_DIGEST_SIZE];
1136    #endif // ALG_SHA256
1137    #if ALG_SHA384
1138        BYTE                        sha384[SHA384_DIGEST_SIZE];
1139    #endif // ALG_SHA384
1140    #if ALG_SHA512
1141        BYTE                        sha512[SHA512_DIGEST_SIZE];
1142    #endif // ALG_SHA512
1143    #if ALG_SM3_256
1144        BYTE                        sm3_256[SM3_256_DIGEST_SIZE];
1145    #endif // ALG_SM3_256
1146    #if ALG_SHA3_256
1147        BYTE                        sha3_256[SHA3_256_DIGEST_SIZE];
1148    #endif // ALG_SHA3_256
1149    #if ALG_SHA3_384
1150        BYTE                        sha3_384[SHA3_384_DIGEST_SIZE];
1151    #endif // ALG_SHA3_384
1152    #if ALG_SHA3_512
1153        BYTE                        sha3_512[SHA3_512_DIGEST_SIZE];
1154    #endif // ALG_SHA3_512
1155    } TPMU_HA;                                              /* Structure */
1156    typedef struct {                                        // Table 2:80
1157        TPMI_ALG_HASH               hashAlg;
1158        TPMU_HA                     digest;
1159    } TPMT_HA;                                              /* Structure */
1160    typedef union {                                         // Table 2:81
1161        struct {
1162            UINT16                  size;
1163            BYTE                    buffer[sizeof(TPMU_HA)];
1164        }            t;
1165        TPM2B        b;
1166    } TPM2B_DIGEST;                                         /* Structure */
1167    typedef union {                                         // Table 2:82
1168        struct {
1169            UINT16                  size;
1170            BYTE                    buffer[sizeof(TPMT_HA)];
1171        }            t;
1172        TPM2B        b;
1173    } TPM2B_DATA;                                           /* Structure */
```

Table 2:83 - Definition of Types for TPM2B_NONCE

```
1174    typedef TPM2B_DIGEST        TPM2B_NONCE;
```

Table 2:84 - Definition of Types for TPM2B_AUTH

```
1175    typedef TPM2B_DIGEST        TPM2B_AUTH;
```

Table 2:85 - Definition of Types for TPM2B_OPERAND

```
1176    typedef TPM2B_DIGEST        TPM2B_OPERAND;
1177    typedef union {                                         // Table 2:86
1178        struct {
1179            UINT16                  size;
1180            BYTE                    buffer[1024];
1181        }            t;
```

 Copyright © TCG 2006-2020

```
1182        TPM2B         b;
1183    } TPM2B_EVENT;                                       /* Structure */
1184    typedef union {                                      // Table 2:87
1185        struct {
1186            UINT16                size;
1187            BYTE                  buffer[MAX_DIGEST_BUFFER];
1188        }             t;
1189        TPM2B         b;
1190    } TPM2B_MAX_BUFFER;                                  /* Structure */
1191    typedef union {                                      // Table 2:88
1192        struct {
1193            UINT16                size;
1194            BYTE                  buffer[MAX_NV_BUFFER_SIZE];
1195        }             t;
1196        TPM2B         b;
1197    } TPM2B_MAX_NV_BUFFER;                               /* Structure */
1198    typedef union {                                      // Table 2:89
1199        struct {
1200            UINT16                size;
1201            BYTE                  buffer[sizeof(UINT64)];
1202        }             t;
1203        TPM2B         b;
1204    } TPM2B_TIMEOUT;                                     /* Structure */
1205    typedef union {                                      // Table 2:90
1206        struct {
1207            UINT16                size;
1208            BYTE                  buffer[MAX_SYM_BLOCK_SIZE];
1209        }             t;
1210        TPM2B         b;
1211    } TPM2B_IV;                                          /* Structure */
1212    typedef union {                                      // Table 2:91
1213        TPMT_HA                   digest;
1214        TPM_HANDLE                handle;
1215    } TPMU_NAME;                                         /* Structure */
1216    typedef union {                                      // Table 2:92
1217        struct {
1218            UINT16                size;
1219            BYTE                  name[sizeof(TPMU_NAME)];
1220        }             t;
1221        TPM2B         b;
1222    } TPM2B_NAME;                                        /* Structure */
1223    typedef struct {                                     // Table 2:93
1224        UINT8                     sizeofSelect;
1225        BYTE                      pcrSelect[PCR_SELECT_MAX];
1226    } TPMS_PCR_SELECT;                                   /* Structure */
1227    typedef struct {                                     // Table 2:94
1228        TPMI_ALG_HASH             hash;
1229        UINT8                     sizeofSelect;
1230        BYTE                      pcrSelect[PCR_SELECT_MAX];
1231    } TPMS_PCR_SELECTION;                                /* Structure */
1232    typedef struct {                                     // Table 2:97
1233        TPM_ST                    tag;
1234        TPMI_RH_HIERARCHY         hierarchy;
1235        TPM2B_DIGEST              digest;
1236    } TPMT_TK_CREATION;                                  /* Structure */
1237    typedef struct {                                     // Table 2:98
1238        TPM_ST                    tag;
1239        TPMI_RH_HIERARCHY         hierarchy;
1240        TPM2B_DIGEST              digest;
1241    } TPMT_TK_VERIFIED;                                  /* Structure */
1242    typedef struct {                                     // Table 2:99
1243        TPM_ST                    tag;
1244        TPMI_RH_HIERARCHY         hierarchy;
1245        TPM2B_DIGEST              digest;
1246    } TPMT_TK_AUTH;                                      /* Structure */
1247    typedef struct {                                     // Table 2:100
```

```
1248        TPM_ST                    tag;
1249        TPMI_RH_HIERARCHY         hierarchy;
1250        TPM2B_DIGEST              digest;
1251    } TPMT_TK_HASHCHECK;                                  /* Structure */
1252    typedef struct {                                      // Table 2:101
1253        TPM_ALG_ID                alg;
1254        TPMA_ALGORITHM            algProperties;
1255    } TPMS_ALG_PROPERTY;                                  /* Structure */
1256    typedef struct {                                      // Table 2:102
1257        TPM_PT                    property;
1258        UINT32                    value;
1259    } TPMS_TAGGED_PROPERTY;                               /* Structure */
1260    typedef struct {                                      // Table 2:103
1261        TPM_PT_PCR                tag;
1262        UINT8                     sizeofSelect;
1263        BYTE                      pcrSelect[PCR_SELECT_MAX];
1264    } TPMS_TAGGED_PCR_SELECT;                             /* Structure */
1265    typedef struct {                                      // Table 2:104
1266        TPM_HANDLE                handle;
1267        TPMT_HA                   policyHash;
1268    } TPMS_TAGGED_POLICY;                                 /* Structure */
1269    typedef struct {                                      // Table 2:105
1270        TPM_HANDLE                handle;
1271        UINT32                    timeout;
1272        TPMA_ACT                  attributes;
1273    } TPMS_ACT_DATA;                                      /* Structure */
1274    typedef struct {                                      // Table 2:106
1275        UINT32                    count;
1276        TPM_CC                    commandCodes[MAX_CAP_CC];
1277    } TPML_CC;                                            /* Structure */
1278    typedef struct {                                      // Table 2:107
1279        UINT32                    count;
1280        TPMA_CC                   commandAttributes[MAX_CAP_CC];
1281    } TPML_CCA;                                           /* Structure */
1282    typedef struct {                                      // Table 2:108
1283        UINT32                    count;
1284        TPM_ALG_ID                algorithms[MAX_ALG_LIST_SIZE];
1285    } TPML_ALG;                                           /* Structure */
1286    typedef struct {                                      // Table 2:109
1287        UINT32                    count;
1288        TPM_HANDLE                handle[MAX_CAP_HANDLES];
1289    } TPML_HANDLE;                                        /* Structure */
1290    typedef struct {                                      // Table 2:110
1291        UINT32                    count;
1292        TPM2B_DIGEST              digests[8];
1293    } TPML_DIGEST;                                        /* Structure */
1294    typedef struct {                                      // Table 2:111
1295        UINT32                    count;
1296        TPMT_HA                   digests[HASH_COUNT];
1297    } TPML_DIGEST_VALUES;                                 /* Structure */
1298    typedef struct {                                      // Table 2:112
1299        UINT32                    count;
1300        TPMS_PCR_SELECTION        pcrSelections[HASH_COUNT];
1301    } TPML_PCR_SELECTION;                                 /* Structure */
1302    typedef struct {                                      // Table 2:113
1303        UINT32                    count;
1304        TPMS_ALG_PROPERTY         algProperties[MAX_CAP_ALGS];
1305    } TPML_ALG_PROPERTY;                                  /* Structure */
1306    typedef struct {                                      // Table 2:114
1307        UINT32                        count;
1308        TPMS_TAGGED_PROPERTY          tpmProperty[MAX_TPM_PROPERTIES];
1309    } TPML_TAGGED_TPM_PROPERTY;                           /* Structure */
1310    typedef struct {                                      // Table 2:115
1311        UINT32                        count;
1312        TPMS_TAGGED_PCR_SELECT        pcrProperty[MAX_PCR_PROPERTIES];
1313    } TPML_TAGGED_PCR_PROPERTY;                           /* Structure */
```

```
1314    typedef struct {                                              // Table 2:116
1315        UINT32                  count;
1316        TPM_ECC_CURVE           eccCurves[MAX_ECC_CURVES];
1317    } TPML_ECC_CURVE;                                             /* Structure */
1318    typedef struct {                                              // Table 2:117
1319        UINT32                  count;
1320        TPMS_TAGGED_POLICY      policies[MAX_TAGGED_POLICIES];
1321    } TPML_TAGGED_POLICY;                                         /* Structure */
1322    typedef struct {                                              // Table 2:118
1323        UINT32                  count;
1324        TPMS_ACT_DATA           actData[MAX_ACT_DATA];
1325    } TPML_ACT_DATA;                                              /* Structure */
1326    typedef union {                                               // Table 2:119
1327        TPML_ALG_PROPERTY               algorithms;
1328        TPML_HANDLE                     handles;
1329        TPML_CCA                        command;
1330        TPML_CC                         ppCommands;
1331        TPML_CC                         auditCommands;
1332        TPML_PCR_SELECTION              assignedPCR;
1333        TPML_TAGGED_TPM_PROPERTY        tpmProperties;
1334        TPML_TAGGED_PCR_PROPERTY        pcrProperties;
1335    #if ALG_ECC
1336        TPML_ECC_CURVE                  eccCurves;
1337    #endif // ALG_ECC
1338        TPML_TAGGED_POLICY              authPolicies;
1339        TPML_ACT_DATA                   actData;
1340    } TPMU_CAPABILITIES;                                          /* Structure */
1341    typedef struct {                                              // Table 2:120
1342        TPM_CAP                 capability;
1343        TPMU_CAPABILITIES       data;
1344    } TPMS_CAPABILITY_DATA;                                       /* Structure */
1345    typedef struct {                                              // Table 2:121
1346        UINT64                  clock;
1347        UINT32                  resetCount;
1348        UINT32                  restartCount;
1349        TPMI_YES_NO             safe;
1350    } TPMS_CLOCK_INFO;                                            /* Structure */
1351    typedef struct {                                              // Table 2:122
1352        UINT64                  time;
1353        TPMS_CLOCK_INFO         clockInfo;
1354    } TPMS_TIME_INFO;                                             /* Structure */
1355    typedef struct {                                              // Table 2:123
1356        TPMS_TIME_INFO          time;
1357        UINT64                  firmwareVersion;
1358    } TPMS_TIME_ATTEST_INFO;                                      /* Structure */
1359    typedef struct {                                              // Table 2:124
1360        TPM2B_NAME              name;
1361        TPM2B_NAME              qualifiedName;
1362    } TPMS_CERTIFY_INFO;                                          /* Structure */
1363    typedef struct {                                              // Table 2:125
1364        TPML_PCR_SELECTION      pcrSelect;
1365        TPM2B_DIGEST            pcrDigest;
1366    } TPMS_QUOTE_INFO;                                            /* Structure */
1367    typedef struct {                                              // Table 2:126
1368        UINT64                  auditCounter;
1369        TPM_ALG_ID              digestAlg;
1370        TPM2B_DIGEST            auditDigest;
1371        TPM2B_DIGEST            commandDigest;
1372    } TPMS_COMMAND_AUDIT_INFO;                                    /* Structure */
1373    typedef struct {                                              // Table 2:127
1374        TPMI_YES_NO             exclusiveSession;
1375        TPM2B_DIGEST            sessionDigest;
1376    } TPMS_SESSION_AUDIT_INFO;                                    /* Structure */
1377    typedef struct {                                              // Table 2:128
1378        TPM2B_NAME              objectName;
1379        TPM2B_DIGEST            creationHash;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1380    } TPMS_CREATION_INFO;                                /* Structure */
1381    typedef struct {                                     // Table 2:129
1382        TPM2B_NAME                   indexName;
1383        UINT16                       offset;
1384        TPM2B_MAX_NV_BUFFER          nvContents;
1385    } TPMS_NV_CERTIFY_INFO;                              /* Structure */
1386    typedef struct {                                     // Table 2:130
1387        TPM2B_NAME               indexName;
1388        TPM2B_DIGEST             nvDigest;
1389    } TPMS_NV_DIGEST_CERTIFY_INFO;                       /* Structure */
1390    typedef TPM_ST               TPMI_ST_ATTEST;         // Table 2:131  /* Interface */
1391    typedef union {                                              // Table 2:132
1392        TPMS_CERTIFY_INFO                    certify;
1393        TPMS_CREATION_INFO                   creation;
1394        TPMS_QUOTE_INFO                      quote;
1395        TPMS_COMMAND_AUDIT_INFO              commandAudit;
1396        TPMS_SESSION_AUDIT_INFO              sessionAudit;
1397        TPMS_TIME_ATTEST_INFO                time;
1398        TPMS_NV_CERTIFY_INFO                 nv;
1399        TPMS_NV_DIGEST_CERTIFY_INFO          nvDigest;
1400    } TPMU_ATTEST;                                               /* Structure */
1401    typedef struct {                                     // Table 2:133
1402        TPM_GENERATED            magic;
1403        TPMI_ST_ATTEST           type;
1404        TPM2B_NAME               qualifiedSigner;
1405        TPM2B_DATA               extraData;
1406        TPMS_CLOCK_INFO          clockInfo;
1407        UINT64                   firmwareVersion;
1408        TPMU_ATTEST              attested;
1409    } TPMS_ATTEST;                                       /* Structure */
1410    typedef union {                                      // Table 2:134
1411        struct {
1412            UINT16               size;
1413            BYTE                 attestationData[sizeof(TPMS_ATTEST)];
1414        }            t;
1415        TPM2B        b;
1416    } TPM2B_ATTEST;                                      /* Structure */
1417    typedef struct {                                     // Table 2:135
1418        TPMI_SH_AUTH_SESSION         sessionHandle;
1419        TPM2B_NONCE                  nonce;
1420        TPMA_SESSION                 sessionAttributes;
1421        TPM2B_AUTH                   hmac;
1422    } TPMS_AUTH_COMMAND;                                 /* Structure */
1423    typedef struct {                                     // Table 2:136
1424        TPM2B_NONCE              nonce;
1425        TPMA_SESSION             sessionAttributes;
1426        TPM2B_AUTH               hmac;
1427    } TPMS_AUTH_RESPONSE;                                /* Structure */
1428    typedef TPM_KEY_BITS         TPMI_TDES_KEY_BITS;     // Table 2:137  /* Interface */
1429    typedef TPM_KEY_BITS         TPMI_AES_KEY_BITS;      // Table 2:137  /* Interface */
1430    typedef TPM_KEY_BITS         TPMI_SM4_KEY_BITS;      // Table 2:137  /* Interface */
1431    typedef TPM_KEY_BITS         TPMI_CAMELLIA_KEY_BITS; // Table 2:137  /* Interface */
1432    typedef union {                                      // Table 2:138
1433    #if ALG_TDES
1434        TPMI_TDES_KEY_BITS           tdes;
1435    #endif // ALG_TDES
1436    #if ALG_AES
1437        TPMI_AES_KEY_BITS            aes;
1438    #endif // ALG_AES
1439    #if ALG_SM4
1440        TPMI_SM4_KEY_BITS            sm4;
1441    #endif // ALG_SM4
1442    #if ALG_CAMELLIA
1443        TPMI_CAMELLIA_KEY_BITS       camellia;
1444    #endif // ALG_CAMELLIA
1445        TPM_KEY_BITS                 sym;
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
1446    #if ALG_XOR
1447        TPMI_ALG_HASH               xor;
1448    #endif // ALG_XOR
1449    } TPMU_SYM_KEY_BITS;                                      /* Structure */
1450    typedef union {                                           // Table 2:139
1451    #if ALG_TDES
1452        TPMI_ALG_SYM_MODE       tdes;
1453    #endif // ALG_TDES
1454    #if ALG_AES
1455        TPMI_ALG_SYM_MODE       aes;
1456    #endif // ALG_AES
1457    #if ALG_SM4
1458        TPMI_ALG_SYM_MODE       sm4;
1459    #endif // ALG_SM4
1460    #if ALG_CAMELLIA
1461        TPMI_ALG_SYM_MODE       camellia;
1462    #endif // ALG_CAMELLIA
1463        TPMI_ALG_SYM_MODE       sym;
1464    } TPMU_SYM_MODE;                                          /* Structure */
1465    typedef struct {                                          // Table 2:141
1466        TPMI_ALG_SYM            algorithm;
1467        TPMU_SYM_KEY_BITS       keyBits;
1468        TPMU_SYM_MODE           mode;
1469    } TPMT_SYM_DEF;                                           /* Structure */
1470    typedef struct {                                          // Table 2:142
1471        TPMI_ALG_SYM_OBJECT         algorithm;
1472        TPMU_SYM_KEY_BITS           keyBits;
1473        TPMU_SYM_MODE               mode;
1474    } TPMT_SYM_DEF_OBJECT;                                    /* Structure */
1475    typedef union {                                           // Table 2:143
1476        struct {
1477            UINT16              size;
1478            BYTE                buffer[MAX_SYM_KEY_BYTES];
1479        }            t;
1480        TPM2B        b;
1481    } TPM2B_SYM_KEY;                                          /* Structure */
1482    typedef struct {                                          // Table 2:144
1483        TPMT_SYM_DEF_OBJECT         sym;
1484    } TPMS_SYMCIPHER_PARMS;                                   /* Structure */
1485    typedef union {                                           // Table 2:145
1486        struct {
1487            UINT16              size;
1488            BYTE                buffer[LABEL_MAX_BUFFER];
1489        }            t;
1490        TPM2B        b;
1491    } TPM2B_LABEL;                                            /* Structure */
1492    typedef struct {                                          // Table 2:146
1493        TPM2B_LABEL             label;
1494        TPM2B_LABEL             context;
1495    } TPMS_DERIVE;                                            /* Structure */
1496    typedef union {                                           // Table 2:147
1497        struct {
1498            UINT16              size;
1499            BYTE                buffer[sizeof(TPMS_DERIVE)];
1500        }            t;
1501        TPM2B        b;
1502    } TPM2B_DERIVE;                                           /* Structure */
1503    typedef union {                                           // Table 2:148
1504        BYTE                    create[MAX_SYM_DATA];
1505        TPMS_DERIVE             derive;
1506    } TPMU_SENSITIVE_CREATE;                                  /* Structure */
1507    typedef union {                                           // Table 2:149
1508        struct {
1509            UINT16              size;
1510            BYTE                buffer[sizeof(TPMU_SENSITIVE_CREATE)];
1511        }            t;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1512        TPM2B          b;
1513    } TPM2B_SENSITIVE_DATA;                               /* Structure */
1514    typedef struct {                                      // Table 2:150
1515        TPM2B_AUTH                    userAuth;
1516        TPM2B_SENSITIVE_DATA          data;
1517    } TPMS_SENSITIVE_CREATE;                              /* Structure */
1518    typedef struct {                                      // Table 2:151
1519        UINT16                        size;
1520        TPMS_SENSITIVE_CREATE         sensitive;
1521    } TPM2B_SENSITIVE_CREATE;                             /* Structure */
1522    typedef struct {                                      // Table 2:152
1523        TPMI_ALG_HASH             hashAlg;
1524    } TPMS_SCHEME_HASH;                                   /* Structure */
1525    typedef struct {                                      // Table 2:153
1526        TPMI_ALG_HASH             hashAlg;
1527        UINT16                    count;
1528    } TPMS_SCHEME_ECDAA;                                  /* Structure */
1529    typedef TPM_ALG_ID            TPMI_ALG_KEYEDHASH_SCHEME;
```

Table 2:155 - Definition of Types for HMAC_SIG_SCHEME

```
1530    typedef TPMS_SCHEME_HASH      TPMS_SCHEME_HMAC;
1531    typedef struct {                                      // Table 2:156
1532        TPMI_ALG_HASH             hashAlg;
1533        TPMI_ALG_KDF              kdf;
1534    } TPMS_SCHEME_XOR;                                    /* Structure */
1535    typedef union {                                       // Table 2:157
1536    #if ALG_HMAC
1537        TPMS_SCHEME_HMAC          hmac;
1538    #endif // ALG_HMAC
1539    #if ALG_XOR
1540        TPMS_SCHEME_XOR           xor;
1541    #endif // ALG_XOR
1542    } TPMU_SCHEME_KEYEDHASH;                              /* Structure */
1543    typedef struct {                                      // Table 2:158
1544        TPMI_ALG_KEYEDHASH_SCHEME         scheme;
1545        TPMU_SCHEME_KEYEDHASH             details;
1546    } TPMT_KEYEDHASH_SCHEME;                              /* Structure */
```

Table 2:159 - Definition of Types for RSA Signature Schemes

```
1547    typedef TPMS_SCHEME_HASH      TPMS_SIG_SCHEME_RSASSA;
1548    typedef TPMS_SCHEME_HASH      TPMS_SIG_SCHEME_RSAPSS;
```

Table 2:160 - Definition of Types for ECC Signature Schemes

```
1549    typedef TPMS_SCHEME_HASH      TPMS_SIG_SCHEME_ECDSA;
1550    typedef TPMS_SCHEME_HASH      TPMS_SIG_SCHEME_SM2;
1551    typedef TPMS_SCHEME_HASH      TPMS_SIG_SCHEME_ECSCHNORR;
1552    typedef TPMS_SCHEME_ECDAA     TPMS_SIG_SCHEME_ECDAA;
1553    typedef union {                                       // Table 2:161
1554    #if ALG_ECC
1555        TPMS_SIG_SCHEME_ECDAA             ecdaa;
1556    #endif // ALG_ECC
1557    #if ALG_RSASSA
1558        TPMS_SIG_SCHEME_RSASSA            rsassa;
1559    #endif // ALG_RSASSA
1560    #if ALG_RSAPSS
1561        TPMS_SIG_SCHEME_RSAPSS            rsapss;
1562    #endif // ALG_RSAPSS
1563    #if ALG_ECDSA
1564        TPMS_SIG_SCHEME_ECDSA             ecdsa;
1565    #endif // ALG_ECDSA
1566    #if ALG_SM2
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
1567        TPMS_SIG_SCHEME_SM2                sm2;
1568    #endif // ALG_SM2
1569    #if ALG_ECSCHNORR
1570        TPMS_SIG_SCHEME_ECSCHNORR          ecschnorr;
1571    #endif // ALG_ECSCHNORR
1572    #if ALG_HMAC
1573        TPMS_SCHEME_HMAC                   hmac;
1574    #endif // ALG_HMAC
1575        TPMS_SCHEME_HASH                   any;
1576    } TPMU_SIG_SCHEME;                                          /* Structure */
1577    typedef struct {                                            // Table 2:162
1578        TPMI_ALG_SIG_SCHEME            scheme;
1579        TPMU_SIG_SCHEME                details;
1580    } TPMT_SIG_SCHEME;                                          /* Structure */
```

Table 2:163 - Definition of Types for Encryption Schemes

```
1581    typedef TPMS_SCHEME_HASH    TPMS_ENC_SCHEME_OAEP;
1582    typedef TPMS_EMPTY          TPMS_ENC_SCHEME_RSAES;
```

Table 2:164 - Definition of Types for ECC Key Exchange

```
1583    typedef TPMS_SCHEME_HASH    TPMS_KEY_SCHEME_ECDH;
1584    typedef TPMS_SCHEME_HASH    TPMS_KEY_SCHEME_ECMQV;
```

Table 2:165 - Definition of Types for KDF Schemes

```
1585    typedef TPMS_SCHEME_HASH    TPMS_SCHEME_MGF1;
1586    typedef TPMS_SCHEME_HASH    TPMS_SCHEME_KDF1_SP800_56A;
1587    typedef TPMS_SCHEME_HASH    TPMS_SCHEME_KDF2;
1588    typedef TPMS_SCHEME_HASH    TPMS_SCHEME_KDF1_SP800_108;
1589    typedef union {                                             // Table 2:166
1590    #if ALG_MGF1
1591        TPMS_SCHEME_MGF1                   mgf1;
1592    #endif // ALG_MGF1
1593    #if ALG_KDF1_SP800_56A
1594        TPMS_SCHEME_KDF1_SP800_56A         kdf1_sp800_56a;
1595    #endif // ALG_KDF1_SP800_56A
1596    #if ALG_KDF2
1597        TPMS_SCHEME_KDF2                   kdf2;
1598    #endif // ALG_KDF2
1599    #if ALG_KDF1_SP800_108
1600        TPMS_SCHEME_KDF1_SP800_108         kdf1_sp800_108;
1601    #endif // ALG_KDF1_SP800_108
1602    } TPMU_KDF_SCHEME;                                          /* Structure */
1603    typedef struct {                                            // Table 2:167
1604        TPMI_ALG_KDF            scheme;
1605        TPMU_KDF_SCHEME         details;
1606    } TPMT_KDF_SCHEME;                                          /* Structure */
1607    typedef TPM_ALG_ID          TPMI_ALG_ASYM_SCHEME;   // Table 2:168  /* Interface */
1608    typedef union {                                             // Table 2:169
1609    #if ALG_ECDH
1610        TPMS_KEY_SCHEME_ECDH               ecdh;
1611    #endif // ALG_ECDH
1612    #if ALG_ECMQV
1613        TPMS_KEY_SCHEME_ECMQV              ecmqv;
1614    #endif // ALG_ECMQV
1615    #if ALG_ECC
1616        TPMS_SIG_SCHEME_ECDAA              ecdaa;
1617    #endif // ALG_ECC
1618    #if ALG_RSASSA
1619        TPMS_SIG_SCHEME_RSASSA             rsassa;
1620    #endif // ALG_RSASSA
1621    #if ALG_RSAPSS
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1622        TPMS_SIG_SCHEME_RSAPSS          rsapss;
1623    #endif // ALG_RSAPSS
1624    #if ALG_ECDSA
1625        TPMS_SIG_SCHEME_ECDSA           ecdsa;
1626    #endif // ALG_ECDSA
1627    #if ALG_SM2
1628        TPMS_SIG_SCHEME_SM2             sm2;
1629    #endif // ALG_SM2
1630    #if ALG_ECSCHNORR
1631        TPMS_SIG_SCHEME_ECSCHNORR       ecschnorr;
1632    #endif // ALG_ECSCHNORR
1633    #if ALG_RSAES
1634        TPMS_ENC_SCHEME_RSAES           rsaes;
1635    #endif // ALG_RSAES
1636    #if ALG_OAEP
1637        TPMS_ENC_SCHEME_OAEP            oaep;
1638    #endif // ALG_OAEP
1639        TPMS_SCHEME_HASH                anySig;
1640    } TPMU_ASYM_SCHEME;                                 /* Structure */
1641    typedef struct {                                    // Table 2:170
1642        TPMI_ALG_ASYM_SCHEME        scheme;
1643        TPMU_ASYM_SCHEME            details;
1644    } TPMT_ASYM_SCHEME;                                 /* Structure */
1645    typedef TPM_ALG_ID          TPMI_ALG_RSA_SCHEME;    // Table 2:171  /* Interface */
1646    typedef struct {                                    // Table 2:172
1647        TPMI_ALG_RSA_SCHEME         scheme;
1648        TPMU_ASYM_SCHEME            details;
1649    } TPMT_RSA_SCHEME;                                  /* Structure */
1650    typedef TPM_ALG_ID          TPMI_ALG_RSA_DECRYPT;   // Table 2:173  /* Interface */
1651    typedef struct {                                    // Table 2:174
1652        TPMI_ALG_RSA_DECRYPT        scheme;
1653        TPMU_ASYM_SCHEME            details;
1654    } TPMT_RSA_DECRYPT;                                 /* Structure */
1655    typedef union {                                     // Table 2:175
1656        struct {
1657            UINT16              size;
1658            BYTE                buffer[MAX_RSA_KEY_BYTES];
1659        }           t;
1660        TPM2B       b;
1661    } TPM2B_PUBLIC_KEY_RSA;                             /* Structure */
1662    typedef TPM_KEY_BITS        TPMI_RSA_KEY_BITS;      // Table 2:176  /* Interface */
1663    typedef union {                                     // Table 2:177
1664        struct {
1665            UINT16              size;
1666            BYTE                buffer[RSA_PRIVATE_SIZE];
1667        }           t;
1668        TPM2B       b;
1669    } TPM2B_PRIVATE_KEY_RSA;                            /* Structure */
1670    typedef union {                                     // Table 2:178
1671        struct {
1672            UINT16              size;
1673            BYTE                buffer[MAX_ECC_KEY_BYTES];
1674        }           t;
1675        TPM2B       b;
1676    } TPM2B_ECC_PARAMETER;                              /* Structure */
1677    typedef struct {                                    // Table 2:179
1678        TPM2B_ECC_PARAMETER         x;
1679        TPM2B_ECC_PARAMETER         y;
1680    } TPMS_ECC_POINT;                                   /* Structure */
1681    typedef struct {                                    // Table 2:180
1682        UINT16                  size;
1683        TPMS_ECC_POINT          point;
1684    } TPM2B_ECC_POINT;                                  /* Structure */
1685    typedef TPM_ALG_ID          TPMI_ALG_ECC_SCHEME;    // Table 2:181  /* Interface */
1686    typedef TPM_ECC_CURVE       TPMI_ECC_CURVE;         // Table 2:182  /* Interface */
1687    typedef struct {                                    // Table 2:183
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
1688        TPMI_ALG_ECC_SCHEME          scheme;
1689        TPMU_ASYM_SCHEME             details;
1690    } TPMT_ECC_SCHEME;                                       /* Structure */
1691    typedef struct {                                         // Table 2:184
1692        TPM_ECC_CURVE                curveID;
1693        UINT16                       keySize;
1694        TPMT_KDF_SCHEME              kdf;
1695        TPMT_ECC_SCHEME              sign;
1696        TPM2B_ECC_PARAMETER          p;
1697        TPM2B_ECC_PARAMETER          a;
1698        TPM2B_ECC_PARAMETER          b;
1699        TPM2B_ECC_PARAMETER          gX;
1700        TPM2B_ECC_PARAMETER          gY;
1701        TPM2B_ECC_PARAMETER          n;
1702        TPM2B_ECC_PARAMETER          h;
1703    } TPMS_ALGORITHM_DETAIL_ECC;                             /* Structure */
1704    typedef struct {                                         // Table 2:185
1705        TPMI_ALG_HASH                hash;
1706        TPM2B_PUBLIC_KEY_RSA         sig;
1707    } TPMS_SIGNATURE_RSA;                                    /* Structure */
```

Table 2:186 - Definition of Types for Signature

```
1708    typedef TPMS_SIGNATURE_RSA  TPMS_SIGNATURE_RSASSA;
1709    typedef TPMS_SIGNATURE_RSA  TPMS_SIGNATURE_RSAPSS;
1710    typedef struct {                                         // Table 2:187
1711        TPMI_ALG_HASH                hash;
1712        TPM2B_ECC_PARAMETER          signatureR;
1713        TPM2B_ECC_PARAMETER          signatureS;
1714    } TPMS_SIGNATURE_ECC;                                    /* Structure */
```

Table 2:188 - Definition of Types for TPMS_SIGNATURE_ECC

```
1715    typedef TPMS_SIGNATURE_ECC  TPMS_SIGNATURE_ECDAA;
1716    typedef TPMS_SIGNATURE_ECC  TPMS_SIGNATURE_ECDSA;
1717    typedef TPMS_SIGNATURE_ECC  TPMS_SIGNATURE_SM2;
1718    typedef TPMS_SIGNATURE_ECC  TPMS_SIGNATURE_ECSCHNORR;
1719    typedef union {                                          // Table 2:189
1720    #if ALG_ECC
1721        TPMS_SIGNATURE_ECDAA             ecdaa;
1722    #endif // ALG_ECC
1723    #if ALG_RSA
1724        TPMS_SIGNATURE_RSASSA            rsassa;
1725    #endif // ALG_RSA
1726    #if ALG_RSA
1727        TPMS_SIGNATURE_RSAPSS            rsapss;
1728    #endif // ALG_RSA
1729    #if ALG_ECC
1730        TPMS_SIGNATURE_ECDSA             ecdsa;
1731    #endif // ALG_ECC
1732    #if ALG_ECC
1733        TPMS_SIGNATURE_SM2               sm2;
1734    #endif // ALG_ECC
1735    #if ALG_ECC
1736        TPMS_SIGNATURE_ECSCHNORR         ecschnorr;
1737    #endif // ALG_ECC
1738    #if ALG_HMAC
1739        TPMT_HA                          hmac;
1740    #endif // ALG_HMAC
1741        TPMS_SCHEME_HASH                 any;
1742    } TPMU_SIGNATURE;                                        /* Structure */
1743    typedef struct {                                         // Table 2:190
1744        TPMI_ALG_SIG_SCHEME          sigAlg;
1745        TPMU_SIGNATURE               signature;
1746    } TPMT_SIGNATURE;                                        /* Structure */
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1747   typedef union {                                          // Table 2:191
1748   #if ALG_ECC
1749       BYTE                        ecc[sizeof(TPMS_ECC_POINT)];
1750   #endif // ALG_ECC
1751   #if ALG_RSA
1752       BYTE                        rsa[MAX_RSA_KEY_BYTES];
1753   #endif // ALG_RSA
1754   #if ALG_SYMCIPHER
1755       BYTE                        symmetric[sizeof(TPM2B_DIGEST)];
1756   #endif // ALG_SYMCIPHER
1757   #if ALG_KEYEDHASH
1758       BYTE                        keyedHash[sizeof(TPM2B_DIGEST)];
1759   #endif // ALG_KEYEDHASH
1760   } TPMU_ENCRYPTED_SECRET;                                 /* Structure */
1761   typedef union {                                          // Table 2:192
1762       struct {
1763           UINT16                  size;
1764           BYTE                    secret[sizeof(TPMU_ENCRYPTED_SECRET)];
1765       }            t;
1766       TPM2B        b;
1767   } TPM2B_ENCRYPTED_SECRET;                                /* Structure */
1768   typedef TPM_ALG_ID          TPMI_ALG_PUBLIC;         // Table 2:193  /* Interface */
1769   typedef union {                                          // Table 2:194
1770   #if ALG_KEYEDHASH
1771       TPM2B_DIGEST                keyedHash;
1772   #endif // ALG_KEYEDHASH
1773   #if ALG_SYMCIPHER
1774       TPM2B_DIGEST                sym;
1775   #endif // ALG_SYMCIPHER
1776   #if ALG_RSA
1777       TPM2B_PUBLIC_KEY_RSA        rsa;
1778   #endif // ALG_RSA
1779   #if ALG_ECC
1780       TPMS_ECC_POINT              ecc;
1781   #endif // ALG_ECC
1782       TPMS_DERIVE                 derive;
1783   } TPMU_PUBLIC_ID;                                        /* Structure */
1784   typedef struct {                                         // Table 2:195
1785       TPMT_KEYEDHASH_SCHEME       scheme;
1786   } TPMS_KEYEDHASH_PARMS;                                  /* Structure */
1787   typedef struct {                                         // Table 2:196
1788       TPMT_SYM_DEF_OBJECT         symmetric;
1789       TPMT_ASYM_SCHEME            scheme;
1790   } TPMS_ASYM_PARMS;                                       /* Structure */
1791   typedef struct {                                         // Table 2:197
1792       TPMT_SYM_DEF_OBJECT         symmetric;
1793       TPMT_RSA_SCHEME             scheme;
1794       TPMI_RSA_KEY_BITS           keyBits;
1795       UINT32                      exponent;
1796   } TPMS_RSA_PARMS;                                        /* Structure */
1797   typedef struct {                                         // Table 2:198
1798       TPMT_SYM_DEF_OBJECT         symmetric;
1799       TPMT_ECC_SCHEME             scheme;
1800       TPMI_ECC_CURVE              curveID;
1801       TPMT_KDF_SCHEME             kdf;
1802   } TPMS_ECC_PARMS;                                        /* Structure */
1803   typedef union {                                          // Table 2:199
1804   #if ALG_KEYEDHASH
1805       TPMS_KEYEDHASH_PARMS        keyedHashDetail;
1806   #endif // ALG_KEYEDHASH
1807   #if ALG_SYMCIPHER
1808       TPMS_SYMCIPHER_PARMS        symDetail;
1809   #endif // ALG_SYMCIPHER
1810   #if ALG_RSA
1811       TPMS_RSA_PARMS              rsaDetail;
1812   #endif // ALG_RSA
```

 Copyright © TCG 2006-2020

```
1813  #if ALG_ECC
1814      TPMS_ECC_PARMS              eccDetail;
1815  #endif // ALG_ECC
1816      TPMS_ASYM_PARMS             asymDetail;
1817  } TPMU_PUBLIC_PARMS;                                      /* Structure */
1818  typedef struct {                                          // Table 2:200
1819      TPMI_ALG_PUBLIC         type;
1820      TPMU_PUBLIC_PARMS       parameters;
1821  } TPMT_PUBLIC_PARMS;                                      /* Structure */
1822  typedef struct {                                          // Table 2:201
1823      TPMI_ALG_PUBLIC         type;
1824      TPMI_ALG_HASH           nameAlg;
1825      TPMA_OBJECT             objectAttributes;
1826      TPM2B_DIGEST            authPolicy;
1827      TPMU_PUBLIC_PARMS       parameters;
1828      TPMU_PUBLIC_ID          unique;
1829  } TPMT_PUBLIC;                                            /* Structure */
1830  typedef struct {                                          // Table 2:202
1831      UINT16                  size;
1832      TPMT_PUBLIC             publicArea;
1833  } TPM2B_PUBLIC;                                           /* Structure */
1834  typedef union {                                           // Table 2:203
1835      struct {
1836          UINT16              size;
1837          BYTE                buffer[sizeof(TPMT_PUBLIC)];
1838      }            t;
1839      TPM2B        b;
1840  } TPM2B_TEMPLATE;                                         /* Structure */
1841  typedef union {                                           // Table 2:204
1842      struct {
1843          UINT16              size;
1844          BYTE                buffer[PRIVATE_VENDOR_SPECIFIC_BYTES];
1845      }            t;
1846      TPM2B        b;
1847  } TPM2B_PRIVATE_VENDOR_SPECIFIC;                          /* Structure */
1848  typedef union {                                           // Table 2:205
1849  #if ALG_RSA
1850      TPM2B_PRIVATE_KEY_RSA               rsa;
1851  #endif // ALG_RSA
1852  #if ALG_ECC
1853      TPM2B_ECC_PARAMETER                 ecc;
1854  #endif // ALG_ECC
1855  #if ALG_KEYEDHASH
1856      TPM2B_SENSITIVE_DATA                bits;
1857  #endif // ALG_KEYEDHASH
1858  #if ALG_SYMCIPHER
1859      TPM2B_SYM_KEY                       sym;
1860  #endif // ALG_SYMCIPHER
1861      TPM2B_PRIVATE_VENDOR_SPECIFIC       any;
1862  } TPMU_SENSITIVE_COMPOSITE;                               /* Structure */
1863  typedef struct {                                          // Table 2:206
1864      TPMI_ALG_PUBLIC                 sensitiveType;
1865      TPM2B_AUTH                      authValue;
1866      TPM2B_DIGEST                    seedValue;
1867      TPMU_SENSITIVE_COMPOSITE        sensitive;
1868  } TPMT_SENSITIVE;                                         /* Structure */
1869  typedef struct {                                          // Table 2:207
1870      UINT16                  size;
1871      TPMT_SENSITIVE          sensitiveArea;
1872  } TPM2B_SENSITIVE;                                        /* Structure */
1873  typedef struct {                                          // Table 2:208
1874      TPM2B_DIGEST            integrityOuter;
1875      TPM2B_DIGEST            integrityInner;
1876      TPM2B_SENSITIVE         sensitive;
1877  } _PRIVATE;                                               /* Structure */
1878  typedef union {                                           // Table 2:209
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1879     struct {
1880         UINT16                  size;
1881         BYTE                    buffer[sizeof(_PRIVATE)];
1882     }           t;
1883     TPM2B       b;
1884 } TPM2B_PRIVATE;                                        /* Structure */
1885 typedef struct {                                        // Table 2:210
1886     TPM2B_DIGEST            integrityHMAC;
1887     TPM2B_DIGEST            encIdentity;
1888 } TPMS_ID_OBJECT;                                       /* Structure */
1889 typedef union {                                         // Table 2:211
1890     struct {
1891         UINT16                  size;
1892         BYTE                    credential[sizeof(TPMS_ID_OBJECT)];
1893     }           t;
1894     TPM2B       b;
1895 } TPM2B_ID_OBJECT;                                      /* Structure */
1896 #define TYPE_OF_TPM_NV_INDEX    UINT32
1897 #define TPM_NV_INDEX_TO_UINT32(a)    (*((UINT32 *)&(a)))
1898 #define UINT32_TO_TPM_NV_INDEX(a)    (*((TPM_NV_INDEX *)&(a)))
1899 #define TPM_NV_INDEX_TO_BYTE_ARRAY(i, a)                                    \
1900             UINT32_TO_BYTE_ARRAY((TPM_NV_INDEX_TO_UINT32(i)), (a))
1901 #define BYTE_ARRAY_TO_TPM_NV_INDEX(i, a)                                    \
1902             { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPM_NV_INDEX(x); }
1903 #if USE_BIT_FIELD_STRUCTURES
1904 typedef struct TPM_NV_INDEX {                           // Table 2:212
1905     unsigned    index                   : 24;
1906     unsigned    RH_NV                   : 8;
1907 } TPM_NV_INDEX;                                         /* Bits */
```

This is the initializer for a TPM_NV_INDEX structure

```
1908 #define TPM_NV_INDEX_INITIALIZER(index, rh_nv) {index, rh_nv}
1909 #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:212 TPM_NV_INDEX using bit masking

```
1910 typedef UINT32                      TPM_NV_INDEX;
1911 #define TYPE_OF_TPM_NV_INDEX        UINT32
1912 #define TPM_NV_INDEX_index_SHIFT    0
1913 #define TPM_NV_INDEX_index          ((TPM_NV_INDEX)0xffffff << 0)
1914 #define TPM_NV_INDEX_RH_NV_SHIFT    24
1915 #define TPM_NV_INDEX_RH_NV          ((TPM_NV_INDEX)0xff << 24)
```

This is the initializer for a TPM_NV_INDEX bit array.

```
1916 #define TPM_NV_INDEX_INITIALIZER(index, rh_nv) {(index << 0) + (rh_nv << 24)}
1917 #endif // USE_BIT_FIELD_STRUCTURES
```

Table 2:213 - Definition of TPM_NT Constants

```
1918 typedef UINT32              TPM_NT;
1919 #define TYPE_OF_TPM_NT      UINT32
1920 #define TPM_NT_ORDINARY     (TPM_NT)(0x0)
1921 #define TPM_NT_COUNTER      (TPM_NT)(0x1)
1922 #define TPM_NT_BITS         (TPM_NT)(0x2)
1923 #define TPM_NT_EXTEND       (TPM_NT)(0x4)
1924 #define TPM_NT_PIN_FAIL     (TPM_NT)(0x8)
1925 #define TPM_NT_PIN_PASS     (TPM_NT)(0x9)
1926 typedef struct {                                        // Table 2:214
1927     UINT32                  pinCount;
1928     UINT32                  pinLimit;
1929 } TPMS_NV_PIN_COUNTER_PARAMETERS;                       /* Structure */
1930 #define TYPE_OF_TPMA_NV     UINT32
```

```
1931    #define TPMA_NV_TO_UINT32(a)      (*((UINT32 *)&(a)))
1932    #define UINT32_TO_TPMA_NV(a)      (*((TPMA_NV *)&(a)))
1933    #define TPMA_NV_TO_BYTE_ARRAY(i, a)                                                  \
1934                UINT32_TO_BYTE_ARRAY((TPMA_NV_TO_UINT32(i)), (a))
1935    #define BYTE_ARRAY_TO_TPMA_NV(i, a)                                                  \
1936                { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_NV(x); }
1937    #if USE_BIT_FIELD_STRUCTURES
1938    typedef struct TPMA_NV {                                    // Table 2:215
1939        unsigned    PPWRITE                 : 1;
1940        unsigned    OWNERWRITE              : 1;
1941        unsigned    AUTHWRITE               : 1;
1942        unsigned    POLICYWRITE             : 1;
1943        unsigned    TPM_NT                  : 4;
1944        unsigned    Reserved_bits_at_8      : 2;
1945        unsigned    POLICY_DELETE           : 1;
1946        unsigned    WRITELOCKED             : 1;
1947        unsigned    WRITEALL                : 1;
1948        unsigned    WRITEDEFINE             : 1;
1949        unsigned    WRITE_STCLEAR           : 1;
1950        unsigned    GLOBALLOCK              : 1;
1951        unsigned    PPREAD                  : 1;
1952        unsigned    OWNERREAD               : 1;
1953        unsigned    AUTHREAD                : 1;
1954        unsigned    POLICYREAD              : 1;
1955        unsigned    Reserved_bits_at_20     : 5;
1956        unsigned    NO_DA                   : 1;
1957        unsigned    ORDERLY                 : 1;
1958        unsigned    CLEAR_STCLEAR           : 1;
1959        unsigned    READLOCKED              : 1;
1960        unsigned    WRITTEN                 : 1;
1961        unsigned    PLATFORMCREATE          : 1;
1962        unsigned    READ_STCLEAR            : 1;
1963    } TPMA_NV;                                                  /* Bits */
```

This is the initializer for a TPMA_NV structure

```
1964    #define TPMA_NV_INITIALIZER(                                                         \
1965                 ppwrite,        ownerwrite,      authwrite,       policywrite,          \
1966                 tpm_nt,         bits_at_8,       policy_delete,   writelocked,          \
1967                 writeall,       writedefine,     write_stclear,   globallock,           \
1968                 ppread,         ownerread,       authread,        policyread,           \
1969                 bits_at_20,     no_da,           orderly,         clear_stclear,        \
1970                 readlocked,     written,         platformcreate,  read_stclear)         \
1971                {ppwrite,        ownerwrite,      authwrite,       policywrite,          \
1972                 tpm_nt,         bits_at_8,       policy_delete,   writelocked,          \
1973                 writeall,       writedefine,     write_stclear,   globallock,           \
1974                 ppread,         ownerread,       authread,        policyread,           \
1975                 bits_at_20,     no_da,           orderly,         clear_stclear,        \
1976                 readlocked,     written,         platformcreate,  read_stclear}
1977    #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:215 TPMA_NV using bit masking

```
1978    typedef UINT32                  TPMA_NV;
1979    #define TYPE_OF_TPMA_NV         UINT32
1980    #define TPMA_NV_PPWRITE          ((TPMA_NV)1 << 0)
1981    #define TPMA_NV_OWNERWRITE       ((TPMA_NV)1 << 1)
1982    #define TPMA_NV_AUTHWRITE        ((TPMA_NV)1 << 2)
1983    #define TPMA_NV_POLICYWRITE      ((TPMA_NV)1 << 3)
1984    #define TPMA_NV_TPM_NT_SHIFT     4
1985    #define TPMA_NV_TPM_NT           ((TPMA_NV)0xf << 4)
1986    #define TPMA_NV_POLICY_DELETE    ((TPMA_NV)1 << 10)
1987    #define TPMA_NV_WRITELOCKED      ((TPMA_NV)1 << 11)
1988    #define TPMA_NV_WRITEALL         ((TPMA_NV)1 << 12)
1989    #define TPMA_NV_WRITEDEFINE      ((TPMA_NV)1 << 13)
```

```
1990 #define TPMA_NV_WRITE_STCLEAR   ((TPMA_NV)1 << 14)
1991 #define TPMA_NV_GLOBALLOCK      ((TPMA_NV)1 << 15)
1992 #define TPMA_NV_PPREAD          ((TPMA_NV)1 << 16)
1993 #define TPMA_NV_OWNERREAD       ((TPMA_NV)1 << 17)
1994 #define TPMA_NV_AUTHREAD        ((TPMA_NV)1 << 18)
1995 #define TPMA_NV_POLICYREAD      ((TPMA_NV)1 << 19)
1996 #define TPMA_NV_NO_DA           ((TPMA_NV)1 << 25)
1997 #define TPMA_NV_ORDERLY         ((TPMA_NV)1 << 26)
1998 #define TPMA_NV_CLEAR_STCLEAR   ((TPMA_NV)1 << 27)
1999 #define TPMA_NV_READLOCKED      ((TPMA_NV)1 << 28)
2000 #define TPMA_NV_WRITTEN         ((TPMA_NV)1 << 29)
2001 #define TPMA_NV_PLATFORMCREATE  ((TPMA_NV)1 << 30)
2002 #define TPMA_NV_READ_STCLEAR    ((TPMA_NV)1 << 31)
```

This is the initializer for a TPMA_NV bit array.

```
2003 #define TPMA_NV_INITIALIZER(                                                      \
2004              ppwrite,        ownerwrite,     authwrite,      policywrite,         \
2005              tpm_nt,         bits_at_8,      policy_delete,  writelocked,         \
2006              writeall,       writedefine,    write_stclear,  globallock,          \
2007              ppread,         ownerread,      authread,       policyread,          \
2008              bits_at_20,     no_da,          orderly,        clear_stclear,       \
2009              readlocked,     written,        platformcreate, read_stclear)        \
2010             {(ppwrite << 0)          + (ownerwrite << 1)      +                  \
2011              (authwrite << 2)        + (policywrite << 3)     +                  \
2012              (tpm_nt << 4)           + (policy_delete << 10)  +                  \
2013              (writelocked << 11)     + (writeall << 12)       +                  \
2014              (writedefine << 13)     + (write_stclear << 14)  +                  \
2015              (globallock << 15)      + (ppread << 16)         +                  \
2016              (ownerread << 17)       + (authread << 18)       +                  \
2017              (policyread << 19)      + (no_da << 25)          +                  \
2018              (orderly << 26)         + (clear_stclear << 27)  +                  \
2019              (readlocked << 28)      + (written << 29)        +                  \
2020              (platformcreate << 30) + (read_stclear << 31)}
2021 #endif // USE_BIT_FIELD_STRUCTURES
2022 typedef struct {                                          // Table 2:216
2023     TPMI_RH_NV_INDEX        nvIndex;
2024     TPMI_ALG_HASH           nameAlg;
2025     TPMA_NV                 attributes;
2026     TPM2B_DIGEST            authPolicy;
2027     UINT16                  dataSize;
2028 } TPMS_NV_PUBLIC;                                         /* Structure */
2029 typedef struct {                                          // Table 2:217
2030     UINT16                  size;
2031     TPMS_NV_PUBLIC          nvPublic;
2032 } TPM2B_NV_PUBLIC;                                        /* Structure */
2033 typedef union {                                           // Table 2:218
2034     struct {
2035         UINT16              size;
2036         BYTE                buffer[MAX_CONTEXT_SIZE];
2037     }           t;
2038     TPM2B       b;
2039 } TPM2B_CONTEXT_SENSITIVE;                                /* Structure */
2040 typedef struct {                                          // Table 2:219
2041     TPM2B_DIGEST                    integrity;
2042     TPM2B_CONTEXT_SENSITIVE         encrypted;
2043 } TPMS_CONTEXT_DATA;                                      /* Structure */
2044 typedef union {                                           // Table 2:220
2045     struct {
2046         UINT16              size;
2047         BYTE                buffer[sizeof(TPMS_CONTEXT_DATA)];
2048     }           t;
2049     TPM2B       b;
2050 } TPM2B_CONTEXT_DATA;                                     /* Structure */
2051 typedef struct {                                          // Table 2:221
```

 Copyright © TCG 2006-2020

```
2052        UINT64                  sequence;
2053        TPMI_DH_SAVED           savedHandle;
2054        TPMI_RH_HIERARCHY       hierarchy;
2055        TPM2B_CONTEXT_DATA      contextBlob;
2056    } TPMS_CONTEXT;                                         /* Structure */
2057    typedef struct {                                        // Table 2:223
2058        TPML_PCR_SELECTION      pcrSelect;
2059        TPM2B_DIGEST            pcrDigest;
2060        TPMA_LOCALITY           locality;
2061        TPM_ALG_ID              parentNameAlg;
2062        TPM2B_NAME              parentName;
2063        TPM2B_NAME              parentQualifiedName;
2064        TPM2B_DATA              outsideInfo;
2065    } TPMS_CREATION_DATA;                                   /* Structure */
2066    typedef struct {                                        // Table 2:224
2067        UINT16                  size;
2068        TPMS_CREATION_DATA      creationData;
2069    } TPM2B_CREATION_DATA;                                  /* Structure */
```

Table 2:225 - Definition of TPM_AT Constants

```
2070    typedef UINT32              TPM_AT;
2071    #define TYPE_OF_TPM_AT      UINT32
2072    #define TPM_AT_ANY          (TPM_AT)(0x00000000)
2073    #define TPM_AT_ERROR        (TPM_AT)(0x00000001)
2074    #define TPM_AT_PV1          (TPM_AT)(0x00000002)
2075    #define TPM_AT_VEND         (TPM_AT)(0x80000000)
```

Table 2:226 - Definition of TPM_AE Constants

```
2076    typedef UINT32              TPM_AE;
2077    #define TYPE_OF_TPM_AE      UINT32
2078    #define TPM_AE_NONE         (TPM_AE)(0x00000000)
2079    typedef struct {                                        // Table 2:227
2080        TPM_AT                  tag;
2081        UINT32                  data;
2082    } TPMS_AC_OUTPUT;                                       /* Structure */
2083    typedef struct {                                        // Table 2:228
2084        UINT32                  count;
2085        TPMS_AC_OUTPUT          acCapabilities[MAX_AC_CAPABILITIES];
2086    } TPML_AC_CAPABILITIES;                                 /* Structure */
2087    #endif // _TPM_TYPES_H_
```

5.20 VendorString.h

```
1   #ifndef      _VENDOR_STRING_H
2   #define      _VENDOR_STRING_H
```

Define up to 4-byte values for MANUFACTURER. This value defines the response for TPM_PT_MANUFACTURER in TPM2_GetCapability(). The following line should be un-commented and a vendor specific string should be provided here.

```
3   #define     MANUFACTURER     "MSFT"
```

The following #if macro may be deleted after a proper MANUFACTURER is provided.

```
4   #ifndef MANUFACTURER
5   #error MANUFACTURER is not provided. \
6   Please modify include/VendorString.h to provide a specific \
7   manufacturer name.
8   #endif
```

Define up to 4, 4-byte values. The values must each be 4 bytes long and the last value used may contain trailing zeros. These values define the response for TPM_PT_VENDOR_STRING_(1-4) in TPM2_GetCapability(). The following line should be un-commented and a vendor specific string should be provided here. The vendor strings 2-4 may also be defined as appropriate.

```
9   #define        VENDOR_STRING_1        "xCG "
10  #define        VENDOR_STRING_2        "fTPM"
11  // #define        VENDOR_STRING_3
12  // #define        VENDOR_STRING_4
```

The following #if macro may be deleted after a proper VENDOR_STRING_1 is provided.

```
13  #ifndef VENDOR_STRING_1
14  #error VENDOR_STRING_1 is not provided. \
15  Please modify include/VendorString.h to provide a vendor-specific string.
16  #endif
```

the more significant 32-bits of a vendor-specific value indicating the version of the firmware The following line should be un-commented and a vendor specific firmware V1 should be provided here. The FIRMWARE_V2 may also be defined as appropriate.

```
17  #define    FIRMWARE_V1          (0x20170619)
```

the less significant 32-bits of a vendor-specific value indicating the version of the firmware

```
18  #define    FIRMWARE_V2          (0x00163636)
```

The following #if macro may be deleted after a proper FIRMWARE_V1 is provided.

```
19  #ifndef FIRMWARE_V1
20  #error  FIRMWARE_V1 is not provided. \
21  Please modify include/VendorString.h to provide a vendor-specific firmware \
22  version
23  #endif
24  #endif
```

 Copyright © TCG 2006-2020

5.21 swap.h

```
1  #ifndef _SWAP_H
2  #define _SWAP_H
3  #if LITTLE_ENDIAN_TPM
4  #define TO_BIG_ENDIAN_UINT16(i)      REVERSE_ENDIAN_16(i)
5  #define FROM_BIG_ENDIAN_UINT16(i)    REVERSE_ENDIAN_16(i)
6  #define TO_BIG_ENDIAN_UINT32(i)      REVERSE_ENDIAN_32(i)
7  #define FROM_BIG_ENDIAN_UINT32(i)    REVERSE_ENDIAN_32(i)
8  #define TO_BIG_ENDIAN_UINT64(i)      REVERSE_ENDIAN_64(i)
9  #define FROM_BIG_ENDIAN_UINT64(i)    REVERSE_ENDIAN_64(i)
10 #else
11 #define TO_BIG_ENDIAN_UINT16(i)      (i)
12 #define FROM_BIG_ENDIAN_UINT16(i)    (i)
13 #define TO_BIG_ENDIAN_UINT32(i)      (i)
14 #define FROM_BIG_ENDIAN_UINT32(i)    (i)
15 #define TO_BIG_ENDIAN_UINT64(i)      (i)
16 #define FROM_BIG_ENDIAN_UINT64(i)    (i)
17 #endif
18 #if   AUTO_ALIGN == NO
```

The aggregation macros for machines that do not allow unaligned access or for little-endian machines. Aggregate bytes into an UINT

```
19 #define BYTE_ARRAY_TO_UINT8(b)   (uint8_t)((b)[0])
20 #define BYTE_ARRAY_TO_UINT16(b) ByteArrayToUint16((BYTE *)(b))
21 #define BYTE_ARRAY_TO_UINT32(b) ByteArrayToUint32((BYTE *)(b))
22 #define BYTE_ARRAY_TO_UINT64(b) ByteArrayToUint64((BYTE *)(b))
23 #define UINT8_TO_BYTE_ARRAY(i, b)  ((b)[0] = (uint8_t)(i))
24 #define UINT16_TO_BYTE_ARRAY(i, b)  Uint16ToByteArray((i), (BYTE *)(b))
25 #define UINT32_TO_BYTE_ARRAY(i, b)  Uint32ToByteArray((i), (BYTE *)(b))
26 #define UINT64_TO_BYTE_ARRAY(i, b)  Uint64ToByteArray((i), (BYTE *)(b))
27 #else // AUTO_ALIGN
28 #if BIG_ENDIAN_TPM
```

the big-endian macros for machines that allow unaligned memory access Aggregate a byte array into a UINT

```
29 #define BYTE_ARRAY_TO_UINT8(b)         *((uint8_t  *)(b))
30 #define BYTE_ARRAY_TO_UINT16(b)        *((uint16_t *)(b))
31 #define BYTE_ARRAY_TO_UINT32(b)        *((uint32_t *)(b))
32 #define BYTE_ARRAY_TO_UINT64(b)        *((uint64_t *)(b))
```

Disaggregate a UINT into a byte array

```
33 #define UINT8_TO_BYTE_ARRAY(i, b)    {*((uint8_t  *)(b)) = (i);}
34 #define UINT16_TO_BYTE_ARRAY(i, b)   {*((uint16_t *)(b)) = (i);}
35 #define UINT32_TO_BYTE_ARRAY(i, b)   {*((uint32_t *)(b)) = (i);}
36 #define UINT64_TO_BYTE_ARRAY(i, b)   {*((uint64_t *)(b)) = (i);}
37 #else
```

the little endian macros for machines that allow unaligned memory access the big-endian macros for machines that allow unaligned memory access Aggregate a byte array into a UINT

```
38 #define BYTE_ARRAY_TO_UINT8(b)         *((uint8_t  *)(b))
39 #define BYTE_ARRAY_TO_UINT16(b)        REVERSE_ENDIAN_16(*((uint16_t *)(b)))
40 #define BYTE_ARRAY_TO_UINT32(b)        REVERSE_ENDIAN_32(*((uint32_t *)(b)))
41 #define BYTE_ARRAY_TO_UINT64(b)        REVERSE_ENDIAN_64(*((uint64_t *)(b)))
```

Disaggregate a UINT into a byte array

```
42 #define UINT8_TO_BYTE_ARRAY(i, b)    {*((uint8_t  *)(b)) = (i);}
```


 Copyright © TCG 2006-2020

```
43    #define UINT16_TO_BYTE_ARRAY(i, b)  {*((uint16_t *)(b)) = REVERSE_ENDIAN_16(i);}
44    #define UINT32_TO_BYTE_ARRAY(i, b)  {*((uint32_t *)(b)) = REVERSE_ENDIAN_32(i);}
45    #define UINT64_TO_BYTE_ARRAY(i, b)  {*((uint64_t *)(b)) = REVERSE_ENDIAN_64(i);}
46    #endif   // BIG_ENDIAN_TPM
47    #endif  // AUTO_ALIGN == NO
48    #endif  // _SWAP_H
```

 Copyright © TCG 2006-2020

5.22 ACT.h

```
1  #ifndef _ACT_H_
2  #define _ACT_H_
3  #include "TpmProfile.h"
4  #if !(defined RH_ACT_0) || (RH_ACT_0 != YES)
5  #   undef   RH_ACT_0
6  #   define  RH_ACT_0 NO
7  #   define IF_ACT_0_IMPLEMENTED(op)
8  #else
9  #   define IF_ACT_0_IMPLEMENTED(op) op(0)
10 #endif
11 #if !(defined RH_ACT_1) || (RH_ACT_1 != YES)
12 #   undef   RH_ACT_1
13 #   define  RH_ACT_1 NO
14 #   define IF_ACT_1_IMPLEMENTED(op)
15 #else
16 #   define IF_ACT_1_IMPLEMENTED(op) op(1)
17 #endif
18 #if !(defined RH_ACT_2) || (RH_ACT_2 != YES)
19 #   undef   RH_ACT_2
20 #   define  RH_ACT_2 NO
21 #   define IF_ACT_2_IMPLEMENTED(op)
22 #else
23 #   define IF_ACT_2_IMPLEMENTED(op) op(2)
24 #endif
25 #if !(defined RH_ACT_3) || (RH_ACT_3 != YES)
26 #   undef   RH_ACT_3
27 #   define  RH_ACT_3 NO
28 #   define IF_ACT_3_IMPLEMENTED(op)
29 #else
30 #   define IF_ACT_3_IMPLEMENTED(op) op(3)
31 #endif
32 #if !(defined RH_ACT_4) || (RH_ACT_4 != YES)
33 #   undef   RH_ACT_4
34 #   define  RH_ACT_4 NO
35 #   define IF_ACT_4_IMPLEMENTED(op)
36 #else
37 #   define IF_ACT_4_IMPLEMENTED(op) op(4)
38 #endif
39 #if !(defined RH_ACT_5) || (RH_ACT_5 != YES)
40 #   undef   RH_ACT_5
41 #   define  RH_ACT_5 NO
42 #   define IF_ACT_5_IMPLEMENTED(op)
43 #else
44 #   define IF_ACT_5_IMPLEMENTED(op) op(5)
45 #endif
46 #if !(defined RH_ACT_6) || (RH_ACT_6 != YES)
47 #   undef   RH_ACT_6
48 #   define  RH_ACT_6 NO
49 #   define IF_ACT_6_IMPLEMENTED(op)
50 #else
51 #   define IF_ACT_6_IMPLEMENTED(op) op(6)
52 #endif
53 #if !(defined RH_ACT_7) || (RH_ACT_7 != YES)
54 #   undef   RH_ACT_7
55 #   define  RH_ACT_7 NO
56 #   define IF_ACT_7_IMPLEMENTED(op)
57 #else
58 #   define IF_ACT_7_IMPLEMENTED(op) op(7)
59 #endif
60 #if !(defined RH_ACT_8) || (RH_ACT_8 != YES)
61 #   undef   RH_ACT_8
62 #   define  RH_ACT_8 NO
63 #   define IF_ACT_8_IMPLEMENTED(op)
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
64  #else
65  #   define IF_ACT_8_IMPLEMENTED(op) op(8)
66  #endif
67  #if !(defined RH_ACT_9) || (RH_ACT_9 != YES)
68  #   undef    RH_ACT_9
69  #   define  RH_ACT_9 NO
70  #   define IF_ACT_9_IMPLEMENTED(op)
71  #else
72  #   define IF_ACT_9_IMPLEMENTED(op) op(9)
73  #endif
74  #if !(defined RH_ACT_A) || (RH_ACT_A != YES)
75  #   undef    RH_ACT_A
76  #   define  RH_ACT_A NO
77  #   define IF_ACT_A_IMPLEMENTED(op)
78  #else
79  #   define IF_ACT_A_IMPLEMENTED(op) op(A)
80  #endif
81  #if !(defined RH_ACT_B) || (RH_ACT_B != YES)
82  #   undef    RH_ACT_B
83  #   define  RH_ACT_B NO
84  #   define IF_ACT_B_IMPLEMENTED(op)
85  #else
86  #   define IF_ACT_B_IMPLEMENTED(op) op(B)
87  #endif
88  #if !(defined RH_ACT_C) || (RH_ACT_C != YES)
89  #   undef    RH_ACT_C
90  #   define  RH_ACT_C NO
91  #   define IF_ACT_C_IMPLEMENTED(op)
92  #else
93  #   define IF_ACT_C_IMPLEMENTED(op) op(C)
94  #endif
95  #if !(defined RH_ACT_D) || (RH_ACT_D != YES)
96  #   undef    RH_ACT_D
97  #   define  RH_ACT_D NO
98  #   define IF_ACT_D_IMPLEMENTED(op)
99  #else
100 #   define IF_ACT_D_IMPLEMENTED(op) op(D)
101 #endif
102 #if !(defined RH_ACT_E) || (RH_ACT_E != YES)
103 #   undef    RH_ACT_E
104 #   define  RH_ACT_E NO
105 #   define IF_ACT_E_IMPLEMENTED(op)
106 #else
107 #   define IF_ACT_E_IMPLEMENTED(op) op(E)
108 #endif
109 #if !(defined RH_ACT_F) || (RH_ACT_F != YES)
110 #   undef    RH_ACT_F
111 #   define  RH_ACT_F NO
112 #   define IF_ACT_F_IMPLEMENTED(op)
113 #else
114 #   define IF_ACT_F_IMPLEMENTED(op) op(F)
115 #endif
116 #ifndef TPM_RH_ACT_0
117 #error Need numeric definition for TPM_RH_ACT_0
118 #endif
119 #ifndef TPM_RH_ACT_1
120 #   define TPM_RH_ACT_1     (TPM_RH_ACT_0 + 1)
121 #endif
122 #ifndef TPM_RH_ACT_2
123 #   define TPM_RH_ACT_2     (TPM_RH_ACT_0 + 2)
124 #endif
125 #ifndef TPM_RH_ACT_3
126 #   define TPM_RH_ACT_3     (TPM_RH_ACT_0 + 3)
127 #endif
128 #ifndef TPM_RH_ACT_4
129 #   define TPM_RH_ACT_4     (TPM_RH_ACT_0 + 4)
```

```
130 #endif
131 #ifndef TPM_RH_ACT_5
132 #   define TPM_RH_ACT_5     (TPM_RH_ACT_0 + 5)
133 #endif
134 #ifndef TPM_RH_ACT_6
135 #   define TPM_RH_ACT_6     (TPM_RH_ACT_0 + 6)
136 #endif
137 #ifndef TPM_RH_ACT_7
138 #   define TPM_RH_ACT_7     (TPM_RH_ACT_0 + 7)
139 #endif
140 #ifndef TPM_RH_ACT_8
141 #   define TPM_RH_ACT_8     (TPM_RH_ACT_0 + 8)
142 #endif
143 #ifndef TPM_RH_ACT_9
144 #   define TPM_RH_ACT_9     (TPM_RH_ACT_0 + 9)
145 #endif
146 #ifndef TPM_RH_ACT_A
147 #   define TPM_RH_ACT_A     (TPM_RH_ACT_0 + 0xA)
148 #endif
149 #ifndef TPM_RH_ACT_B
150 #   define TPM_RH_ACT_B     (TPM_RH_ACT_0 + 0xB)
151 #endif
152 #ifndef TPM_RH_ACT_C
153 #   define TPM_RH_ACT_C     (TPM_RH_ACT_0 + 0xC)
154 #endif
155 #ifndef TPM_RH_ACT_D
156 #   define TPM_RH_ACT_D     (TPM_RH_ACT_0 + 0xD)
157 #endif
158 #ifndef TPM_RH_ACT_E
159 #   define TPM_RH_ACT_E     (TPM_RH_ACT_0 + 0xE)
160 #endif
161 #ifndef TPM_RH_ACT_F
162 #   define TPM_RH_ACT_F     (TPM_RH_ACT_0 + 0xF)
163 #endif
164 #define FOR_EACH_ACT(op)                                                    \
165     IF_ACT_0_IMPLEMENTED(op)                                                \
166     IF_ACT_1_IMPLEMENTED(op)                                                \
167     IF_ACT_2_IMPLEMENTED(op)                                                \
168     IF_ACT_3_IMPLEMENTED(op)                                                \
169     IF_ACT_4_IMPLEMENTED(op)                                                \
170     IF_ACT_5_IMPLEMENTED(op)                                                \
171     IF_ACT_6_IMPLEMENTED(op)                                                \
172     IF_ACT_7_IMPLEMENTED(op)                                                \
173     IF_ACT_8_IMPLEMENTED(op)                                                \
174     IF_ACT_9_IMPLEMENTED(op)                                                \
175     IF_ACT_A_IMPLEMENTED(op)                                                \
176     IF_ACT_B_IMPLEMENTED(op)                                                \
177     IF_ACT_C_IMPLEMENTED(op)                                                \
178     IF_ACT_D_IMPLEMENTED(op)                                                \
179     IF_ACT_E_IMPLEMENTED(op)                                                \
180     IF_ACT_F_IMPLEMENTED(op)
```

This is the mask for ACT that are implemented

```
181 //#define ACT_MASK(N)       | (1 << 0x##N)
182 //#define ACT_IMPLEMENTED_MASK     (0 FOR_EACH_ACT(ACT_MASK))
183 #define CASE_ACT_HANDLE(N)      case TPM_RH_ACT_##N:
184 #define CASE_ACT_NUMBER(N)      case 0x##N:
185 typedef struct ACT_STATE
186 {
187     UINT32          remaining;
188     TPM_ALG_ID      hashAlg;
189     TPM2B_DIGEST    authPolicy;
190 } ACT_STATE, *P_ACT_STATE;
191 #endif // _ACT_H_
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

6 Main

6.1 Introduction

The files in this section are the main processing blocks for the TPM. ExecuteCommand.c contains the entry point into the TPM code and the parsing of the command header. SessionProcess.c handles the parsing of the session area and the authorization checks, and CommandDispatch.c does the parameter unmarshaling and command dispatch.

6.2 ExecCommand.c

6.2.1 Introduction

This file contains the entry function ExecuteCommand() which provides the main control flow for TPM command execution.

6.2.2 Includes

```
1   #include "Tpm.h"
2   #include "ExecCommand_fp.h"
```

Uncomment this next #include if doing static command/response buffer sizing

```
3   // #include "CommandResponseSizes_fp.h"
```

6.2.3 ExecuteCommand()

The function performs the following steps.

 Copyright © TCG 2006-2020

a) Parses the command header from input buffer.

b) Calls ParseHandleBuffer() to parse the handle area of the command.

c) Validates that each of the handles references a loaded entity.

d) Calls ParseSessionBuffer() () to:

   1) unmarshal and parse the session area;

   2) check the authorizations; and

   3) when necessary, decrypt a parameter.

e) Calls CommandDispatcher() to:

   1) unmarshal the command parameters from the command buffer;

   2) call the routine that performs the command actions; and

   3) marshal the responses into the response buffer.

f) If any error occurs in any of the steps above create the error response and return.

g) Calls BuildResponseSessions() to:

   1) when necessary, encrypt a parameter

   2) build the response authorization sessions

   3) update the audit sessions and nonces

h) Calls BuildResponseHeader() to complete the construction of the response.

responseSize is set by the caller to the maximum number of bytes available in the output buffer. ExecuteCommand() will adjust the value and return the number of bytes placed in the buffer.

response is also set by the caller to indicate the buffer into which ExecuteCommand() is to place the response.

request and *response* may point to the same buffer

NOTE: As of February, 2016, the failure processing has been moved to the platform-specific code. When the TPM code encounters an unrecoverable failure, it will SET *g_inFailureMode* and call _plat__Fail(). That function should not return but may call ExecuteCommand().

```
4   LIB_EXPORT void
5   ExecuteCommand(
6       uint32_t          requestSize,    // IN: command buffer size
7       unsigned char    *request,        // IN: command buffer
8       uint32_t         *responseSize,   // IN/OUT: response buffer size
9       unsigned char    **response       // IN/OUT: response buffer
10      )
11  {
12      // Command local variables
13      UINT32                commandSize;
14      COMMAND               command;
15
16      // Response local variables
17      UINT32                maxResponse = *responseSize;
18      TPM_RC                result;            // return code for the command
19
20  // This next function call is used in development to size the command and response
21  // buffers. The values printed are the sizes of the internal structures and
22  // not the sizes of the canonical forms of the command response structures. Also,
23  // the sizes do not include the tag, command.code, requestSize, or the authorization
24  // fields.
25  //CommandResponseSizes();
26      // Set flags for NV access state. This should happen before any other
27      // operation that may require a NV write. Note, that this needs to be done
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
28      // even when in failure mode. Otherwise, g_updateNV would stay SET while in
29      // Failure mode and the NV would be written on each call.
30      g_updateNV = UT_NONE;
31      g_clearOrderly = FALSE;
32      if(g_inFailureMode)
33      {
34          // Do failure mode processing
35          TpmFailureMode(requestSize, request, responseSize, response);
36          return;
37      }
38      // Query platform to get the NV state.  The result state is saved internally
39      // and will be reported by NvIsAvailable(). The reference code requires that
40      // accessibility of NV does not change during the execution of a command.
41      // Specifically, if NV is available when the command execution starts and then
42      // is not available later when it is necessary to write to NV, then the TPM
43      // will go into failure mode.
44      NvCheckState();
45
46      // Due to the limitations of the simulation, TPM clock must be explicitly
47      // synchronized with the system clock whenever a command is received.
48      // This function call is not necessary in a hardware TPM. However, taking
49      // a snapshot of the hardware timer at the beginning of the command allows
50      // the time value to be consistent for the duration of the command execution.
51      TimeUpdateToCurrent();
52
53      // Any command through this function will unceremoniously end the
54      // _TPM_Hash_Data/_TPM_Hash_End sequence.
55      if(g_DRTMHandle != TPM_RH_UNASSIGNED)
56          ObjectTerminateEvent();
57
58      // Get command buffer size and command buffer.
59      command.parameterBuffer = request;
60      command.parameterSize = requestSize;
61
62      // Parse command header: tag, commandSize and command.code.
63      // First parse the tag. The unmarshaling routine will validate
64      // that it is either TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS.
65      result = TPMI_ST_COMMAND_TAG_Unmarshal(&command.tag,
66                                             &command.parameterBuffer,
67                                             &command.parameterSize);
68      if(result != TPM_RC_SUCCESS)
69          goto Cleanup;
70      // Unmarshal the commandSize indicator.
71      result = UINT32_Unmarshal(&commandSize,
72                                &command.parameterBuffer,
73                                &command.parameterSize);
74      if(result != TPM_RC_SUCCESS)
75          goto Cleanup;
76      // On a TPM that receives bytes on a port, the number of bytes that were
77      // received on that port is requestSize it must be identical to commandSize.
78      // In addition, commandSize must not be larger than MAX_COMMAND_SIZE allowed
79      // by the implementation. The check against MAX_COMMAND_SIZE may be redundant
80      // as the input processing (the function that receives the command bytes and
81      // places them in the input buffer) would likely have the input truncated when
82      // it reaches MAX_COMMAND_SIZE, and requestSize would not equal commandSize.
83      if(commandSize != requestSize || commandSize > MAX_COMMAND_SIZE)
84      {
85          result = TPM_RC_COMMAND_SIZE;
86          goto Cleanup;
87      }
88      // Unmarshal the command code.
89      result = TPM_CC_Unmarshal(&command.code, &command.parameterBuffer,
90                                &command.parameterSize);
91      if(result != TPM_RC_SUCCESS)
92          goto Cleanup;
93      // Check to see if the command is implemented.
```

```
94      command.index = CommandCodeToCommandIndex(command.code);
95      if(UNIMPLEMENTED_COMMAND_INDEX == command.index)
96      {
97          result = TPM_RC_COMMAND_CODE;
98          goto Cleanup;
99      }
100 #if  FIELD_UPGRADE_IMPLEMENTED  == YES
101     // If the TPM is in FUM, then the only allowed command is
102     // TPM_CC_FieldUpgradeData.
103     if(IsFieldUgradeMode() && (command.code != TPM_CC_FieldUpgradeData))
104     {
105         result = TPM_RC_UPGRADE;
106         goto Cleanup;
107     }
108     else
109 #endif
110     // Excepting FUM, the TPM only accepts TPM2_Startup() after
111     // _TPM_Init. After getting a TPM2_Startup(), TPM2_Startup()
112     // is no longer allowed.
113     if((!TPMIsStarted() && command.code != TPM_CC_Startup)
114         || (TPMIsStarted() && command.code == TPM_CC_Startup))
115     {
116         result = TPM_RC_INITIALIZE;
117         goto Cleanup;
118     }
119 // Start regular command process.
120     NvIndexCacheInit();
121     // Parse Handle buffer.
122     result = ParseHandleBuffer(&command);
123     if(result != TPM_RC_SUCCESS)
124         goto Cleanup;
125     // All handles in the handle area are required to reference TPM-resident
126     // entities.
127     result = EntityGetLoadStatus(&command);
128     if(result != TPM_RC_SUCCESS)
129         goto Cleanup;
130     // Authorization session handling for the command.
131     ClearCpRpHashes(&command);
132     if(command.tag == TPM_ST_SESSIONS)
133     {
134         // Find out session buffer size.
135         result = UINT32_Unmarshal((UINT32 *)&command.authSize,
136                                   &command.parameterBuffer,
137                                   &command.parameterSize);
138         if(result != TPM_RC_SUCCESS)
139             goto Cleanup;
140         // Perform sanity check on the unmarshaled value. If it is smaller than
141         // the smallest possible session or larger than the remaining size of
142         // the command, then it is an error. NOTE: This check could pass but the
143         // session size could still be wrong. That will be determined after the
144         // sessions are unmarshaled.
145         if(command.authSize < 9
146            || command.authSize > command.parameterSize)
147         {
148             result = TPM_RC_SIZE;
149             goto Cleanup;
150         }
151         command.parameterSize -= command.authSize;
152
153         // The actions of ParseSessionBuffer() are described in the introduction.
154         // As the sessions are parsed command.parameterBuffer is advanced so, on a
155         // successful return, command.parameterBuffer should be pointing at the
156         // first byte of the parameters.
157         result = ParseSessionBuffer(&command);
158         if(result != TPM_RC_SUCCESS)
159             goto Cleanup;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
160         }
161         else
162         {
163             command.authSize = 0;
164             // The command has no authorization sessions.
165             // If the command requires authorizations, then CheckAuthNoSession() will
166             // return an error.
167             result = CheckAuthNoSession(&command);
168             if(result != TPM_RC_SUCCESS)
169                 goto Cleanup;
170         }
171         // Set up the response buffer pointers. CommandDispatch will marshal the
172         // response parameters starting at the address in command.responseBuffer.
173     //*response = MemoryGetResponseBuffer(command.index);
174         // leave space for the command header
175         command.responseBuffer = *response + STD_RESPONSE_HEADER;
176
177         // leave space for the parameter size field if needed
178         if(command.tag == TPM_ST_SESSIONS)
179             command.responseBuffer += sizeof(UINT32);
180         if(IsHandleInResponse(command.index))
181             command.responseBuffer += sizeof(TPM_HANDLE);
182
183         // CommandDispatcher returns a response handle buffer and a response parameter
184         // buffer if it succeeds. It will also set the parameterSize field in the
185         // buffer if the tag is TPM_RC_SESSIONS.
186         result = CommandDispatcher(&command);
187         if(result != TPM_RC_SUCCESS)
188             goto Cleanup;
189
190         // Build the session area at the end of the parameter area.
191         BuildResponseSession(&command);
192
193     Cleanup:
194         if(g_clearOrderly == TRUE
195             && NV_IS_ORDERLY)
196         {
197     #if USE_DA_USED
198             gp.orderlyState = g_daUsed ? SU_DA_USED_VALUE : SU_NONE_VALUE;
199     #else
200             gp.orderlyState = SU_NONE_VALUE;
201     #endif
202             NV_SYNC_PERSISTENT(orderlyState);
203         }
204         // This implementation loads an "evict" object to a transient object slot in
205         // RAM whenever an "evict" object handle is used in a command so that the
206         // access to any object is the same. These temporary objects need to be
207         // cleared from RAM whether the command succeeds or fails.
208         ObjectCleanupEvict();
209
210         // The parameters and sessions have been marshaled. Now tack on the header and
211         // set the sizes
212         BuildResponseHeader(&command, *response, result);
213
214         // Try to commit all the writes to NV if any NV write happened during this
215         // command execution. This check should be made for both succeeded and failed
216         // commands, because a failed one may trigger a NV write in DA logic as well.
217         // This is the only place in the command execution path that may call the NV
218         // commit. If the NV commit fails, the TPM should be put in failure mode.
219         if((g_updateNV != UT_NONE) && !g_inFailureMode)
220         {
221             if(g_updateNV == UT_ORDERLY)
222                 NvUpdateIndexOrderlyData();
223             if(!NvCommit())
224                 FAIL(FATAL_ERROR_INTERNAL);
225             g_updateNV = UT_NONE;
```

```
226     }
227     pAssert((UINT32)command.parameterSize <= maxResponse);
228
229     // Clear unused bits in response buffer.
230     MemorySet(*response + *responseSize, 0, maxResponse - *responseSize);
231
232     // as a final act, and not before, update the response size.
233     *responseSize = (UINT32)command.parameterSize;
234
235     return;
236 }
```

6.3 CommandDispatcher.c

6.3.1 Introduction

CommandDispatcher() performs the following operations:

- unmarshals command parameters from the input buffer;

NOTE 1 Unlike other unmarshaling functions, *parmBufferStart* does not advance. *parmBufferSize* Is reduced.

- invokes the function that performs the command actions;
- marshals the returned handles, if any; and
- marshals the returned parameters, if any, into the output buffer putting in the *parameterSize* field if authorization sessions are present.

NOTE 2 The output buffer is the return from the MemoryGetResponseBuffer() function. It includes the header, handles, response parameters, and authorization area. *respParmSize* is the response parameter size, and does not include the header, handles, or authorization area.

NOTE 3 The reference implementation is permitted to do compare operations over a union as a byte array. Therefore, the command parameter *in* structure must be initialized (e.g., zeroed) before unmarshaling so that the compare operation is valid in cases where some bytes are unused.

6.3.1.1 Includes and Typedefs

```
1  #include "Tpm.h"
2  #include "Marshal.h"
3  #if TABLE_DRIVEN_DISPATCH
4  typedef TPM_RC(NoFlagFunction)(void *target, BYTE **buffer, INT32 *size);
5  typedef TPM_RC(FlagFunction)(void *target, BYTE **buffer, INT32 *size, BOOL flag);
6  typedef FlagFunction *UNMARSHAL_t;
7  typedef INT16(MarshalFunction)(void *source, BYTE **buffer, INT32 *size);
8  typedef MarshalFunction *MARSHAL_t;
9  typedef TPM_RC(COMMAND_NO_ARGS)(void);
10 typedef TPM_RC(COMMAND_IN_ARG)(void *in);
11 typedef TPM_RC(COMMAND_OUT_ARG)(void *out);
12 typedef TPM_RC(COMMAND_INOUT_ARG)(void *in, void *out);
13 typedef union COMMAND_t
14 {
15     COMMAND_NO_ARGS          *noArgs;
16     COMMAND_IN_ARG           *inArg;
17     COMMAND_OUT_ARG          *outArg;
18     COMMAND_INOUT_ARG        *inOutArg;
19 } COMMAND_t;
```

This structure is used by ParseHandleBuffer() and CommandDispatcher(). The parameters in this structure are unique for each command. The parameters are:

 Copyright © TCG 2006-2020

command	**holds the address of the command processing function that is called by Command Dispatcher.**
inSize	this is the size of the command-dependent input structure. The input structure holds the unmarshaled handles and command parameters. If the command takes no arguments (handles or parameters) then *inSize* will have a value of 0.
outSize	this is the size of the command-dependent output structure. The output structure holds the results of the command in an unmarshaled form. When command processing is completed, these values are marshaled into the output buffer. It is always the case that the unmarshaled version of an output structure is larger then the marshaled version. This is because the marshaled version contains the exact same number of significant bytes but with padding removed.
typesOffsets	this parameter points to the list of data types that are to be marshaled or unmarshaled. The list of types follows the *offsets* array. The offsets array is variable sized so the *typesOffset* filed is necessary for the handle and command processing to be able to find the types that are being handled. The *offsets* array may be empty. The types structure is described below.
offsets	this is an array of offsets of each of the parameters in the command or response. When processing the command parameters (not handles) the list contains the offset of the next parameter. For example, if the first command parameter has a size of 4 and there is a second command parameter, then the offset would be 4, indicating that the second parameter starts at 4. If the second parameter has a size of 8, and there is a third parameter, then the second entry in offsets is 12 (4 for the first parameter and 8 for the second). An offset value of 0 in the list indicates the start of the response parameter list. When CommandDispatcher() hits this value, it will stop unmarshaling the parameters and call *command*. If a command has no response parameters and only one command parameter, then offsets can be an empty list.

```
20  typedef struct COMMAND_DESCRIPTOR_t
21  {
22      COMMAND_t       command;        // Address of the command
23      UINT16          inSize;         // Maximum size of the input structure
24      UINT16          outSize;        // Maximum size of the output structure
25      UINT16          typesOffset;    // address of the types field
26      UINT16          offsets[1];
27  } COMMAND_DESCRIPTOR_t;
```

The *types* list is an encoded byte array. The byte value has two parts. The most significant bit is used when a parameter takes a flag and indicates if the flag should be SET or not. The remaining 7 bits are an index into an array of addresses of marshaling and unmarshaling functions. The array of functions is divided into 6 sections with a value assigned to denote the start of that section (and the end of the previous section). The defined offset values for each section are:

 Copyright © TCG 2006-2020

0	unmarshaling for handles that do not take flags
HANDLE_FIRST_FLAG_TYPE	unmarshaling for handles that take flags
PARAMETER_FIRST_TYPE	unmarshaling for parameters that do not take flags
PARAMETER_FIRST_FLAG_TYPE	unmarshaling for parameters that take flags
1	marshaling for handles
RESPONSE_PARAMETER_FIRST_TYPE	marshaling for parameters
RESPONSE_PARAMETER_LAST_TYPE	is the last value in the list of marshaling and unmarshaling functions.

The types list is constructed with a byte of 0xff at the end of the command parameters and with an 0xff at the end of the response parameters.

```
28  #if COMPRESSED_LISTS
29  #   define PAD_LIST 0
30  #else
31  #   define PAD_LIST 1
32  #endif
33  #define _COMMAND_TABLE_DISPATCH_
34  #include "CommandDispatchData.h"
35  #define TEST_COMMAND    TPM_CC_Startup
36  #define NEW_CC
37  #else
38  #include "Commands.h"
39  #endif
```

6.3.1.2 Marshal/Unmarshal Functions

6.3.1.2.1 ParseHandleBuffer()

This is the table-driven version of the handle buffer unmarshaling code

```
40  TPM_RC
41  ParseHandleBuffer(
42      COMMAND                 *command
43      )
44  {
45      TPM_RC                   result;
46  #if TABLE_DRIVEN_DISPATCH
47      COMMAND_DESCRIPTOR_t    *desc;
48      BYTE                    *types;
49      BYTE                     type;
50      BYTE                     dType;
51
52      // Make sure that nothing strange has happened
53      pAssert(command->index
54              < sizeof(s_CommandDataArray) / sizeof(COMMAND_DESCRIPTOR_t *));
55      // Get the address of the descriptor for this command
56      desc = s_CommandDataArray[command->index];
57
58      pAssert(desc != NULL);
59      // Get the associated list of unmarshaling data types.
60      types = &((BYTE *)desc)[desc->typesOffset];
61
62  //    if(s_ccAttr[commandIndex].commandIndex == TEST_COMMAND)
63  //        commandIndex = commandIndex;
64      // No handles yet
65      command->handleNum = 0;
66
```

 Copyright © TCG 2006-2020

```
67        // Get the first type value
68        for(type = *types++;
69            // check each byte to make sure that we have not hit the start
70            // of the parameters
71        (dType = (type & 0x7F)) < PARAMETER_FIRST_TYPE;
72        // get the next type
73            type = *types++)
74        {
75    #if TABLE_DRIVEN_MARSHAL
76            marshalIndex_t      index;
77            index = UnmarshalArray[dType] | ((type & 0x80) ? NULL_FLAG : 0);
78            result = Unmarshal(index, &(command->handles[command->handleNum]),
79                                  &command->parameterBuffer, &command->parameterSize);
80
81    #else
82            // See if unmarshaling of this handle type requires a flag
83            if(dType < HANDLE_FIRST_FLAG_TYPE)
84            {
85                // Look up the function to do the unmarshaling
86                NoFlagFunction  *f = (NoFlagFunction *)UnmarshalArray[dType];
87                // call it
88                result = f(&(command->handles[command->handleNum]),
89                           &command->parameterBuffer,
90                           &command->parameterSize);
91            }
92            else
93            {
94                //  Look up the function
95                FlagFunction    *f = UnmarshalArray[dType];
96
97                // Call it setting the flag to the appropriate value
98                result = f(&(command->handles[command->handleNum]),
99                           &command->parameterBuffer,
100                          &command->parameterSize, (type & 0x80) != 0);
101           }
102   #endif
103
104           // Got a handle
105           // We do this first so that the match for the handle offset of the
106           // response code works correctly.
107           command->handleNum += 1;
108           if(result != TPM_RC_SUCCESS)
109               // if the unmarshaling failed, return the response code with the
110               // handle indication set
111               return result + TPM_RC_H + (command->handleNum * TPM_RC_1);
112       }
113   #else
114       BYTE            **handleBufferStart = &command->parameterBuffer;
115       INT32           *bufferRemainingSize = &command->parameterSize;
116       TPM_HANDLE      *handles = &command->handles[0];
117       UINT32          *handleCount = &command->handleNum;
118       *handleCount = 0;
119       switch(command->code)
120       {
121   #include "HandleProcess.h"
122   #undef handles
123           default:
124               FAIL(FATAL_ERROR_INTERNAL);
125               break;
126       }
127   #endif
128       return TPM_RC_SUCCESS;
129   }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

6.3.1.2.2 CommandDispatcher()

Function to unmarshal the command parameters, call the selected action code, and marshal the response parameters.

```
130 TPM_RC
131 CommandDispatcher(
132     COMMAND                 *command
133     )
134 {
135 #if !TABLE_DRIVEN_DISPATCH
136     TPM_RC        result;
137     BYTE         **paramBuffer = &command->parameterBuffer;
138     INT32        *paramBufferSize = &command->parameterSize;
139     BYTE         **responseBuffer = &command->responseBuffer;
140     INT32        *respParmSize = &command->parameterSize;
141     INT32         rSize;
142     TPM_HANDLE   *handles = &command->handles[0];
143 //
144     command->handleNum = 0;                 // The command-specific code knows how
145                                             // many handles there are. This is for
146                                             // cataloging the number of response
147                                             // handles
148     MemoryIoBufferAllocationReset();        // Initialize so that allocation will
149                                             // work properly
150     switch(GetCommandCode(command->index))
151     {
152 #include "CommandDispatcher.h"
153 
154         default:
155             FAIL(FATAL_ERROR_INTERNAL);
156             break;
157     }
158 Exit:
159     MemoryIoBufferZero();
160     return result;
161 #else
162     COMMAND_DESCRIPTOR_t    *desc;
163     BYTE                    *types;
164     BYTE                     type;
165     UINT16                  *offsets;
166     UINT16                   offset = 0;
167     UINT32                   maxInSize;
168     BYTE                    *commandIn;
169     INT32                    maxOutSize;
170     BYTE                    *commandOut;
171     COMMAND_t                cmd;
172     TPM_HANDLE              *handles;
173     UINT32                   hasInParameters = 0;
174     BOOL                     hasOutParameters = FALSE;
175     UINT32                   pNum = 0;
176     BYTE                     dType;     // dispatch type
177     TPM_RC                   result;
178 //
179     // Get the address of the descriptor for this command
180     pAssert(command->index
181             < sizeof(s_CommandDataArray) / sizeof(COMMAND_DESCRIPTOR_t *));
182     desc = s_CommandDataArray[command->index];
183 
184     // Get the list of parameter types for this command
185     pAssert(desc != NULL);
186     types = &((BYTE *)desc)[desc->typesOffset];
187 
188     // Get a pointer to the list of parameter offsets
189     offsets = &desc->offsets[0];
```

```
190     // pointer to handles
191     handles = command->handles;
192
193     // Get the size required to hold all the unmarshaled parameters for this command
194     maxInSize = desc->inSize;
195     // and the size of the output parameter structure returned by this command
196     maxOutSize = desc->outSize;
197
198     MemoryIoBufferAllocationReset();
199     // Get a buffer for the input parameters
200     commandIn = MemoryGetInBuffer(maxInSize);
201     // And the output parameters
202     commandOut = (BYTE *)MemoryGetOutBuffer((UINT32)maxOutSize);
203
204     // Get the address of the action code dispatch
205     cmd = desc->command;
206
207     // Copy any handles into the input buffer
208     for(type = *types++; (type & 0x7F) < PARAMETER_FIRST_TYPE; type = *types++)
209     {
210         // 'offset' was initialized to zero so the first unmarshaling will always
211         // be to the start of the data structure
212         *(TPM_HANDLE *)&(commandIn[offset]) = *handles++;
213         // This check is used so that we don't have to add an additional offset
214         // value to the offsets list to correspond to the stop value in the
215         // command parameter list.
216         if(*types != 0xFF)
217             offset = *offsets++;
218 //          maxInSize -= sizeof(TPM_HANDLE);
219         hasInParameters++;
220     }
221     // Exit loop with type containing the last value read from types
222     // maxInSize has the amount of space remaining in the command action input
223     // buffer. Make sure that we don't have more data to unmarshal than is going to
224     // fit.
225
226     // type contains the last value read from types so it is not necessary to
227     // reload it, which is good because *types now points to the next value
228     for(; (dType = (type & 0x7F)) <= PARAMETER_LAST_TYPE; type = *types++)
229     {
230         pNum++;
231 #if TABLE_DRIVEN_MARSHAL
232         {
233             marshalIndex_t      index = UnmarshalArray[dType];
234             index |= (type & 0x80) ? NULL_FLAG : 0;
235             result = Unmarshal(index, &commandIn[offset], &command->parameterBuffer,
236                                &command->parameterSize);
237         }
238 #else
239         if(dType < PARAMETER_FIRST_FLAG_TYPE)
240         {
241             NoFlagFunction      *f = (NoFlagFunction *)UnmarshalArray[dType];
242             result = f(&commandIn[offset], &command->parameterBuffer,
243                        &command->parameterSize);
244         }
245         else
246         {
247             FlagFunction        *f = UnmarshalArray[dType];
248             result = f(&commandIn[offset], &command->parameterBuffer,
249                        &command->parameterSize,
250                        (type & 0x80) != 0);
251         }
252 #endif
253         if(result != TPM_RC_SUCCESS)
254         {
255             result += TPM_RC_P + (TPM_RC_1 * pNum);
```

```
256                 goto Exit;
257             }
258             // This check is used so that we don't have to add an additional offset
259             // value to the offsets list to correspond to the stop value in the
260             // command parameter list.
261             if(*types != 0xFF)
262                 offset = *offsets++;
263             hasInParameters++;
264         }
265         // Should have used all the bytes in the input
266         if(command->parameterSize != 0)
267         {
268             result = TPM_RC_SIZE;
269             goto Exit;
270         }
271
272         // The command parameter unmarshaling stopped when it hit a value that was out
273         // of range for unmarshaling values and left *types pointing to the first
274         // marshaling type. If that type happens to be the STOP value, then there
275         // are no response parameters. So, set the flag to indicate if there are
276         // output parameters.
277         hasOutParameters = *types != 0xFF;
278
279         // There are four cases for calling, with and without input parameters and with
280         // and without output parameters.
281         if(hasInParameters > 0)
282         {
283             if(hasOutParameters)
284                 result = cmd.inOutArg(commandIn, commandOut);
285             else
286                 result = cmd.inArg(commandIn);
287         }
288         else
289         {
290             if(hasOutParameters)
291                 result = cmd.outArg(commandOut);
292             else
293                 result = cmd.noArgs();
294         }
295         if(result != TPM_RC_SUCCESS)
296            goto Exit;
297
298         // Offset in the marshaled output structure
299         offset = 0;
300
301         // Process the return handles, if any
302         command->handleNum = 0;
303
304         // Could make this a loop to process output handles but there is only ever
305         // one handle in the outputs (for now).
306         type = *types++;
307         if((dType = (type & 0x7F)) < RESPONSE_PARAMETER_FIRST_TYPE)
308         {
309             // The out->handle value was referenced as TPM_HANDLE in the
310             // action code so it has to be properly aligned.
311             command->handles[command->handleNum++] =
312                 *((TPM_HANDLE *)&(commandOut[offset]));
313             maxOutSize -= sizeof(UINT32);
314             type = *types++;
315             offset = *offsets++;
316         }
317         // Use the size of the command action output buffer as the maximum for the
318         // number of bytes that can get marshaled. Since the marshaling code has
319         // no pointers to data, all of the data being returned has to be in the
320         // command action output buffer. If we try to marshal more bytes than
321         // could fit into the output buffer, we need to fail.
```

```
322         for(;(dType = (type & 0x7F)) <= RESPONSE_PARAMETER_LAST_TYPE
323             && !g_inFailureMode; type = *types++)
324         {
325 #if TABLE_DRIVEN_MARSHAL
326             marshalIndex_t      index = MarshalArray[dType];
327             command->parameterSize += Marshal(index, &commandOut[offset],
328                                              &command->responseBuffer,
329                                              &maxOutSize);
330 #else
331             const MARSHAL_t     f = MarshalArray[dType];
332
333             command->parameterSize += f(&commandOut[offset],
334                                        &command->responseBuffer,
335                                        &maxOutSize);
336 #endif
337             offset = *offsets++;
338         }
339         result = (maxOutSize < 0) ? TPM_RC_FAILURE : TPM_RC_SUCCESS;
340     Exit:
341         MemoryIoBufferZero();
342         return result;
343 #endif
344     }
```

 Copyright © TCG 2006-2020

6.4 SessionProcess.c

6.4.1 Introduction

This file contains the subsystem that process the authorization sessions including implementation of the Dictionary Attack logic. ExecCommand() uses ParseSessionBuffer() to process the authorization session area of a command and BuildResponseSession() to create the authorization session area of a response.

6.4.2 Includes and Data Definitions

```
1   #define SESSION_PROCESS_C
2   #include "Tpm.h"
3   #include "ACT.h"
```

6.4.3 Authorization Support Functions

6.4.3.1 IsDAExempted()

This function indicates if a handle is exempted from DA logic. A handle is exempted if it is

a) a primary seed handle,

b) an object with *noDA* bit SET,

c) an NV Index with TPMA_NV_NO_DA bit SET, or

d) a PCR handle.

Return Value	Meaning
TRUE(1)	handle is exempted from DA logic
FALSE(0)	handle is not exempted from DA logic

```
4   BOOL
5   IsDAExempted(
6       TPM_HANDLE       handle         // IN: entity handle
7       )
8   {
9       BOOL         result = FALSE;
10  //
11      switch(HandleGetType(handle))
12      {
13          case TPM_HT_PERMANENT:
14              // All permanent handles, other than TPM_RH_LOCKOUT, are exempt from
15              // DA protection.
16              result = (handle != TPM_RH_LOCKOUT);
17              break;
18          // When this function is called, a persistent object will have been loaded
19          // into an object slot and assigned a transient handle.
20          case TPM_HT_TRANSIENT:
21          {
22              TPMA_OBJECT     attributes = ObjectGetPublicAttributes(handle);
23              result = IS_ATTRIBUTE(attributes, TPMA_OBJECT, noDA);
24              break;
25          }
26          case TPM_HT_NV_INDEX:
27          {
28              NV_INDEX             *nvIndex = NvGetIndexInfo(handle, NULL);
29              result = IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, NO_DA);
30              break;
```

 Copyright © TCG 2006-2020

```
31          }
32          case TPM_HT_PCR:
33              // PCRs are always exempted from DA.
34              result = TRUE;
35              break;
36          default:
37              break;
38      }
39      return result;
40  }
```

6.4.3.2 IncrementLockout()

This function is called after an authorization failure that involves use of an *authValue*. If the entity referenced by the handle is not exempt from DA protection, then the *failedTries* counter will be incremented.

Error Returns	Meaning
TPM_RC_AUTH_FAIL	authorization failure that caused DA lockout to increment
TPM_RC_BAD_AUTH	authorization failure did not cause DA lockout to increment

```
41  static TPM_RC
42  IncrementLockout(
43      UINT32              sessionIndex
44      )
45  {
46      TPM_HANDLE          handle = s_associatedHandles[sessionIndex];
47      TPM_HANDLE          sessionHandle = s_sessionHandles[sessionIndex];
48      SESSION            *session = NULL;
49  //
50      // Don't increment lockout unless the handle associated with the session
51      // is DA protected or the session is bound to a DA protected entity.
52      if(sessionHandle == TPM_RS_PW)
53      {
54          if(IsDAExempted(handle))
55              return TPM_RC_BAD_AUTH;
56      }
57      else
58      {
59          session = SessionGet(sessionHandle);
60          // If the session is bound to lockout, then use that as the relevant
61          // handle. This means that an authorization failure with a bound session
62          // bound to lockoutAuth will take precedence over any other
63          // lockout check
64          if(session->attributes.isLockoutBound == SET)
65              handle = TPM_RH_LOCKOUT;
66          if(session->attributes.isDaBound == CLEAR
67             && (IsDAExempted(handle) || session->attributes.includeAuth == CLEAR))
68             // If the handle was changed to TPM_RH_LOCKOUT, this will not return
69             // TPM_RC_BAD_AUTH
70              return TPM_RC_BAD_AUTH;
71      }
72      if(handle == TPM_RH_LOCKOUT)
73      {
74          pAssert(gp.lockOutAuthEnabled == TRUE);
75
76          // lockout is no longer enabled
77          gp.lockOutAuthEnabled = FALSE;
78
79          // For TPM_RH_LOCKOUT, if lockoutRecovery is 0, no need to update NV since
80          // the lockout authorization will be reset at startup.
81          if(gp.lockoutRecovery != 0)
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
82          {
83              if(NV_IS_AVAILABLE)
84                  // Update NV.
85                  NV_SYNC_PERSISTENT(lockOutAuthEnabled);
86              else
87                  // No NV access for now. Put the TPM in pending mode.
88                  s_DAPendingOnNV = TRUE;
89          }
90      }
91      else
92      {
93          if(gp.recoveryTime != 0)
94          {
95              gp.failedTries++;
96              if(NV_IS_AVAILABLE)
97                  // Record changes to NV. NvWrite will SET g_updateNV
98                  NV_SYNC_PERSISTENT(failedTries);
99              else
100                 // No NV access for now.  Put the TPM in pending mode.
101                 s_DAPendingOnNV = TRUE;
102         }
103     }
104     // Register a DA failure and reset the timers.
105     DARegisterFailure(handle);
106
107     return TPM_RC_AUTH_FAIL;
108 }
```

6.4.3.3 IsSessionBindEntity()

This function indicates if the entity associated with the handle is the entity, to which this session is bound. The binding would occur by making the **bind** parameter in TPM2_StartAuthSession() not equal to TPM_RH_NULL. The binding only occurs if the session is an HMAC session. The bind value is a combination of the Name and the *authValue* of the entity.

Return Value	Meaning
TRUE(1)	handle points to the session start entity
FALSE(0)	handle does not point to the session start entity

```
109 static BOOL
110 IsSessionBindEntity(
111     TPM_HANDLE       associatedHandle,  // IN: handle to be authorized
112     SESSION         *session            // IN: associated session
113     )
114 {
115     TPM2B_NAME     entity;             // The bind value for the entity
116 //
117     // If the session is not bound, return FALSE.
118     if(session->attributes.isBound)
119     {
120         // Compute the bind value for the entity.
121         SessionComputeBoundEntity(associatedHandle, &entity);
122
123         // Compare to the bind value in the session.
124         return MemoryEqual2B(&entity.b, &session->u1.boundEntity.b);
125     }
126     return FALSE;
127 }
```

 Copyright © TCG 2006-2020

6.4.3.4 IsPolicySessionRequired()

Checks if a policy session is required for a command. If a command requires DUP or ADMIN role authorization, then the handle that requires that role is the first handle in the command. This simplifies this checking. If a new command is created that requires multiple ADMIN role authorizations, then it will have to be special-cased in this function. A policy session is required if:

a) the command requires the DUP role,

b) the command requires the ADMIN role and the authorized entity is an object and its *adminWithPolicy* bit is SET, or

c) the command requires the ADMIN role and the authorized entity is a permanent handle or an NV Index.

d) The authorized entity is a PCR belonging to a policy group, and has its policy initialized

Return Value	Meaning
TRUE(1)	policy session is required
FALSE(0)	policy session is not required

```
128 static BOOL
129 IsPolicySessionRequired(
130     COMMAND_INDEX    commandIndex,  // IN: command index
131     UINT32           sessionIndex   // IN: session index
132     )
133 {
134     AUTH_ROLE       role = CommandAuthRole(commandIndex, sessionIndex);
135     TPM_HT          type = HandleGetType(s_associatedHandles[sessionIndex]);
136 //
137     if(role == AUTH_DUP)
138         return TRUE;
139     if(role == AUTH_ADMIN)
140     {
141         // We allow an exception for ADMIN role in a transient object. If the object
142         // allows ADMIN role actions with authorization, then policy is not
143         // required. For all other cases, there is no way to override the command
144         // requirement that a policy be used
145         if(type == TPM_HT_TRANSIENT)
146         {
147             OBJECT      *object = HandleToObject(s_associatedHandles[sessionIndex]);
148
149             if(!IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT,
150                              adminWithPolicy))
151                 return FALSE;
152         }
153         return TRUE;
154     }
155
156     if(type == TPM_HT_PCR)
157     {
158         if(PCRPolicyIsAvailable(s_associatedHandles[sessionIndex]))
159         {
160             TPM2B_DIGEST        policy;
161             TPMI_ALG_HASH       policyAlg;
162             policyAlg = PCRGetAuthPolicy(s_associatedHandles[sessionIndex],
163                                          &policy);
164             if(policyAlg != TPM_ALG_NULL)
165                 return TRUE;
166         }
167     }
168     return FALSE;
169 }
```

 Copyright © TCG 2006-2020

6.4.3.5 IsAuthValueAvailable()

This function indicates if *authValue* is available and allowed for USER role authorization of an entity.

This function is similar to IsAuthPolicyAvailable() except that it does not check the size of the *authValue* as IsAuthPolicyAvailable() does (a null *authValue* is a valid authorization, but a null policy is not a valid policy).

This function does not check that the handle reference is valid or if the entity is in an enabled hierarchy. Those checks are assumed to have been performed during the handle unmarshaling.

Return Value	Meaning
TRUE(1)	*authValue* is available
FALSE(0)	*authValue* is not available

```
170  static BOOL
171  IsAuthValueAvailable(
172      TPM_HANDLE        handle,         // IN: handle of entity
173      COMMAND_INDEX     commandIndex,   // IN: command index
174      UINT32            sessionIndex    // IN: session index
175      )
176  {
177      BOOL              result = FALSE;
178  //
179      switch(HandleGetType(handle))
180      {
181          case TPM_HT_PERMANENT:
182              switch(handle)
183              {
184                      // At this point hierarchy availability has already been
185                      // checked so primary seed handles are always available here
186                  case TPM_RH_OWNER:
187                  case TPM_RH_ENDORSEMENT:
188                  case TPM_RH_PLATFORM:
189  #ifdef VENDOR_PERMANENT
190                      // This vendor defined handle associated with the
191                      // manufacturer's shared secret
192                  case VENDOR_PERMANENT:
193  #endif
194                      // The DA checking has been performed on LockoutAuth but we
195                      // bypass the DA logic if we are using lockout policy. The
196                      // policy would allow execution to continue an lockoutAuth
197                      // could be used, even if direct use of lockoutAuth is disabled
198                  case TPM_RH_LOCKOUT:
199                      // NullAuth is always available.
200                  case TPM_RH_NULL:
201                      result = TRUE;
202                      break;
203                  FOR_EACH_ACT(CASE_ACT_HANDLE)
204                  {
205                      // The ACT auth value is not available if the platform is disabled
206                       result = g_phEnable == SET;
207                       break;
208                  }
209                  default:
210                      // Otherwise authValue is not available.
211                      break;
212              }
213              break;
214          case TPM_HT_TRANSIENT:
215              // A persistent object has already been loaded and the internal
216              // handle changed.
217          {
```

```
218             OBJECT          *object;
219             TPMA_OBJECT      attributes;
220 //
221             object = HandleToObject(handle);
222             attributes = object->publicArea.objectAttributes;
223
224             // authValue is always available for a sequence object.
225             // An alternative for this is to
226             // SET_ATTRIBUTE(object->publicArea, TPMA_OBJECT, userWithAuth) when the
227             // sequence is started.
228             if(ObjectIsSequence(object))
229             {
230                 result = TRUE;
231                 break;
232             }
233             // authValue is available for an object if it has its sensitive
234             // portion loaded and
235             //  1. userWithAuth bit is SET, or
236             //  2. ADMIN role is required
237             if(object->attributes.publicOnly == CLEAR
238                && (IS_ATTRIBUTE(attributes, TPMA_OBJECT, userWithAuth)
239                    || (CommandAuthRole(commandIndex, sessionIndex) == AUTH_ADMIN
240                        && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, adminWithPolicy))))
241                 result = TRUE;
242         }
243         break;
244         case TPM_HT_NV_INDEX:
245             // NV Index.
246         {
247             NV_REF           locator;
248             NV_INDEX        *nvIndex = NvGetIndexInfo(handle, &locator);
249             TPMA_NV          nvAttributes;
250 //
251             pAssert(nvIndex != 0);
252
253             nvAttributes = nvIndex->publicArea.attributes;
254
255             if(IsWriteOperation(commandIndex))
256             {
257                 // AuthWrite can't be set for a PIN index
258                 if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, AUTHWRITE))
259                     result = TRUE;
260             }
261             else
262             {
263                 // A "read" operation
264                 // For a PIN Index, the authValue is available as long as the
265                 // Index has been written and the pinCount is less than pinLimit
266                 if(IsNvPinFailIndex(nvAttributes)
267                    || IsNvPinPassIndex(nvAttributes))
268                 {
269                     NV_PIN          pin;
270                     if(!IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN))
271                         break; // return false
272                     // get the index values
273                     pin.intVal = NvGetUINT64Data(nvIndex, locator);
274                     if(pin.pin.pinCount < pin.pin.pinLimit)
275                         result = TRUE;
276                 }
277                 // For non-PIN Indexes, need to allow use of the authValue
278                 else if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, AUTHREAD))
279                     result = TRUE;
280             }
281         }
282         break;
283         case TPM_HT_PCR:
```


 Copyright © TCG 2006-2020

```
284                 // PCR handle.
285                 // authValue is always allowed for PCR
286                 result = TRUE;
287                 break;
288             default:
289                 // Otherwise, authValue is not available
290                 break;
291         }
292         return result;
293     }
```

6.4.3.6 IsAuthPolicyAvailable()

This function indicates if an *authPolicy* is available and allowed.

This function does not check that the handle reference is valid or if the entity is in an enabled hierarchy. Those checks are assumed to have been performed during the handle unmarshaling.

Return Value	Meaning
TRUE(1)	*authPolicy* is available
FALSE(0)	*authPolicy* is not available

```
294 static BOOL
295 IsAuthPolicyAvailable(
296     TPM_HANDLE       handle,        // IN: handle of entity
297     COMMAND_INDEX    commandIndex,  // IN: command index
298     UINT32           sessionIndex   // IN: session index
299     )
300 {
301     BOOL            result = FALSE;
302 //
303     switch(HandleGetType(handle))
304     {
305         case TPM_HT_PERMANENT:
306             switch(handle)
307             {
308                 // At this point hierarchy availability has already been checked.
309                 case TPM_RH_OWNER:
310                     if(gp.ownerPolicy.t.size != 0)
311                         result = TRUE;
312                     break;
313                 case TPM_RH_ENDORSEMENT:
314                     if(gp.endorsementPolicy.t.size != 0)
315                         result = TRUE;
316                     break;
317                 case TPM_RH_PLATFORM:
318                     if(gc.platformPolicy.t.size != 0)
319                         result = TRUE;
320                     break;
321 #define ACT_GET_POLICY(N)                                                       \
322                 case TPM_RH_ACT_##N:                                            \
323                     if(go.ACT_##N.authPolicy.t.size != 0)                       \
324                         result = TRUE;                                          \
325                     break;
326
327                 FOR_EACH_ACT(ACT_GET_POLICY)
328
329                 case TPM_RH_LOCKOUT:
330                     if(gp.lockoutPolicy.t.size != 0)
331                         result = TRUE;
332                     break;
333                 default:
```

 Copyright © TCG 2006-2020

```
334                         break;
335                 }
336                 break;
337         case TPM_HT_TRANSIENT:
338         {
339                 // Object handle.
340                 // An evict object would already have been loaded and given a
341                 // transient object handle by this point.
342                 OBJECT  *object = HandleToObject(handle);
343                 // Policy authorization is not available for an object with only
344                 // public portion loaded.
345                 if(object->attributes.publicOnly == CLEAR)
346                 {
347                     // Policy authorization is always available for an object but
348                     // is never available for a sequence.
349                     if(!ObjectIsSequence(object))
350                         result = TRUE;
351                 }
352                 break;
353         }
354         case TPM_HT_NV_INDEX:
355                 // An NV Index.
356         {
357                 NV_INDEX         *nvIndex = NvGetIndexInfo(handle, NULL);
358                 TPMA_NV           nvAttributes = nvIndex->publicArea.attributes;
359 //
360                 // If the policy size is not zero, check if policy can be used.
361                 if(nvIndex->publicArea.authPolicy.t.size != 0)
362                 {
363                     // If policy session is required for this handle, always
364                     // uses policy regardless of the attributes bit setting
365                     if(IsPolicySessionRequired(commandIndex, sessionIndex))
366                         result = TRUE;
367                     // Otherwise, the presence of the policy depends on the NV
368                     // attributes.
369                     else if(IsWriteOperation(commandIndex))
370                     {
371                         if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, POLICYWRITE))
372                             result = TRUE;
373                     }
374                     else
375                     {
376                         if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, POLICYREAD))
377                             result = TRUE;
378                     }
379                 }
380         }
381         break;
382         case TPM_HT_PCR:
383                 // PCR handle.
384                 if(PCRPolicyIsAvailable(handle))
385                     result = TRUE;
386                 break;
387         default:
388                 break;
389     }
390     return result;
391 }
```

6.4.4 Session Parsing Functions

6.4.4.1 ClearCpRpHashes()

```
392 void
```

```
393 ClearCpRpHashes(
394     COMMAND           *command
395     )
396 {
397 #if ALG_SHA1
398     command->sha1CpHash.t.size = 0;
399     command->sha1RpHash.t.size = 0;
400 #endif
401 #if ALG_SHA256
402     command->sha256CpHash.t.size = 0;
403     command->sha256RpHash.t.size = 0;
404 #endif
405 #if ALG_SHA384
406     command->sha384CpHash.t.size = 0;
407     command->sha384RpHash.t.size = 0;
408 #endif
409 #if ALG_SHA512
410     command->sha512CpHash.t.size = 0;
411     command->sha512RpHash.t.size = 0;
412 #endif
413 #if ALG_SM3_256
414     command->sm3_256CpHash.t.size = 0;
415     command->sm3_256RpHash.t.size = 0;
416 #endif
417 }
```

6.4.4.2 GetCpHashPointer()

Function to get a pointer to the *cpHash* of the command

```
418 static TPM2B_DIGEST *
419 GetCpHashPointer(
420     COMMAND           *command,
421     TPMI_ALG_HASH      hashAlg
422     )
423 {
424     TPM2B_DIGEST      *retVal;
425 //
426     switch(hashAlg)
427     {
428 #if ALG_SHA1
429         case ALG_SHA1_VALUE:
430             retVal = (TPM2B_DIGEST *)&command->sha1CpHash;
431             break;
432 #endif
433 #if ALG_SHA256
434         case ALG_SHA256_VALUE:
435             retVal = (TPM2B_DIGEST *)&command->sha256CpHash;
436             break;
437 #endif
438 #if ALG_SHA384
439         case ALG_SHA384_VALUE:
440             retVal = (TPM2B_DIGEST *)&command->sha384CpHash;
441             break;
442 #endif
443 #if ALG_SHA512
444         case ALG_SHA512_VALUE:
445             retVal = (TPM2B_DIGEST *)&command->sha512CpHash;
446             break;
447 #endif
448 #if ALG_SM3_256
449         case ALG_SM3_256_VALUE:
450             retVal = (TPM2B_DIGEST *)&command->sm3_256CpHash;
451             break;
```

 Copyright © TCG 2006-2020

```
452  #endif
453          default:
454              retVal = NULL;
455              break;
456      }
457      return retVal;
458  }
```

6.4.4.3 GetRpHashPointer()

Function to get a pointer to the RpHash() of the command

```
459  static TPM2B_DIGEST *
460  GetRpHashPointer(
461      COMMAND          *command,
462      TPMI_ALG_HASH     hashAlg
463      )
464  {
465      TPM2B_DIGEST    *retVal;
466  //
467      switch(hashAlg)
468      {
469  #if ALG_SHA1
470          case ALG_SHA1_VALUE:
471              retVal = (TPM2B_DIGEST *)&command->sha1RpHash;
472              break;
473  #endif
474  #if ALG_SHA256
475          case ALG_SHA256_VALUE:
476              retVal = (TPM2B_DIGEST *)&command->sha256RpHash;
477              break;
478  #endif
479  #if ALG_SHA384
480          case ALG_SHA384_VALUE:
481              retVal = (TPM2B_DIGEST *)&command->sha384RpHash;
482              break;
483  #endif
484  #if ALG_SHA512
485          case ALG_SHA512_VALUE:
486              retVal = (TPM2B_DIGEST *)&command->sha512RpHash;
487              break;
488  #endif
489  #if ALG_SM3_256
490          case ALG_SM3_256_VALUE:
491              retVal = (TPM2B_DIGEST *)&command->sm3_256RpHash;
492              break;
493  #endif
494          default:
495              retVal = NULL;
496              break;
497      }
498      return retVal;
499  }
```

6.4.4.4 ComputeCpHash()

This function computes the *cpHash* as defined in Part 2 and described in Part 1.

```
500  static TPM2B_DIGEST *
501  ComputeCpHash(
502      COMMAND          *command,       // IN: command parsing structure
503      TPMI_ALG_HASH     hashAlg        // IN: hash algorithm
504      )
```

 Copyright © TCG 2006-2020

```
505 {
506     UINT32                  i;
507     HASH_STATE              hashState;
508     TPM2B_NAME              name;
509     TPM2B_DIGEST           *cpHash;
510 //
511     // cpHash = hash(commandCode [ || authName1
512     //                           [ || authName2
513     //                           [ || authName 3 ]]]
514     //                           [ || parameters])
515     // A cpHash can contain just a commandCode only if the lone session is
516     // an audit session.
517     // Get pointer to the hash value
518     cpHash = GetCpHashPointer(command, hashAlg);
519     if(cpHash->t.size == 0)
520     {
521         cpHash->t.size = CryptHashStart(&hashState, hashAlg);
522             //  Add commandCode.
523         CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), command->code);
524             //  Add authNames for each of the handles.
525         for(i = 0; i < command->handleNum; i++)
526             CryptDigestUpdate2B(&hashState, &EntityGetName(command->handles[i],
527                                                            &name)->b);
528             //  Add the parameters.
529         CryptDigestUpdate(&hashState, command->parameterSize,
530                           command->parameterBuffer);
531             //  Complete the hash.
532         CryptHashEnd2B(&hashState, &cpHash->b);
533     }
534     return cpHash;
535 }
```

6.4.4.5 GetCpHash()

This function is used to access a precomputed *cpHash*.

```
536 static TPM2B_DIGEST *
537 GetCpHash(
538     COMMAND          *command,
539     TPMI_ALG_HASH     hashAlg
540     )
541 {
542     TPM2B_DIGEST         *cpHash = GetCpHashPointer(command, hashAlg);
543  //
544     pAssert(cpHash->t.size != 0);
545     return cpHash;
546 }
```

6.4.4.6 CompareTemplateHash()

This function computes the template hash and compares it to the session *templateHash*. It is the hash of the second parameter assuming that the command is TPM2_Create(), TPM2_CreatePrimary(), or TPM2_CreateLoaded()

Return Value	Meaning
TRUE(1)	template hash equal to session->*templateHash*
FALSE(0)	template hash not equal to session->*templateHash*

```
547 static BOOL
548 CompareTemplateHash(
549     COMMAND          *command,        // IN: parsing structure
```

```
550     SESSION         *session        // IN: session data
551     )
552 {
553     BYTE                *pBuffer = command->parameterBuffer;
554     INT32                pSize = command->parameterSize;
555     TPM2B_DIGEST         tHash;
556     UINT16               size;
557 //
558     // Only try this for the three commands for which it is intended
559     if(command->code != TPM_CC_Create
560        && command->code != TPM_CC_CreatePrimary
561 #if CC_CreateLoaded
562        && command->code != TPM_CC_CreateLoaded
563 #endif
564        )
565         return FALSE;
566     // Assume that the first parameter is a TPM2B and unmarshal the size field
567     // Note: this will not affect the parameter buffer and size in the calling
568     // function.
569     if(UINT16_Unmarshal(&size, &pBuffer, &pSize) != TPM_RC_SUCCESS)
570         return FALSE;
571     // reduce the space in the buffer.
572     // NOTE: this could make pSize go negative if the parameters are not correct but
573     // the unmarshaling code does not try to unmarshal if the remaining size is
574     // negative.
575     pSize -= size;
576
577     // Advance the pointer
578     pBuffer += size;
579
580     // Get the size of what should be the template
581     if(UINT16_Unmarshal(&size, &pBuffer, &pSize) != TPM_RC_SUCCESS)
582         return FALSE;
583     // See if this is reasonable
584     if(size > pSize)
585         return FALSE;
586     // Hash the template data
587     tHash.t.size = CryptHashBlock(session->authHashAlg, size, pBuffer,
588                                   sizeof(tHash.t.buffer), tHash.t.buffer);
589     return(MemoryEqual2B(&session->u1.templateHash.b, &tHash.b));
590 }
```

6.4.4.7 CompareNameHash()

This function computes the name hash and compares it to the *nameHash* in the session data.

```
591 BOOL
592 CompareNameHash(
593     COMMAND         *command,       // IN: main parsing structure
594     SESSION         *session        // IN: session structure with nameHash
595     )
596 {
597     HASH_STATE           hashState;
598     TPM2B_DIGEST         nameHash;
599     UINT32               i;
600     TPM2B_NAME           name;
601 //
602     nameHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
603     //  Add names.
604     for(i = 0; i < command->handleNum; i++)
605         CryptDigestUpdate2B(&hashState, &EntityGetName(command->handles[i],
606                                                        &name)->b);
607     //  Complete hash.
608     CryptHashEnd2B(&hashState, &nameHash.b);
```

 Copyright © TCG 2006-2020

```
609         // and compare
610         return MemoryEqual(session->u1.nameHash.t.buffer, nameHash.t.buffer,
611                            nameHash.t.size);
612     }
```

6.4.4.8 CheckPWAuthSession()

This function validates the authorization provided in a PWAP session. It compares the input value to *authValue* of the authorized entity. Argument *sessionIndex* is used to get handles handle of the referenced entities from *s_inputAuthValues*[] and *s_associatedHandles*[].

Error Returns	Meaning
TPM_RC_AUTH_FAIL	authorization fails and increments DA failure count
TPM_RC_BAD_AUTH	authorization fails but DA does not apply

```
613     static TPM_RC
614     CheckPWAuthSession(
615         UINT32           sessionIndex   // IN: index of session to be processed
616         )
617     {
618         TPM2B_AUTH      authValue;
619         TPM_HANDLE      associatedHandle = s_associatedHandles[sessionIndex];
620     //
621         // Strip trailing zeros from the password.
622         MemoryRemoveTrailingZeros(&s_inputAuthValues[sessionIndex]);
623
624         // Get the authValue with trailing zeros removed
625         EntityGetAuthValue(associatedHandle, &authValue);
626
627         // Success if the values are identical.
628         if(MemoryEqual2B(&s_inputAuthValues[sessionIndex].b, &authValue.b))
629         {
630             return TPM_RC_SUCCESS;
631         }
632         else                        // if the digests are not identical
633         {
634             // Invoke DA protection if applicable.
635             return IncrementLockout(sessionIndex);
636         }
637     }
```

6.4.4.9 ComputeCommandHMAC()

This function computes the HMAC for an authorization session in a command.

```
638     static TPM2B_DIGEST *
639     ComputeCommandHMAC(
640         COMMAND         *command,       // IN: primary control structure
641         UINT32           sessionIndex,  // IN: index of session to be processed
642         TPM2B_DIGEST    *hmac           // OUT: authorization HMAC
643         )
644     {
645         TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
646         TPM2B_KEY        key;
647         BYTE             marshalBuffer[sizeof(TPMA_SESSION)];
648         BYTE            *buffer;
649         UINT32           marshalSize;
650         HMAC_STATE       hmacState;
651         TPM2B_NONCE     *nonceDecrypt;
652         TPM2B_NONCE     *nonceEncrypt;
653         SESSION         *session;
```

 Copyright © TCG 2006-2020

```
654  //
655      nonceDecrypt = NULL;
656      nonceEncrypt = NULL;
657
658      // Determine if extra nonceTPM values are going to be required.
659      // If this is the first session (sessionIndex = 0) and it is an authorization
660      // session that uses an HMAC, then check if additional session nonces are to be
661      // included.
662      if(sessionIndex == 0
663         && s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
664      {
665          // If there is a decrypt session and if this is not the decrypt session,
666          // then an extra nonce may be needed.
667          if(s_decryptSessionIndex != UNDEFINED_INDEX
668             && s_decryptSessionIndex != sessionIndex)
669          {
670              // Will add the nonce for the decrypt session.
671              SESSION *decryptSession
672                  = SessionGet(s_sessionHandles[s_decryptSessionIndex]);
673              nonceDecrypt = &decryptSession->nonceTPM;
674          }
675          // Now repeat for the encrypt session.
676          if(s_encryptSessionIndex != UNDEFINED_INDEX
677             && s_encryptSessionIndex != sessionIndex
678             && s_encryptSessionIndex != s_decryptSessionIndex)
679          {
680              // Have to have the nonce for the encrypt session.
681              SESSION *encryptSession
682                  = SessionGet(s_sessionHandles[s_encryptSessionIndex]);
683              nonceEncrypt = &encryptSession->nonceTPM;
684          }
685      }
686
687      // Continue with the HMAC processing.
688      session = SessionGet(s_sessionHandles[sessionIndex]);
689
690      // Generate HMAC key.
691      MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
692
693      // Check if the session has an associated handle and if the associated entity
694      // is the one to which the session is bound. If not, add the authValue of
695      // this entity to the HMAC key.
696      // If the session is bound to the object or the session is a policy session
697      // with no authValue required, do not include the authValue in the HMAC key.
698      // Note: For a policy session, its isBound attribute is CLEARED.
699      //
700      // Include the entity authValue if it is needed
701      if(session->attributes.includeAuth == SET)
702      {
703          TPM2B_AUTH          authValue;
704          // Get the entity authValue with trailing zeros removed
705          EntityGetAuthValue(s_associatedHandles[sessionIndex], &authValue);
706          // add the authValue to the HMAC key
707          MemoryConcat2B(&key.b, &authValue.b, sizeof(key.t.buffer));
708      }
709       // if the HMAC key size is 0, a NULL string HMAC is allowed
710      if(key.t.size == 0
711         && s_inputAuthValues[sessionIndex].t.size == 0)
712      {
713          hmac->t.size = 0;
714          return hmac;
715      }
716      // Start HMAC
717      hmac->t.size = CryptHmacStart2B(&hmacState, session->authHashAlg, &key.b);
718
719          //  Add cpHash
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
720         CryptDigestUpdate2B(&hmacState.hashState,
721                             &ComputeCpHash(command, session->authHashAlg)->b);
722             //  Add nonces as required
723         CryptDigestUpdate2B(&hmacState.hashState, &s_nonceCaller[sessionIndex].b);
724         CryptDigestUpdate2B(&hmacState.hashState, &session->nonceTPM.b);
725         if(nonceDecrypt != NULL)
726             CryptDigestUpdate2B(&hmacState.hashState, &nonceDecrypt->b);
727         if(nonceEncrypt != NULL)
728             CryptDigestUpdate2B(&hmacState.hashState, &nonceEncrypt->b);
729             //  Add sessionAttributes
730         buffer = marshalBuffer;
731         marshalSize = TPMA_SESSION_Marshal(&(s_attributes[sessionIndex]),
732                                            &buffer, NULL);
733         CryptDigestUpdate(&hmacState.hashState, marshalSize, marshalBuffer);
734             // Complete the HMAC computation
735         CryptHmacEnd2B(&hmacState, &hmac->b);
736
737         return hmac;
738     }
```

6.4.4.10 CheckSessionHMAC()

This function checks the HMAC of in a session. It uses ComputeCommandHMAC() to compute the expected HMAC value and then compares the result with the HMAC in the authorization session. The authorization is successful if they are the same.

If the authorizations are not the same, IncrementLockout() is called. It will return TPM_RC_AUTH_FAIL if the failure caused the *failureCount* to increment. Otherwise, it will return TPM_RC_BAD_AUTH.

Error Returns	Meaning
TPM_RC_AUTH_FAIL	authorization failure caused *failureCount* increment
TPM_RC_BAD_AUTH	authorization failure did not cause *failureCount* increment

```
739     static TPM_RC
740     CheckSessionHMAC(
741         COMMAND         *command,       // IN: primary control structure
742         UINT32           sessionIndex   // IN: index of session to be processed
743         )
744     {
745         TPM2B_DIGEST         hmac;           // authHMAC for comparing
746     //
747         // Compute authHMAC
748         ComputeCommandHMAC(command, sessionIndex, &hmac);
749
750         // Compare the input HMAC with the authHMAC computed above.
751         if(!MemoryEqual2B(&s_inputAuthValues[sessionIndex].b, &hmac.b))
752         {
753             // If an HMAC session has a failure, invoke the anti-hammering
754             // if it applies to the authorized entity or the session.
755             // Otherwise, just indicate that the authorization is bad.
756             return IncrementLockout(sessionIndex);
757         }
758         return TPM_RC_SUCCESS;
759     }
```

6.4.4.11 CheckPolicyAuthSession()

This function is used to validate the authorization in a policy session. This function performs the following comparisons to see if a policy authorization is properly provided. The check are:

a) compare *policyDigest* in session with *authPolicy* associated with the entity to be authorized;
b) compare timeout if applicable;
c) compare *commandCode* if applicable;
d) compare *cpHash* if applicable; and
e) see if PCR values have changed since computed.

If all the above checks succeed, the handle is authorized. The order of these comparisons is not important because any failure will result in the same error code.

Error Returns	Meaning
TPM_RC_PCR_CHANGED	PCR value is not current
TPM_RC_POLICY_FAIL	policy session fails
TPM_RC_LOCALITY	command locality is not allowed
TPM_RC_POLICY_CC	CC doesn't match
TPM_RC_EXPIRED	policy session has expired
TPM_RC_PP	PP is required but not asserted
TPM_RC_NV_UNAVAILABLE	NV is not available for write
TPM_RC_NV_RATE	NV is rate limiting

```
760  static TPM_RC
761  CheckPolicyAuthSession(
762      COMMAND          *command,       // IN: primary parsing structure
763      UINT32            sessionIndex   // IN: index of session to be processed
764      )
765  {
766      SESSION              *session;
767      TPM2B_DIGEST          authPolicy;
768      TPMI_ALG_HASH         policyAlg;
769      UINT8                 locality;
770  //
771      // Initialize pointer to the authorization session.
772      session = SessionGet(s_sessionHandles[sessionIndex]);
773
774      // If the command is TPM2_PolicySecret(), make sure that
775      // either password or authValue is required
776      if(command->code == TPM_CC_PolicySecret
777         &&  session->attributes.isPasswordNeeded == CLEAR
778         &&  session->attributes.isAuthValueNeeded == CLEAR)
779          return TPM_RC_MODE;
780      // See if the PCR counter for the session is still valid.
781      if(!SessionPCRValueIsCurrent(session))
782          return TPM_RC_PCR_CHANGED;
783      // Get authPolicy.
784      policyAlg = EntityGetAuthPolicy(s_associatedHandles[sessionIndex],
785                                      &authPolicy);
786      // Compare authPolicy.
787      if(!MemoryEqual2B(&session->u2.policyDigest.b, &authPolicy.b))
788          return TPM_RC_POLICY_FAIL;
789      // Policy is OK so check if the other factors are correct
790
791      // Compare policy hash algorithm.
792      if(policyAlg != session->authHashAlg)
793          return TPM_RC_POLICY_FAIL;
794
795      // Compare timeout.
796      if(session->timeout != 0)
```

```
797         {
798             // Cannot compare time if clock stop advancing.  An TPM_RC_NV_UNAVAILABLE
799             // or TPM_RC_NV_RATE error may be returned here. This doesn't mean that
800             // a new nonce will be created just that, because TPM time can't advance
801             // we can't do time-based operations.
802             RETURN_IF_NV_IS_NOT_AVAILABLE;
803
804             if((session->timeout < g_time)
805                || (session->epoch != g_timeEpoch))
806                 return TPM_RC_EXPIRED;
807         }
808         // If command code is provided it must match
809         if(session->commandCode != 0)
810         {
811             if(session->commandCode != command->code)
812                 return TPM_RC_POLICY_CC;
813         }
814         else
815         {
816             // If command requires a DUP or ADMIN authorization, the session must have
817             // command code set.
818             AUTH_ROLE   role = CommandAuthRole(command->index, sessionIndex);
819             if(role == AUTH_ADMIN || role == AUTH_DUP)
820                 return TPM_RC_POLICY_FAIL;
821         }
822         // Check command locality.
823         {
824             BYTE         sessionLocality[sizeof(TPMA_LOCALITY)];
825             BYTE        *buffer = sessionLocality;
826
827             // Get existing locality setting in canonical form
828             sessionLocality[0] = 0;
829             TPMA_LOCALITY_Marshal(&session->commandLocality, &buffer, NULL);
830
831             // See if the locality has been set
832             if(sessionLocality[0] != 0)
833             {
834                 // If so, get the current locality
835                 locality = _plat__LocalityGet();
836                 if(locality < 5)
837                 {
838                     if(((sessionLocality[0] & (1 << locality)) == 0)
839                        || sessionLocality[0] > 31)
840                         return TPM_RC_LOCALITY;
841                 }
842                 else if(locality > 31)
843                 {
844                     if(sessionLocality[0] != locality)
845                         return TPM_RC_LOCALITY;
846                 }
847                 else
848                 {
849                     // Could throw an assert here but a locality error is just
850                     // as good. It just means that, whatever the locality is, it isn't
851                     // the locality requested so...
852                     return TPM_RC_LOCALITY;
853                 }
854             }
855         } // end of locality check
856         // Check physical presence.
857         if(session->attributes.isPPRequired == SET
858            && !_plat__PhysicalPresenceAsserted())
859             return TPM_RC_PP;
860         // Compare cpHash/nameHash if defined, or if the command requires an ADMIN or
861         // DUP role for this handle.
862         if(session->u1.cpHash.b.size != 0)
```

```
863         {
864             BOOL        OK;
865             if(session->attributes.isCpHashDefined)
866                 // Compare cpHash.
867                 OK = MemoryEqual2B(&session->u1.cpHash.b,
868                                    &ComputeCpHash(command, session->authHashAlg)->b);
869             else if(session->attributes.isTemplateSet)
870                 OK = CompareTemplateHash(command, session);
871             else
872                 OK = CompareNameHash(command, session);
873             if(!OK)
874                 return TPM_RCS_POLICY_FAIL;
875         }
876         if(session->attributes.checkNvWritten)
877         {
878             NV_REF           locator;
879             NV_INDEX        *nvIndex;
880     //
881             // If this is not an NV index, the policy makes no sense so fail it.
882             if(HandleGetType(s_associatedHandles[sessionIndex]) != TPM_HT_NV_INDEX)
883                 return TPM_RC_POLICY_FAIL;
884             // Get the index data
885             nvIndex = NvGetIndexInfo(s_associatedHandles[sessionIndex], &locator);
886
887             // Make sure that the TPMA_WRITTEN_ATTRIBUTE has the desired state
888             if((IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
889                != (session->attributes.nvWrittenState == SET))
890                 return TPM_RC_POLICY_FAIL;
891         }
892         return TPM_RC_SUCCESS;
893     }
```

6.4.4.12 RetrieveSessionData()

This function will unmarshal the sessions in the session area of a command. The values are placed in the arrays that are defined at the beginning of this file. The normal unmarshaling errors are possible.

Error Returns	Meaning
TPM_RC_SUCCSS	unmarshaled without error
TPM_RC_SIZE	the number of bytes unmarshaled is not the same as the value for *authorizationSize* in the command

```
894     static TPM_RC
895     RetrieveSessionData(
896         COMMAND          *command        // IN: main parsing structure for command
897         )
898     {
899         int               i;
900         TPM_RC            result;
901         SESSION          *session;
902         TPMA_SESSION      sessionAttributes;
903         TPM_HT            sessionType;
904         INT32             sessionIndex;
905         TPM_RC            errorIndex;
906     //
907         s_decryptSessionIndex = UNDEFINED_INDEX;
908         s_encryptSessionIndex = UNDEFINED_INDEX;
909         s_auditSessionIndex = UNDEFINED_INDEX;
910
911         for(sessionIndex = 0; command->authSize > 0; sessionIndex++)
912         {
913             errorIndex = TPM_RC_S + g_rcIndex[sessionIndex];
```

```
914
915         // If maximum allowed number of sessions has been parsed, return a size
916         // error with a session number that is larger than the number of allowed
917         // sessions
918         if(sessionIndex == MAX_SESSION_NUM)
919             return TPM_RCS_SIZE + errorIndex;
920         // make sure that the associated handle for each session starts out
921         // unassigned
922         s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
923
924         // First parameter: Session handle.
925         result = TPMI_SH_AUTH_SESSION_Unmarshal(
926             &s_sessionHandles[sessionIndex],
927             &command->parameterBuffer,
928             &command->authSize, TRUE);
929         if(result != TPM_RC_SUCCESS)
930             return result + TPM_RC_S + g_rcIndex[sessionIndex];
931         // Second parameter: Nonce.
932         result = TPM2B_NONCE_Unmarshal(&s_nonceCaller[sessionIndex],
933                                        &command->parameterBuffer,
934                                        &command->authSize);
935         if(result != TPM_RC_SUCCESS)
936             return result + TPM_RC_S + g_rcIndex[sessionIndex];
937         // Third parameter: sessionAttributes.
938         result = TPMA_SESSION_Unmarshal(&s_attributes[sessionIndex],
939                                         &command->parameterBuffer,
940                                         &command->authSize);
941         if(result != TPM_RC_SUCCESS)
942             return result + TPM_RC_S + g_rcIndex[sessionIndex];
943         // Fourth parameter: authValue (PW or HMAC).
944         result = TPM2B_AUTH_Unmarshal(&s_inputAuthValues[sessionIndex],
945                                       &command->parameterBuffer,
946                                       &command->authSize);
947         if(result != TPM_RC_SUCCESS)
948             return result + errorIndex;
949
950         sessionAttributes = s_attributes[sessionIndex];
951         if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
952         {
953             // A PWAP session needs additional processing.
954             //     Can't have any attributes set other than continueSession bit
955             if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, encrypt)
956                || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, decrypt)
957                || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, audit)
958                || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditExclusive)
959                || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditReset))
960                 return TPM_RCS_ATTRIBUTES + errorIndex;
961             //     The nonce size must be zero.
962             if(s_nonceCaller[sessionIndex].t.size != 0)
963                 return TPM_RCS_NONCE + errorIndex;
964             continue;
965         }
966         // For not password sessions...
967         // Find out if the session is loaded.
968         if(!SessionIsLoaded(s_sessionHandles[sessionIndex]))
969             return TPM_RC_REFERENCE_S0 + sessionIndex;
970         sessionType = HandleGetType(s_sessionHandles[sessionIndex]);
971         session = SessionGet(s_sessionHandles[sessionIndex]);
972
973         // Check if the session is an HMAC/policy session.
974         if((session->attributes.isPolicy == SET
975             && sessionType == TPM_HT_HMAC_SESSION)
976            || (session->attributes.isPolicy == CLEAR
977                && sessionType == TPM_HT_POLICY_SESSION))
978             return TPM_RCS_HANDLE + errorIndex;
979         // Check that this handle has not previously been used.
```

 Copyright © TCG 2006-2020

```
980         for(i = 0; i < sessionIndex; i++)
981         {
982             if(s_sessionHandles[i] == s_sessionHandles[sessionIndex])
983                 return TPM_RCS_HANDLE + errorIndex;
984         }
985         // If the session is used for parameter encryption or audit as well, set
986         // the corresponding Indexes.
987
988         // First process decrypt.
989         if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, decrypt))
990         {
991             // Check if the commandCode allows command parameter encryption.
992             if(DecryptSize(command->index) == 0)
993                 return TPM_RCS_ATTRIBUTES + errorIndex;
994             // Encrypt attribute can only appear in one session
995             if(s_decryptSessionIndex != UNDEFINED_INDEX)
996                 return TPM_RCS_ATTRIBUTES + errorIndex;
997             // Can't decrypt if the session's symmetric algorithm is TPM_ALG_NULL
998             if(session->symmetric.algorithm == TPM_ALG_NULL)
999                 return TPM_RCS_SYMMETRIC + errorIndex;
1000            // All checks passed, so set the index for the session used to decrypt
1001            // a command parameter.
1002            s_decryptSessionIndex = sessionIndex;
1003        }
1004        // Now process encrypt.
1005        if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, encrypt))
1006        {
1007            // Check if the commandCode allows response parameter encryption.
1008            if(EncryptSize(command->index) == 0)
1009                return TPM_RCS_ATTRIBUTES + errorIndex;
1010            // Encrypt attribute can only appear in one session.
1011            if(s_encryptSessionIndex != UNDEFINED_INDEX)
1012                return TPM_RCS_ATTRIBUTES + errorIndex;
1013            // Can't encrypt if the session's symmetric algorithm is TPM_ALG_NULL
1014            if(session->symmetric.algorithm == TPM_ALG_NULL)
1015                return TPM_RCS_SYMMETRIC + errorIndex;
1016            // All checks passed, so set the index for the session used to encrypt
1017            // a response parameter.
1018            s_encryptSessionIndex = sessionIndex;
1019        }
1020        // At last process audit.
1021        if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, audit))
1022        {
1023            // Audit attribute can only appear in one session.
1024            if(s_auditSessionIndex != UNDEFINED_INDEX)
1025                return TPM_RCS_ATTRIBUTES + errorIndex;
1026            // An audit session can not be policy session.
1027            if(HandleGetType(s_sessionHandles[sessionIndex])
1028               == TPM_HT_POLICY_SESSION)
1029                return TPM_RCS_ATTRIBUTES + errorIndex;
1030            // If this is a reset of the audit session, or the first use
1031            // of the session as an audit session, it doesn't matter what
1032            // the exclusive state is. The session will become exclusive.
1033            if(!IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditReset)
1034               && session->attributes.isAudit == SET)
1035            {
1036                // Not first use or reset. If auditExlusive is SET, then this
1037                // session must be the current exclusive session.
1038                if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditExclusive)
1039                   && g_exclusiveAuditSession != s_sessionHandles[sessionIndex])
1040                    return TPM_RC_EXCLUSIVE;
1041            }
1042            s_auditSessionIndex = sessionIndex;
1043        }
1044        // Initialize associated handle as undefined. This will be changed when
1045        // the handles are processed.
```

```
1046            s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
1047        }
1048        command->sessionNum = sessionIndex;
1049        return TPM_RC_SUCCESS;
1050    }
```

6.4.4.13 CheckLockedOut()

This function checks to see if the TPM is in lockout. This function should only be called if the entity being checked is subject to DA protection. The TPM is in lockout if the NV is not available and a DA write is pending. Otherwise the TPM is locked out if checking for *lockoutAuth* (*lockoutAuthCheck* == TRUE) and use of *lockoutAuth* is disabled, or *failedTries* >= *maxTries*

Error Returns	Meaning
TPM_RC_NV_RATE	NV is rate limiting
TPM_RC_NV_UNAVAILABLE	NV is not available at this time
TPM_RC_LOCKOUT	TPM is in lockout

```
1051    static TPM_RC
1052    CheckLockedOut(
1053        BOOL             lockoutAuthCheck   // IN: TRUE if checking is for lockoutAuth
1054        )
1055    {
1056        // If NV is unavailable, and current cycle state recorded in NV is not
1057        // SU_NONE_VALUE, refuse to check any authorization because we would
1058        // not be able to handle a DA failure.
1059        if(!NV_IS_AVAILABLE && NV_IS_ORDERLY)
1060            return g_NvStatus;
1061        // Check if DA info needs to be updated in NV.
1062        if(s_DAPendingOnNV)
1063        {
1064            // If NV is accessible,
1065            RETURN_IF_NV_IS_NOT_AVAILABLE;
1066
1067            // ... write the pending DA data and proceed.
1068            NV_SYNC_PERSISTENT(lockOutAuthEnabled);
1069            NV_SYNC_PERSISTENT(failedTries);
1070            s_DAPendingOnNV = FALSE;
1071        }
1072        // Lockout is in effect if checking for lockoutAuth and use of lockoutAuth
1073        // is disabled...
1074        if(lockoutAuthCheck)
1075        {
1076            if(gp.lockOutAuthEnabled == FALSE)
1077                return TPM_RC_LOCKOUT;
1078        }
1079        else
1080        {
1081            // ... or if the number of failed tries has been maxed out.
1082            if(gp.failedTries >= gp.maxTries)
1083                return TPM_RC_LOCKOUT;
1084    #if USE_DA_USED
1085            // If the daUsed flag is not SET, then no DA validation until the
1086            // daUsed state is written to NV
1087            if(!g_daUsed)
1088            {
1089                RETURN_IF_NV_IS_NOT_AVAILABLE;
1090                g_daUsed = TRUE;
1091                gp.orderlyState = SU_DA_USED_VALUE;
1092                NV_SYNC_PERSISTENT(orderlyState);
1093                return TPM_RC_RETRY;
```

```
1094            }
1095    #endif
1096        }
1097        return TPM_RC_SUCCESS;
1098    }
```

6.4.4.14 CheckAuthSession()

This function checks that the authorization session properly authorizes the use of the associated handle.

Error Returns	Meaning
TPM_RC_LOCKOUT	entity is protected by DA and TPM is in lockout, or TPM is locked out on NV update pending on DA parameters
TPM_RC_PP	Physical Presence is required but not provided
TPM_RC_AUTH_FAIL	HMAC or PW authorization failed with DA side-effects (can be a policy session)
TPM_RC_BAD_AUTH	HMAC or PW authorization failed without DA side-effects (can be a policy session)
TPM_RC_POLICY_FAIL	if policy session fails
TPM_RC_POLICY_CC	command code of policy was wrong
TPM_RC_EXPIRED	the policy session has expired
TPM_RC_PCR	???
TPM_RC_AUTH_UNAVAILABLE	*authValue* or *authPolicy* unavailable

```
1099    static TPM_RC
1100    CheckAuthSession(
1101        COMMAND          *command,        // IN: primary parsing structure
1102        UINT32            sessionIndex    // IN: index of session to be processed
1103        )
1104    {
1105        TPM_RC            result = TPM_RC_SUCCESS;
1106        SESSION          *session = NULL;
1107        TPM_HANDLE        sessionHandle = s_sessionHandles[sessionIndex];
1108        TPM_HANDLE        associatedHandle = s_associatedHandles[sessionIndex];
1109        TPM_HT            sessionHandleType = HandleGetType(sessionHandle);
1110        BOOL              authUsed;
1111    //
1112        pAssert(sessionHandle != TPM_RH_UNASSIGNED);
1113
1114        // Take care of physical presence
1115        if(associatedHandle == TPM_RH_PLATFORM)
1116        {
1117            // If the physical presence is required for this command, check for PP
1118            // assertion. If it isn't asserted, no point going any further.
1119            if(PhysicalPresenceIsRequired(command->index)
1120               && !_plat__PhysicalPresenceAsserted())
1121                return TPM_RC_PP;
1122        }
1123        if(sessionHandle != TPM_RS_PW)
1124        {
1125            session = SessionGet(sessionHandle);
1126
1127            // Set includeAuth to indicate if DA checking will be required and if the
1128            // authValue will be included in any HMAC.
1129            if(sessionHandleType == TPM_HT_POLICY_SESSION)
1130            {
1131                // For a policy session, will check the DA status of the entity if either
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1132                // isAuthValueNeeded or isPasswordNeeded is SET.
1133                session->attributes.includeAuth =
1134                    session->attributes.isAuthValueNeeded
1135                    || session->attributes.isPasswordNeeded;
1136            }
1137            else
1138            {
1139                // For an HMAC session, need to check unless the session
1140                // is bound.
1141                session->attributes.includeAuth =
1142                    !IsSessionBindEntity(s_associatedHandles[sessionIndex], session);
1143            }
1144            authUsed = session->attributes.includeAuth;
1145        }
1146        else
1147            // Password session
1148            authUsed = TRUE;
1149        // If the authorization session is going to use an authValue, then make sure
1150        // that access to that authValue isn't locked out.
1151        if(authUsed)
1152        {
1153            // See if entity is subject to lockout.
1154            if(!IsDAExempted(associatedHandle))
1155            {
1156                // See if in lockout
1157                result = CheckLockedOut(associatedHandle == TPM_RH_LOCKOUT);
1158                if(result != TPM_RC_SUCCESS)
1159                    return result;
1160            }
1161        }
1162        // Policy or HMAC+PW?
1163        if(sessionHandleType != TPM_HT_POLICY_SESSION)
1164        {
1165            // for non-policy session make sure that a policy session is not required
1166            if(IsPolicySessionRequired(command->index, sessionIndex))
1167                return TPM_RC_AUTH_TYPE;
1168            // The authValue must be available.
1169            // Note: The authValue is going to be "used" even if it is an EmptyAuth.
1170            // and the session is bound.
1171            if(!IsAuthValueAvailable(associatedHandle, command->index, sessionIndex))
1172                return TPM_RC_AUTH_UNAVAILABLE;
1173        }
1174        else
1175        {
1176            // ... see if the entity has a policy, ...
1177            // Note: IsAuthPolicyAvalable will return FALSE if the sensitive area of the
1178            // object is not loaded
1179            if(!IsAuthPolicyAvailable(associatedHandle, command->index, sessionIndex))
1180                return TPM_RC_AUTH_UNAVAILABLE;
1181            // ... and check the policy session.
1182            result = CheckPolicyAuthSession(command, sessionIndex);
1183            if(result != TPM_RC_SUCCESS)
1184                return result;
1185        }
1186        // Check authorization according to the type
1187        if((TPM_RS_PW == sessionHandle) || (session->attributes.isPasswordNeeded == SET))
1188            result = CheckPWAuthSession(sessionIndex);
1189        else
1190            result = CheckSessionHMAC(command, sessionIndex);
1191        // Do processing for PIN Indexes are only three possibilities for 'result' at
1192        // this point: TPM_RC_SUCCESS, TPM_RC_AUTH_FAIL, and TPM_RC_BAD_AUTH.
1193        // For all these cases, we would have to process a PIN index if the
1194        // authValue of the index was used for authorization.
1195        if((TPM_HT_NV_INDEX == HandleGetType(associatedHandle)) && authUsed)
1196        {
1197            NV_REF           locator;
```

```
1198            NV_INDEX        *nvIndex = NvGetIndexInfo(associatedHandle, &locator);
1199            NV_PIN           pinData;
1200            TPMA_NV          nvAttributes;
1201    //
1202            pAssert(nvIndex != NULL);
1203            nvAttributes = nvIndex->publicArea.attributes;
1204            // If this is a PIN FAIL index and the value has been written
1205            // then we can update the counter (increment or clear)
1206            if(IsNvPinFailIndex(nvAttributes)
1207               && IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN))
1208            {
1209                pinData.intVal = NvGetUINT64Data(nvIndex, locator);
1210                if(result != TPM_RC_SUCCESS)
1211                    pinData.pin.pinCount++;
1212                else
1213                    pinData.pin.pinCount = 0;
1214                NvWriteUINT64Data(nvIndex, pinData.intVal);
1215            }
1216            // If this is a PIN PASS Index, increment if we have used the
1217            // authorization value.
1218            // NOTE: If the counter has already hit the limit, then we
1219            // would not get here because the authorization value would not
1220            // be available and the TPM would have returned before it gets here
1221            else if(IsNvPinPassIndex(nvAttributes)
1222                    && IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN)
1223                    && result == TPM_RC_SUCCESS)
1224            {
1225                // If the access is valid, then increment the use counter
1226                pinData.intVal = NvGetUINT64Data(nvIndex, locator);
1227                pinData.pin.pinCount++;
1228                NvWriteUINT64Data(nvIndex, pinData.intVal);
1229            }
1230        }
1231        return result;
1232    }
1233    #ifdef  TPM_CC_GetCommandAuditDigest
```

6.4.4.15 CheckCommandAudit()

This function is called before the command is processed if audit is enabled for the command. It will check to see if the audit can be performed and will ensure that the *cpHash* is available for the audit.

Error Returns	Meaning
TPM_RC_NV_UNAVAILABLE	NV is not available for write
TPM_RC_NV_RATE	NV is rate limiting

```
1234    static TPM_RC
1235    CheckCommandAudit(
1236        COMMAND         *command
1237        )
1238    {
1239        // If the audit digest is clear and command audit is required, NV must be
1240        // available so that TPM2_GetCommandAuditDigest() is able to increment
1241        // audit counter. If NV is not available, the function bails out to prevent
1242        // the TPM from attempting an operation that would fail anyway.
1243        if(gr.commandAuditDigest.t.size == 0
1244           || GetCommandCode(command->index) == TPM_CC_GetCommandAuditDigest)
1245        {
1246            RETURN_IF_NV_IS_NOT_AVAILABLE;
1247        }
1248        // Make sure that the cpHash is computed for the algorithm
1249        ComputeCpHash(command, gp.auditHashAlg);
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1250        return TPM_RC_SUCCESS;
1251    }
1252    #endif
```

6.4.4.16 ParseSessionBuffer()

This function is the entry function for command session processing. It iterates sessions in session area and reports if the required authorization has been properly provided. It also processes audit session and passes the information of encryption sessions to parameter encryption module.

Error Returns	Meaning
various	parsing failure or authorization failure

```
1253    TPM_RC
1254    ParseSessionBuffer(
1255        COMMAND         *command        // IN: the structure that contains
1256        )
1257    {
1258        TPM_RC                  result;
1259        UINT32                  i;
1260        INT32                   size = 0;
1261        TPM2B_AUTH              extraKey;
1262        UINT32                  sessionIndex;
1263        TPM_RC                  errorIndex;
1264        SESSION                *session = NULL;
1265    //
1266        // Check if a command allows any session in its session area.
1267        if(!IsSessionAllowed(command->index))
1268            return TPM_RC_AUTH_CONTEXT;
1269        // Default-initialization.
1270        command->sessionNum = 0;
1271
1272        result = RetrieveSessionData(command);
1273        if(result != TPM_RC_SUCCESS)
1274            return result;
1275        // There is no command in the TPM spec that has more handles than
1276        // MAX_SESSION_NUM.
1277        pAssert(command->handleNum <= MAX_SESSION_NUM);
1278
1279        // Associate the session with an authorization handle.
1280        for(i = 0; i < command->handleNum; i++)
1281        {
1282            if(CommandAuthRole(command->index, i) != AUTH_NONE)
1283            {
1284                // If the received session number is less than the number of handles
1285                // that requires authorization, an error should be returned.
1286                // Note: for all the TPM 2.0 commands, handles requiring
1287                // authorization come first in a command input and there are only ever
1288                // two values requiring authorization
1289                if(i > (command->sessionNum - 1))
1290                    return TPM_RC_AUTH_MISSING;
1291                // Record the handle associated with the authorization session
1292                s_associatedHandles[i] = command->handles[i];
1293            }
1294        }
1295        // Consistency checks are done first to avoid authorization failure when the
1296        // command will not be executed anyway.
1297        for(sessionIndex = 0; sessionIndex < command->sessionNum; sessionIndex++)
1298        {
1299            errorIndex = TPM_RC_S + g_rcIndex[sessionIndex];
1300            // PW session must be an authorization session
1301            if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
1302            {
```

```
1303                if(s_associatedHandles[sessionIndex] == TPM_RH_UNASSIGNED)
1304                    return TPM_RCS_HANDLE + errorIndex;
1305                // a password session can't be audit, encrypt or decrypt
1306                if(IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit)
1307                   || IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, encrypt)
1308                   || IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, decrypt))
1309                    return TPM_RCS_ATTRIBUTES + errorIndex;
1310                session = NULL;
1311            }
1312            else
1313            {
1314                session = SessionGet(s_sessionHandles[sessionIndex]);
1315
1316                // A trial session can not appear in session area, because it cannot
1317                // be used for authorization, audit or encrypt/decrypt.
1318                if(session->attributes.isTrialPolicy == SET)
1319                    return TPM_RCS_ATTRIBUTES + errorIndex;
1320
1321                // See if the session is bound to a DA protected entity
1322                // NOTE: Since a policy session is never bound, a policy is still
1323                // usable even if the object is DA protected and the TPM is in
1324                // lockout.
1325                if(session->attributes.isDaBound == SET)
1326                {
1327                    result = CheckLockedOut(session->attributes.isLockoutBound == SET);
1328                    if(result != TPM_RC_SUCCESS)
1329                        return result;
1330                }
1331                // If this session is for auditing, make sure the cpHash is computed.
1332                if(IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit))
1333                    ComputeCpHash(command, session->authHashAlg);
1334            }
1335
1336            // if the session has an associated handle, check the authorization
1337            if(s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
1338            {
1339                result = CheckAuthSession(command, sessionIndex);
1340                if(result != TPM_RC_SUCCESS)
1341                    return RcSafeAddToResult(result, errorIndex);
1342            }
1343            else
1344            {
1345                // a session that is not for authorization must either be encrypt,
1346                // decrypt, or audit
1347                if(!IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit)
1348                   &&  !IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, encrypt)
1349                   &&  !IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, decrypt))
1350                    return TPM_RCS_ATTRIBUTES + errorIndex;
1351
1352                // no authValue included in any of the HMAC computations
1353                pAssert(session != NULL);
1354                session->attributes.includeAuth = CLEAR;
1355
1356                // check HMAC for encrypt/decrypt/audit only sessions
1357                result = CheckSessionHMAC(command, sessionIndex);
1358                if(result != TPM_RC_SUCCESS)
1359                    return RcSafeAddToResult(result, errorIndex);
1360            }
1361        }
1362    #ifdef  TPM_CC_GetCommandAuditDigest
1363        // Check if the command should be audited. Need to do this before any parameter
1364        // encryption so that the cpHash for the audit is correct
1365        if(CommandAuditIsRequired(command->index))
1366        {
1367            result = CheckCommandAudit(command);
1368            if(result != TPM_RC_SUCCESS)
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1369                return result;              // No session number to reference
1370        }
1371    #endif
1372        // Decrypt the first parameter if applicable. This should be the last operation
1373        // in session processing.
1374        // If the encrypt session is associated with a handle and the handle's
1375        // authValue is available, then authValue is concatenated with sessionKey to
1376        // generate encryption key, no matter if the handle is the session bound entity
1377        // or not.
1378        if(s_decryptSessionIndex != UNDEFINED_INDEX)
1379        {
1380            // If this is an authorization session, include the authValue in the
1381            // generation of the decryption key
1382            if(s_associatedHandles[s_decryptSessionIndex] != TPM_RH_UNASSIGNED)
1383            {
1384                EntityGetAuthValue(s_associatedHandles[s_decryptSessionIndex],
1385                                   &extraKey);
1386            }
1387            else
1388            {
1389                extraKey.b.size = 0;
1390            }
1391            size = DecryptSize(command->index);
1392            result = CryptParameterDecryption(s_sessionHandles[s_decryptSessionIndex],
1393                                              &s_nonceCaller[s_decryptSessionIndex].b,
1394                                              command->parameterSize, (UINT16)size,
1395                                              &extraKey,
1396                                              command->parameterBuffer);
1397            if(result != TPM_RC_SUCCESS)
1398                return RcSafeAddToResult(result,
1399                                         TPM_RC_S + g_rcIndex[s_decryptSessionIndex]);
1400        }
1401
1402        return TPM_RC_SUCCESS;
1403    }
```

6.4.4.17 CheckAuthNoSession()

Function to process a command with no session associated. The function makes sure all the handles in the command require no authorization.

Error Returns	Meaning
TPM_RC_AUTH_MISSING	failure - one or more handles require authorization

```
1404    TPM_RC
1405    CheckAuthNoSession(
1406        COMMAND          *command        // IN: command parsing structure
1407        )
1408    {
1409        UINT32 i;
1410        TPM_RC           result = TPM_RC_SUCCESS;
1411    //
1412        // Check if the command requires authorization
1413        for(i = 0; i < command->handleNum; i++)
1414        {
1415            if(CommandAuthRole(command->index, i) != AUTH_NONE)
1416                return TPM_RC_AUTH_MISSING;
1417        }
1418    #ifdef  TPM_CC_GetCommandAuditDigest
1419        // Check if the command should be audited.
1420        if(CommandAuditIsRequired(command->index))
1421        {
1422            result = CheckCommandAudit(command);
```

```
1423            if(result != TPM_RC_SUCCESS)
1424                return result;
1425        }
1426    #endif
1427        // Initialize number of sessions to be 0
1428        command->sessionNum = 0;
1429
1430        return TPM_RC_SUCCESS;
1431    }
```

6.4.5 Response Session Processing

6.4.5.1 Introduction

The following functions build the session area in a response and handle the audit sessions (if present).

6.4.5.2 ComputeRpHash()

Function to compute *rpHash* (Response Parameter Hash). The *rpHash* is only computed if there is an HMAC authorization session and the return code is TPM_RC_SUCCESS.

```
1432    static TPM2B_DIGEST *
1433    ComputeRpHash(
1434        COMMAND          *command,       // IN: command structure
1435        TPM_ALG_ID        hashAlg        // IN: hash algorithm to compute rpHash
1436        )
1437    {
1438        TPM2B_DIGEST     *rpHash = GetRpHashPointer(command, hashAlg);
1439        HASH_STATE        hashState;
1440    //
1441        if(rpHash->t.size == 0)
1442        {
1443        //   rpHash := hash(responseCode || commandCode || parameters)
1444
1445        // Initiate hash creation.
1446            rpHash->t.size = CryptHashStart(&hashState, hashAlg);
1447
1448            // Add hash constituents.
1449            CryptDigestUpdateInt(&hashState, sizeof(TPM_RC), TPM_RC_SUCCESS);
1450            CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), command->code);
1451            CryptDigestUpdate(&hashState, command->parameterSize,
1452                              command->parameterBuffer);
1453            // Complete hash computation.
1454            CryptHashEnd2B(&hashState, &rpHash->b);
1455        }
1456        return rpHash;
1457    }
```

6.4.5.3 InitAuditSession()

This function initializes the audit data in an audit session.

```
1458    static void
1459    InitAuditSession(
1460        SESSION          *session        // session to be initialized
1461        )
1462    {
1463        // Mark session as an audit session.
1464        session->attributes.isAudit = SET;
1465
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1466        // Audit session can not be bound.
1467        session->attributes.isBound = CLEAR;
1468
1469        // Size of the audit log is the size of session hash algorithm digest.
1470        session->u2.auditDigest.t.size = CryptHashGetDigestSize(session->authHashAlg);
1471
1472        // Set the original digest value to be 0.
1473        MemorySet(&session->u2.auditDigest.t.buffer,
1474                  0,
1475                  session->u2.auditDigest.t.size);
1476        return;
1477    }
```

6.4.5.4 UpdateAuditDigest

Function to update an audit digest

```
1478    static void
1479    UpdateAuditDigest(
1480        COMMAND          *command,
1481        TPMI_ALG_HASH     hashAlg,
1482        TPM2B_DIGEST     *digest
1483        )
1484    {
1485        HASH_STATE        hashState;
1486        TPM2B_DIGEST     *cpHash = GetCpHash(command, hashAlg);
1487        TPM2B_DIGEST     *rpHash = ComputeRpHash(command, hashAlg);
1488    //
1489        pAssert(cpHash != NULL);
1490
1491        // digestNew :=  hash (digestOld || cpHash || rpHash)
1492            // Start hash computation.
1493        digest->t.size = CryptHashStart(&hashState, hashAlg);
1494            // Add old digest.
1495        CryptDigestUpdate2B(&hashState, &digest->b);
1496            // Add cpHash
1497        CryptDigestUpdate2B(&hashState, &cpHash->b);
1498            // Add rpHash
1499        CryptDigestUpdate2B(&hashState, &rpHash->b);
1500            // Finalize the hash.
1501        CryptHashEnd2B(&hashState, &digest->b);
1502    }
```

6.4.5.5 Audit()

This function updates the audit digest in an audit session.

```
1503    static void
1504    Audit(
1505        COMMAND          *command,       // IN: primary control structure
1506        SESSION          *auditSession   // IN: loaded audit session
1507        )
1508    {
1509        UpdateAuditDigest(command, auditSession->authHashAlg,
1510                          &auditSession->u2.auditDigest);
1511        return;
1512    }
1513    #ifdef  TPM_CC_GetCommandAuditDigest
```

6.4.5.6 CommandAudit()

This function updates the command audit digest.

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
1514 static void
1515 CommandAudit(
1516     COMMAND           *command         // IN:
1517     )
1518 {
1519     // If the digest.size is one, it indicates the special case of changing
1520     // the audit hash algorithm. For this case, no audit is done on exit.
1521     // NOTE: When the hash algorithm is changed, g_updateNV is set in order to
1522     // force an update to the NV on exit so that the change in digest will
1523     // be recorded. So, it is safe to exit here without setting any flags
1524     // because the digest change will be written to NV when this code exits.
1525     if(gr.commandAuditDigest.t.size == 1)
1526     {
1527         gr.commandAuditDigest.t.size = 0;
1528         return;
1529     }
1530     // If the digest size is zero, need to start a new digest and increment
1531     // the audit counter.
1532     if(gr.commandAuditDigest.t.size == 0)
1533     {
1534         gr.commandAuditDigest.t.size = CryptHashGetDigestSize(gp.auditHashAlg);
1535         MemorySet(gr.commandAuditDigest.t.buffer,
1536                   0,
1537                   gr.commandAuditDigest.t.size);
1538
1539         // Bump the counter and save its value to NV.
1540         gp.auditCounter++;
1541         NV_SYNC_PERSISTENT(auditCounter);
1542     }
1543     UpdateAuditDigest(command, gp.auditHashAlg, &gr.commandAuditDigest);
1544     return;
1545 }
1546 #endif
```

6.4.5.7 UpdateAuditSessionStatus()

Function to update the internal audit related states of a session. It

a) initializes the session as audit session and sets it to be exclusive if this is the first time it is used for audit or audit reset was requested;

b) reports exclusive audit session;

c) extends audit log; and

d) clears exclusive audit session if no audit session found in the command.

```
1547 static void
1548 UpdateAuditSessionStatus(
1549     COMMAND           *command         // IN: primary control structure
1550     )
1551 {
1552     UINT32             i;
1553     TPM_HANDLE         auditSession = TPM_RH_UNASSIGNED;
1554 //
1555     // Iterate through sessions
1556     for(i = 0; i < command->sessionNum; i++)
1557     {
1558         SESSION      *session;
1559 //
1560         // PW session do not have a loaded session and can not be an audit
1561         // session either.  Skip it.
1562         if(s_sessionHandles[i] == TPM_RS_PW)
1563             continue;
1564         session = SessionGet(s_sessionHandles[i]);
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1565
1566            // If a session is used for audit
1567            if(IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, audit))
1568            {
1569                // An audit session has been found
1570                auditSession = s_sessionHandles[i];
1571
1572                // If the session has not been an audit session yet, or
1573                // the auditSetting bits indicate a reset, initialize it and set
1574                // it to be the exclusive session
1575                if(session->attributes.isAudit == CLEAR
1576                   || IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, auditReset))
1577                {
1578                    InitAuditSession(session);
1579                    g_exclusiveAuditSession = auditSession;
1580                }
1581                else
1582                {
1583                    // Check if the audit session is the current exclusive audit
1584                    // session and, if not, clear previous exclusive audit session.
1585                    if(g_exclusiveAuditSession != auditSession)
1586                        g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1587                }
1588                // Report audit session exclusivity.
1589                if(g_exclusiveAuditSession == auditSession)
1590                {
1591                    SET_ATTRIBUTE(s_attributes[i], TPMA_SESSION, auditExclusive);
1592                }
1593                else
1594                {
1595                    CLEAR_ATTRIBUTE(s_attributes[i], TPMA_SESSION, auditExclusive);
1596                }
1597                // Extend audit log.
1598                Audit(command, session);
1599            }
1600        }
1601        // If no audit session is found in the command, and the command allows
1602        // a session then, clear the current exclusive
1603        // audit session.
1604        if(auditSession == TPM_RH_UNASSIGNED && IsSessionAllowed(command->index))
1605        {
1606            g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1607        }
1608        return;
1609    }
```

6.4.5.8 ComputeResponseHMAC()

Function to compute HMAC for authorization session in a response.

```
1610    static void
1611    ComputeResponseHMAC(
1612        COMMAND         *command,         // IN: command structure
1613        UINT32           sessionIndex,    // IN: session index to be processed
1614        SESSION         *session,         // IN: loaded session
1615        TPM2B_DIGEST    *hmac             // OUT: authHMAC
1616        )
1617    {
1618        TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
1619        TPM2B_KEY        key;       // HMAC key
1620        BYTE             marshalBuffer[sizeof(TPMA_SESSION)];
1621        BYTE            *buffer;
1622        UINT32           marshalSize;
1623        HMAC_STATE       hmacState;
```

```
1624        TPM2B_DIGEST    *rpHash = ComputeRpHash(command, session->authHashAlg);
1625    //
1626        // Generate HMAC key
1627        MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
1628
1629        // Add the object authValue if required
1630        if(session->attributes.includeAuth == SET)
1631        {
1632            // Note: includeAuth may be SET for a policy that is used in
1633            // UndefineSpaceSpecial(). At this point, the Index has been deleted
1634            // so the includeAuth will have no meaning. However, the
1635            // s_associatedHandles[] value for the session is now set to TPM_RH_NULL so
1636            // this will return the authValue associated with TPM_RH_NULL and that is
1637            // and empty buffer.
1638            TPM2B_AUTH          authValue;
1639    //
1640            // Get the authValue with trailing zeros removed
1641            EntityGetAuthValue(s_associatedHandles[sessionIndex], &authValue);
1642
1643            // Add it to the key
1644            MemoryConcat2B(&key.b, &authValue.b, sizeof(key.t.buffer));
1645        }
1646
1647        // if the HMAC key size is 0, the response HMAC is computed according to the
1648        // input HMAC
1649        if(key.t.size == 0
1650           && s_inputAuthValues[sessionIndex].t.size == 0)
1651        {
1652            hmac->t.size = 0;
1653            return;
1654        }
1655        // Start HMAC computation.
1656        hmac->t.size = CryptHmacStart2B(&hmacState, session->authHashAlg, &key.b);
1657
1658        // Add hash components.
1659        CryptDigestUpdate2B(&hmacState.hashState, &rpHash->b);
1660        CryptDigestUpdate2B(&hmacState.hashState, &session->nonceTPM.b);
1661        CryptDigestUpdate2B(&hmacState.hashState, &s_nonceCaller[sessionIndex].b);
1662
1663        // Add session attributes.
1664        buffer = marshalBuffer;
1665        marshalSize = TPMA_SESSION_Marshal(&s_attributes[sessionIndex], &buffer, NULL);
1666        CryptDigestUpdate(&hmacState.hashState, marshalSize, marshalBuffer);
1667
1668        // Finalize HMAC.
1669        CryptHmacEnd2B(&hmacState, &hmac->b);
1670
1671        return;
1672    }
```

6.4.5.9 UpdateInternalSession()

Updates internal sessions:

a) Restarts session time.

b) Clears a policy session since nonce is rolling.

```
1673    static void
1674    UpdateInternalSession(
1675        SESSION         *session,       // IN: the session structure
1676        UINT32           i              // IN: session number
1677        )
1678    {
1679        // If nonce is rolling in a policy session, the policy related data
```


 Copyright © TCG 2006-2020

```
1680        // will be re-initialized.
1681        if(HandleGetType(s_sessionHandles[i]) == TPM_HT_POLICY_SESSION
1682           && IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession))
1683        {
1684            // When the nonce rolls it starts a new timing interval for the
1685            // policy session.
1686            SessionResetPolicyData(session);
1687            SessionSetStartTime(session);
1688        }
1689        return;
1690    }
```

6.4.5.10 BuildSingleResponseAuth()

Function to compute response HMAC value for a policy or HMAC session.

```
1691    static TPM2B_NONCE *
1692    BuildSingleResponseAuth(
1693        COMMAND          *command,         // IN: command structure
1694        UINT32            sessionIndex,    // IN: session index to be processed
1695        TPM2B_AUTH       *auth             // OUT: authHMAC
1696        )
1697    {
1698        // Fill in policy/HMAC based session response.
1699        SESSION     *session = SessionGet(s_sessionHandles[sessionIndex]);
1700    //
1701        // If the session is a policy session with isPasswordNeeded SET, the
1702        // authorization field is empty.
1703        if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
1704           && session->attributes.isPasswordNeeded == SET)
1705            auth->t.size = 0;
1706        else
1707            // Compute response HMAC.
1708            ComputeResponseHMAC(command, sessionIndex, session, auth);
1709
1710        UpdateInternalSession(session, sessionIndex);
1711        return &session->nonceTPM;
1712    }
```

6.4.5.11 UpdateAllNonceTPM()

Updates TPM nonce for all sessions in command.

```
1713    static void
1714    UpdateAllNonceTPM(
1715        COMMAND          *command         // IN: controlling structure
1716        )
1717    {
1718        UINT32      i;
1719        SESSION     *session;
1720    //
1721        for(i = 0; i < command->sessionNum; i++)
1722        {
1723            // If not a PW session, compute the new nonceTPM.
1724            if(s_sessionHandles[i] != TPM_RS_PW)
1725            {
1726                session = SessionGet(s_sessionHandles[i]);
1727                // Update nonceTPM in both internal session and response.
1728                CryptRandomGenerate(session->nonceTPM.t.size,
1729                                    session->nonceTPM.t.buffer);
1730            }
1731        }
1732        return;
```

```
1733 }
```

6.4.5.12 BuildResponseSession()

Function to build Session buffer in a response. The authorization data is added to the end of command->*responseBuffer*. The size of the authorization area is accumulated in command->*authSize*. When this is called, command->*responseBuffer* is pointing at the next location in the response buffer to be filled. This is where the authorization sessions will go, if any. command->*parameterSize* is the number of bytes that have been marshaled as parameters in the output buffer.

```
1734 void
1735 BuildResponseSession(
1736     COMMAND         *command        // IN: structure that has relevant command
1737                                     //     information
1738     )
1739 {
1740     pAssert(command->authSize == 0);
1741
1742     // Reset the parameter buffer to point to the start of the parameters so that
1743     // there is a starting point for any rpHash that might be generated and so there
1744     // is a place where parameter encryption would start
1745     command->parameterBuffer = command->responseBuffer - command->parameterSize;
1746
1747     // Session nonces should be updated before parameter encryption
1748     if(command->tag == TPM_ST_SESSIONS)
1749     {
1750         UpdateAllNonceTPM(command);
1751
1752         // Encrypt first parameter if applicable. Parameter encryption should
1753         // happen after nonce update and before any rpHash is computed.
1754         // If the encrypt session is associated with a handle, the authValue of
1755         // this handle will be concatenated with sessionKey to generate
1756         // encryption key, no matter if the handle is the session bound entity
1757         // or not. The authValue is added to sessionKey only when the authValue
1758         // is available.
1759         if(s_encryptSessionIndex != UNDEFINED_INDEX)
1760         {
1761             UINT32          size;
1762             TPM2B_AUTH      extraKey;
1763 //
1764             extraKey.b.size = 0;
1765             // If this is an authorization session, include the authValue in the
1766             // generation of the encryption key
1767             if(s_associatedHandles[s_encryptSessionIndex] != TPM_RH_UNASSIGNED)
1768             {
1769                 EntityGetAuthValue(s_associatedHandles[s_encryptSessionIndex],
1770                                    &extraKey);
1771             }
1772             size = EncryptSize(command->index);
1773             CryptParameterEncryption(s_sessionHandles[s_encryptSessionIndex],
1774                                      &s_nonceCaller[s_encryptSessionIndex].b,
1775                                      (UINT16)size,
1776                                      &extraKey,
1777                                      command->parameterBuffer);
1778         }
1779     }
1780     // Audit sessions should be processed regardless of the tag because
1781     // a command with no session may cause a change of the exclusivity state.
1782     UpdateAuditSessionStatus(command);
1783 #if CC_GetCommandAuditDigest
1784     // Command Audit
1785     if(CommandAuditIsRequired(command->index))
1786         CommandAudit(command);
1787 #endif
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1788      // Process command with sessions.
1789      if(command->tag == TPM_ST_SESSIONS)
1790      {
1791          UINT32           i;
1792  //
1793          pAssert(command->sessionNum > 0);
1794
1795          // Iterate over each session in the command session area, and create
1796          // corresponding sessions for response.
1797          for(i = 0; i < command->sessionNum; i++)
1798          {
1799              TPM2B_NONCE     *nonceTPM;
1800              TPM2B_DIGEST     responseAuth;
1801              // Make sure that continueSession is SET on any Password session.
1802              // This makes it marginally easier for the management software
1803              // to keep track of the closed sessions.
1804              if(s_sessionHandles[i] == TPM_RS_PW)
1805              {
1806                  SET_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession);
1807                  responseAuth.t.size = 0;
1808                  nonceTPM = (TPM2B_NONCE *)&responseAuth;
1809              }
1810              else
1811              {
1812                  // Compute the response HMAC and get a pointer to the nonce used.
1813                  // This function will also update the values if needed. Note, the
1814                  nonceTPM = BuildSingleResponseAuth(command, i, &responseAuth);
1815              }
1816              command->authSize += TPM2B_NONCE_Marshal(nonceTPM,
1817                                                       &command->responseBuffer,
1818                                                       NULL);
1819              command->authSize += TPMA_SESSION_Marshal(&s_attributes[i],
1820                                                        &command->responseBuffer,
1821                                                        NULL);
1822              command->authSize += TPM2B_DIGEST_Marshal(&responseAuth,
1823                                                        &command->responseBuffer,
1824                                                        NULL);
1825              if(!IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession))
1826                  SessionFlush(s_sessionHandles[i]);
1827          }
1828      }
1829      return;
1830  }
```

6.4.5.13 SessionRemoveAssociationToHandle()

This function deals with the case where an entity associated with an authorization is deleted during command processing. The primary use of this is to support UndefineSpaceSpecial().

```
1831  void
1832  SessionRemoveAssociationToHandle(
1833      TPM_HANDLE       handle
1834      )
1835  {
1836      UINT32               i;
1837  //
1838      for(i = 0; i < MAX_SESSION_NUM; i++)
1839      {
1840          if(s_associatedHandles[i] == handle)
1841          {
1842              s_associatedHandles[i] = TPM_RH_NULL;
1843          }
1844      }
1845  }
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

7 Command Support Functions

7.1 Introduction

This clause contains support routines that are called by the command action code in TPM 2.0 Part 3. The functions are grouped by the command group that is supported by the functions.

7.2 Attestation Command Support (Attest_spt.c)

7.2.1 Includes

```
1  #include "Tpm.h"
2  #include "Attest_spt_fp.h"
```

7.2.2 Functions

7.2.2.1 FillInAttestInfo()

Fill in common fields of TPMS_ATTEST structure.

```
3  void
4  FillInAttestInfo(
5      TPMI_DH_OBJECT       signHandle,     // IN: handle of signing object
6      TPMT_SIG_SCHEME     *scheme,         // IN/OUT: scheme to be used for signing
7      TPM2B_DATA          *data,           // IN: qualifying data
8      TPMS_ATTEST         *attest          // OUT: attest structure
9      )
10 {
11     OBJECT              *signObject = HandleToObject(signHandle);
12
13         // Magic number
14     attest->magic = TPM_GENERATED_VALUE;
15
16     if(signObject == NULL)
17     {
18         // The name for a null handle is TPM_RH_NULL
19         // This is defined because UINT32_TO_BYTE_ARRAY does a cast. If the
20         // size of the cast is smaller than a constant, the compiler warns
21         // about the truncation of a constant value.
22         TPM_HANDLE      nullHandle = TPM_RH_NULL;
23         attest->qualifiedSigner.t.size = sizeof(TPM_HANDLE);
24         UINT32_TO_BYTE_ARRAY(nullHandle, attest->qualifiedSigner.t.name);
25     }
26     else
27     {
28         // Certifying object qualified name
29         // if the scheme is anonymous, this is an empty buffer
30         if(CryptIsSchemeAnonymous(scheme->scheme))
31             attest->qualifiedSigner.t.size = 0;
32         else
33             attest->qualifiedSigner = signObject->qualifiedName;
34     }
35     // current clock in plain text
36     TimeFillInfo(&attest->clockInfo);
37
38     // Firmware version in plain text
39     attest->firmwareVersion = ((UINT64)gp.firmwareV1 << (sizeof(UINT32) * 8));
40     attest->firmwareVersion += gp.firmwareV2;
41
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
42      // Check the hierarchy of sign object.  For NULL sign handle, the hierarchy
43      // will be TPM_RH_NULL
44      if((signObject == NULL)
45         || (!signObject->attributes.epsHierarchy
46             && !signObject->attributes.ppsHierarchy))
47      {
48          // For signing key that is not in platform or endorsement hierarchy,
49          // obfuscate the reset, restart and firmware version information
50          UINT64          obfuscation[2];
51          CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &gp.shProof.b, OBFUSCATE_STRING,
52                    &attest->qualifiedSigner.b, NULL, 128,
53                    (BYTE *)&obfuscation[0], NULL, FALSE);
54          // Obfuscate data
55          attest->firmwareVersion += obfuscation[0];
56          attest->clockInfo.resetCount += (UINT32)(obfuscation[1] >> 32);
57          attest->clockInfo.restartCount += (UINT32)obfuscation[1];
58      }
59      // External data
60      if(CryptIsSchemeAnonymous(scheme->scheme))
61          attest->extraData.t.size = 0;
62      else
63      {
64          // If we move the data to the attestation structure, then it is not
65          // used in the signing operation except as part of the signed data
66          attest->extraData = *data;
67          data->t.size = 0;
68      }
69  }
```

7.2.2.2 SignAttestInfo()

Sign a TPMS_ATTEST structure. If *signHandle* is TPM_RH_NULL, a null signature is returned.

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*signHandle* references not a signing key
TPM_RC_SCHEME	*scheme* is not compatible with *signHandle* type
TPM_RC_VALUE	digest generated for the given *scheme* is greater than the modulus of *signHandle* (for an RSA key); invalid commit status or failed to generate r value (for an ECC key)

```
70  TPM_RC
71  SignAttestInfo(
72      OBJECT              *signKey,           // IN: sign object
73      TPMT_SIG_SCHEME     *scheme,            // IN: sign scheme
74      TPMS_ATTEST         *certifyInfo,       // IN: the data to be signed
75      TPM2B_DATA          *qualifyingData,    // IN: extra data for the signing
76                                              //     process
77      TPM2B_ATTEST        *attest,            // OUT: marshaled attest blob to be
78                                              //     signed
79      TPMT_SIGNATURE      *signature          // OUT: signature
80      )
81  {
82      BYTE                    *buffer;
83      HASH_STATE              hashState;
84      TPM2B_DIGEST            digest;
85      TPM_RC                  result;
86
87      // Marshal TPMS_ATTEST structure for hash
88      buffer = attest->t.attestationData;
89      attest->t.size = TPMS_ATTEST_Marshal(certifyInfo, &buffer, NULL);
90
91      if(signKey == NULL)
```

```
 92         {
 93             signature->sigAlg = TPM_ALG_NULL;
 94             result = TPM_RC_SUCCESS;
 95         }
 96         else
 97         {
 98             TPMI_ALG_HASH           hashAlg;
 99             // Compute hash
100             hashAlg = scheme->details.any.hashAlg;
101             // need to set the receive buffer to get something put in it
102             digest.t.size = sizeof(digest.t.buffer);
103             digest.t.size = CryptHashBlock(hashAlg, attest->t.size,
104                                            attest->t.attestationData,
105                                            digest.t.size, digest.t.buffer);
106             // If there is qualifying data, need to rehash the data
107             // hash(qualifyingData || hash(attestationData))
108             if(qualifyingData->t.size != 0)
109             {
110                 CryptHashStart(&hashState, hashAlg);
111                 CryptDigestUpdate2B(&hashState, &qualifyingData->b);
112                 CryptDigestUpdate2B(&hashState, &digest.b);
113                 CryptHashEnd2B(&hashState, &digest.b);
114             }
115             // Sign the hash. A TPM_RC_VALUE, TPM_RC_SCHEME, or
116             // TPM_RC_ATTRIBUTES error may be returned at this point
117             result = CryptSign(signKey, scheme, &digest, signature);
118
119             // Since the clock is used in an attestation, the state in NV is no longer
120             // "orderly" with respect to the data in RAM if the signature is valid
121             if(result == TPM_RC_SUCCESS)
122             {
123                 // Command uses the clock so need to clear the orderly state if it is
124                 // set.
125                 result = NvClearOrderly();
126             }
127         }
128         return result;
129     }
```

7.2.2.3 IsSigningObject()

Checks to see if the object is OK for signing. This is here rather than in Object_spt.c because all the attestation commands use this file but not Object_spt.c.

Return Value	Meaning
TRUE(1)	object may sign
FALSE(0)	object may not sign

```
130     BOOL
131     IsSigningObject(
132         OBJECT          *object         // IN:
133         )
134     {
135         return ((object == NULL)
136                 || ((IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, sign)
137                    && object->publicArea.type != TPM_ALG_SYMCIPHER)));
138     }
```

7.3 Context Management Command Support (Context_spt.c)

7.3.1 Includes

```
1  #include "Tpm.h"
2  #include "Context_spt_fp.h"
```

7.3.2 Functions

7.3.2.1 ComputeContextProtectionKey()

This function retrieves the symmetric protection key for context encryption It is used by TPM2_ConextSave() and TPM2_ContextLoad() to create the symmetric encryption key and iv

```
3  void
4  ComputeContextProtectionKey(
5      TPMS_CONTEXT     *contextBlob,    // IN: context blob
6      TPM2B_SYM_KEY    *symKey,         // OUT: the symmetric key
7      TPM2B_IV         *iv              // OUT: the IV.
8      )
9  {
10     UINT16            symKeyBits;     // number of bits in the parent's
11                                       //   symmetric key
12     TPM2B_PROOF      *proof = NULL;   // the proof value to use. Is null for
13                                       //   everything but a primary object in
14                                       //   the Endorsement Hierarchy
15
16     BYTE              kdfResult[sizeof(TPMU_HA) * 2];// Value produced by the KDF
17
18     TPM2B_DATA        sequence2B, handle2B;
19
20     // Get proof value
21     proof = HierarchyGetProof(contextBlob->hierarchy);
22
23     // Get sequence value in 2B format
24     sequence2B.t.size = sizeof(contextBlob->sequence);
25     cAssert(sizeof(contextBlob->sequence) <= sizeof(sequence2B.t.buffer));
26     MemoryCopy(sequence2B.t.buffer, &contextBlob->sequence,
27                sizeof(contextBlob->sequence));
28
29     // Get handle value in 2B format
30     handle2B.t.size = sizeof(contextBlob->savedHandle);
31     cAssert(sizeof(contextBlob->savedHandle) <= sizeof(handle2B.t.buffer));
32     MemoryCopy(handle2B.t.buffer, &contextBlob->savedHandle,
33                sizeof(contextBlob->savedHandle));
34
35     // Get the symmetric encryption key size
36     symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
37     symKeyBits = CONTEXT_ENCRYPT_KEY_BITS;
38     // Get the size of the IV for the algorithm
39     iv->t.size = CryptGetSymmetricBlockSize(CONTEXT_ENCRYPT_ALG, symKeyBits);
40
41     // KDFa to generate symmetric key and IV value
42     CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &proof->b, CONTEXT_KEY, &sequence2B.b,
43               &handle2B.b, (symKey->t.size + iv->t.size) * 8, kdfResult, NULL,
44               FALSE);
45
46         // Copy part of the returned value as the key
47     pAssert(symKey->t.size <= sizeof(symKey->t.buffer));
48     MemoryCopy(symKey->t.buffer, kdfResult, symKey->t.size);
49
```

```
50      // Copy the rest as the IV
51      pAssert(iv->t.size <= sizeof(iv->t.buffer));
52      MemoryCopy(iv->t.buffer, &kdfResult[symKey->t.size], iv->t.size);
53
54      return;
55  }
```

7.3.2.2 ComputeContextIntegrity()

Generate the integrity hash for a context It is used by TPM2_ContextSave() to create an integrity hash and by TPM2_ContextLoad() to compare an integrity hash

```
56  void
57  ComputeContextIntegrity(
58      TPMS_CONTEXT    *contextBlob,    // IN: context blob
59      TPM2B_DIGEST    *integrity       // OUT: integrity
60      )
61  {
62      HMAC_STATE          hmacState;
63      TPM2B_PROOF        *proof;
64      UINT16              integritySize;
65
66      // Get proof value
67      proof = HierarchyGetProof(contextBlob->hierarchy);
68
69      // Start HMAC
70      integrity->t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
71                                           &proof->b);
72
73      // Compute integrity size at the beginning of context blob
74      integritySize = sizeof(integrity->t.size) + integrity->t.size;
75
76      // Adding total reset counter so that the context cannot be
77      // used after a TPM Reset
78      CryptDigestUpdateInt(&hmacState.hashState, sizeof(gp.totalResetCount),
79                           gp.totalResetCount);
80
81      // If this is a ST_CLEAR object, add the clear count
82      // so that this contest cannot be loaded after a TPM Restart
83      if(contextBlob->savedHandle == 0x80000002)
84          CryptDigestUpdateInt(&hmacState.hashState, sizeof(gr.clearCount),
85                               gr.clearCount);
86
87      // Adding sequence number to the HMAC to make sure that it doesn't
88      // get changed
89      CryptDigestUpdateInt(&hmacState.hashState, sizeof(contextBlob->sequence),
90                           contextBlob->sequence);
91
92      // Protect the handle
93      CryptDigestUpdateInt(&hmacState.hashState, sizeof(contextBlob->savedHandle),
94                           contextBlob->savedHandle);
95
96      // Adding sensitive contextData, skip the leading integrity area
97      CryptDigestUpdate(&hmacState.hashState,
98                        contextBlob->contextBlob.t.size - integritySize,
99                        contextBlob->contextBlob.t.buffer + integritySize);
100
101     // Complete HMAC
102     CryptHmacEnd2B(&hmacState, &integrity->b);
103
104     return;
105 }
```

 Copyright © TCG 2006-2020

7.3.2.3 SequenceDataExport()

This function is used scan through the sequence object and either modify the hash state data for export (*contextSave*) or to import it into the internal format (*contextLoad*). This function should only be called after the sequence object has been copied to the context buffer (*contextSave*) or from the context buffer into the sequence object. The presumption is that the context buffer version of the data is the same size as the internal representation so nothing outsize of the hash context area gets modified.

```
106  void
107  SequenceDataExport(
108      HASH_OBJECT          *object,         // IN: an internal hash object
109      HASH_OBJECT_BUFFER   *exportObject    // OUT: a sequence context in a buffer
110      )
111  {
112      // If the hash object is not an event, then only one hash context is needed
113      int                   count = (object->attributes.eventSeq) ? HASH_COUNT : 1;
114
115      for(count--; count >= 0; count--)
116      {
117          HASH_STATE          *hash = &object->state.hashState[count];
118          size_t               offset = (BYTE *)hash - (BYTE *)object;
119          BYTE                *exportHash = &((BYTE *)exportObject)[offset];
120
121          CryptHashExportState(hash, (EXPORT_HASH_STATE *)exportHash);
122      }
123  }
```

7.3.2.4 SequenceDataImport()

This function is used scan through the sequence object and either modify the hash state data for export (*contextSave*) or to import it into the internal format (*contextLoad*). This function should only be called after the sequence object has been copied to the context buffer (*contextSave*) or from the context buffer into the sequence object. The presumption is that the context buffer version of the data is the same size as the internal representation so nothing outsize of the hash context area gets modified.

```
124  void
125  SequenceDataImport(
126      HASH_OBJECT          *object,         // IN/OUT: an internal hash object
127      HASH_OBJECT_BUFFER   *exportObject    // IN/OUT: a sequence context in a buffer
128      )
129  {
130      // If the hash object is not an event, then only one hash context is needed
131      int                   count = (object->attributes.eventSeq) ? HASH_COUNT : 1;
132
133      for(count--; count >= 0; count--)
134      {
135          HASH_STATE          *hash = &object->state.hashState[count];
136          size_t               offset = (BYTE *)hash - (BYTE *)object;
137          BYTE                *importHash = &((BYTE *)exportObject)[offset];
138  //
139          CryptHashImportState(hash, (EXPORT_HASH_STATE *)importHash);
140      }
141  }
```


 Copyright © TCG 2006-2020

7.4 Policy Command Support (Policy_spt.c)

7.4.1 Includes

```
1  #include "Tpm.h"
2  #include "Policy_spt_fp.h"
3  #include "PolicySigned_fp.h"
4  #include "PolicySecret_fp.h"
5  #include "PolicyTicket_fp.h"
```

7.4.2 Functions

7.4.2.1 PolicyParameterChecks()

This function validates the common parameters of TPM2_PolicySiged() and TPM2_PolicySecret(). The common parameters are *nonceTPM*, *expiration*, and *cpHashA*.

```
6  TPM_RC
7  PolicyParameterChecks(
8      SESSION          *session,
9      UINT64            authTimeout,
10     TPM2B_DIGEST     *cpHashA,
11     TPM2B_NONCE      *nonce,
12     TPM_RC            blameNonce,
13     TPM_RC            blameCpHash,
14     TPM_RC            blameExpiration
15     )
16 {
17     // Validate that input nonceTPM is correct if present
18     if(nonce != NULL && nonce->t.size != 0)
19     {
20         if(!MemoryEqual2B(&nonce->b, &session->nonceTPM.b))
21             return TPM_RCS_NONCE + blameNonce;
22     }
23     // If authTimeout is set (expiration != 0...
24     if(authTimeout != 0)
25     {
26         // Validate input expiration.
27         // Cannot compare time if clock stop advancing.  A TPM_RC_NV_UNAVAILABLE
28         // or TPM_RC_NV_RATE error may be returned here.
29         RETURN_IF_NV_IS_NOT_AVAILABLE;
30
31         // if the time has already passed or the time epoch has changed then the
32         // time value is no longer good.
33         if((authTimeout < g_time)
34            || (session->epoch != g_timeEpoch))
35             return TPM_RCS_EXPIRED + blameExpiration;
36     }
37     // If the cpHash is present, then check it
38     if(cpHashA != NULL && cpHashA->t.size != 0)
39     {
40         // The cpHash input has to have the correct size
41         if(cpHashA->t.size != session->u2.policyDigest.t.size)
42             return TPM_RCS_SIZE + blameCpHash;
43
44         // If the cpHash has already been set, then this input value
45         // must match the current value.
46         if(session->u1.cpHash.b.size != 0
47            && !MemoryEqual2B(&cpHashA->b, &session->u1.cpHash.b))
48             return TPM_RC_CPHASH;
49     }
```

```
50      return TPM_RC_SUCCESS;
51  }
```

7.4.2.2 PolicyContextUpdate()

Update policy hash Update the *policyDigest* in policy session by extending *policyRef* and *objectName* to it. This will also update the *cpHash* if it is present.

```
52  void
53  PolicyContextUpdate(
54      TPM_CC           commandCode,   // IN: command code
55      TPM2B_NAME      *name,          // IN: name of entity
56      TPM2B_NONCE     *ref,           // IN: the reference data
57      TPM2B_DIGEST    *cpHash,        // IN: the cpHash (optional)
58      UINT64           policyTimeout, // IN: the timeout value for the policy
59      SESSION         *session        // IN/OUT: policy session to be updated
60      )
61  {
62      HASH_STATE           hashState;
63
64      // Start hash
65     CryptHashStart(&hashState, session->authHashAlg);
66
67      // policyDigest size should always be the digest size of session hash algorithm.
68      pAssert(session->u2.policyDigest.t.size
69              == CryptHashGetDigestSize(session->authHashAlg));
70
71      // add old digest
72      CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
73
74      // add commandCode
75      CryptDigestUpdateInt(&hashState, sizeof(commandCode), commandCode);
76
77      // add name if applicable
78      if(name != NULL)
79          CryptDigestUpdate2B(&hashState, &name->b);
80
81      // Complete the digest and get the results
82      CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
83
84      // If the policy reference is not null, do a second update to the digest.
85      if(ref != NULL)
86      {
87
88          // Start second hash computation
89          CryptHashStart(&hashState, session->authHashAlg);
90
91          // add policyDigest
92          CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
93
94          // add policyRef
95          CryptDigestUpdate2B(&hashState, &ref->b);
96
97          // Complete second digest
98          CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
99      }
100     // Deal with the cpHash. If the cpHash value is present
101     // then it would have already been checked to make sure that
102     // it is compatible with the current value so all we need
103     // to do here is copy it and set the isCpHashDefined attribute
104     if(cpHash != NULL && cpHash->t.size != 0)
105     {
106         session->u1.cpHash = *cpHash;
107         session->attributes.isCpHashDefined = SET;
```

 Copyright © TCG 2006-2020

```
108     }
109
110     // update the timeout if it is specified
111     if(policyTimeout != 0)
112     {
113         // If the timeout has not been set, then set it to the new value
114         // than the current timeout then set it to the new value
115         if(session->timeout == 0 || session->timeout > policyTimeout)
116             session->timeout = policyTimeout;
117     }
118     return;
119 }
```

7.4.2.3 ComputeAuthTimeout()

This function is used to determine what the authorization timeout value for the session should be.

```
120 UINT64
121 ComputeAuthTimeout(
122     SESSION         *session,               // IN: the session containing the time
123                                             //     values
124     INT32            expiration,            // IN: either the number of seconds from
125                                             //     the start of the session or the
126                                             //     time in g_timer;
127     TPM2B_NONCE     *nonce                  // IN: indicator of the time base
128     )
129 {
130     UINT64           policyTime;
131     // If no expiration, policy time is 0
132     if(expiration == 0)
133         policyTime = 0;
134     else
135     {
136         if(expiration < 0)
137             expiration = -expiration;
138         if(nonce->t.size == 0)
139             // The input time is absolute Time (not Clock), but it is expressed
140             // in seconds. To make sure that we don't time out too early, take the
141             // current value of milliseconds in g_time and add that to the input
142             // seconds value.
143             policyTime = (((UINT64)expiration) * 1000) + g_time % 1000;
144         else
145             // The policy timeout is the absolute value of the expiration in seconds
146             // added to the start time of the policy.
147             policyTime = session->startTime + (((UINT64)expiration) * 1000);
148
149     }
150     return policyTime;
151 }
```

7.4.2.4 PolicyDigestClear()

Function to reset the *policyDigest* of a session

```
152 void
153 PolicyDigestClear(
154     SESSION         *session
155     )
156 {
157     session->u2.policyDigest.t.size = CryptHashGetDigestSize(session->authHashAlg);
158     MemorySet(session->u2.policyDigest.t.buffer, 0,
159               session->u2.policyDigest.t.size);
160 }
```

```
161    BOOL
162    PolicySptCheckCondition(
163        TPM_EO           operation,
164        BYTE             *opA,
165        BYTE             *opB,
166        UINT16            size
167        )
168    {
169        // Arithmetic Comparison
170        switch(operation)
171        {
172            case TPM_EO_EQ:
173                // compare A = B
174                return (UnsignedCompareB(size, opA, size, opB) == 0);
175                break;
176            case TPM_EO_NEQ:
177                // compare A != B
178                return (UnsignedCompareB(size, opA, size, opB) != 0);
179                break;
180            case TPM_EO_SIGNED_GT:
181                // compare A > B signed
182                return (SignedCompareB(size, opA, size, opB) > 0);
183                break;
184            case TPM_EO_UNSIGNED_GT:
185                // compare A > B unsigned
186                return (UnsignedCompareB(size, opA, size, opB) > 0);
187                break;
188            case TPM_EO_SIGNED_LT:
189                // compare A < B signed
190                return (SignedCompareB(size, opA, size, opB) < 0);
191                break;
192            case TPM_EO_UNSIGNED_LT:
193                // compare A < B unsigned
194                return (UnsignedCompareB(size, opA, size, opB) < 0);
195                break;
196            case TPM_EO_SIGNED_GE:
197                // compare A >= B signed
198                return (SignedCompareB(size, opA, size, opB) >= 0);
199                break;
200            case TPM_EO_UNSIGNED_GE:
201                // compare A >= B unsigned
202                return (UnsignedCompareB(size, opA, size, opB) >= 0);
203                break;
204            case TPM_EO_SIGNED_LE:
205                // compare A <= B signed
206                return (SignedCompareB(size, opA, size, opB) <= 0);
207                break;
208            case TPM_EO_UNSIGNED_LE:
209                // compare A <= B unsigned
210                return (UnsignedCompareB(size, opA, size, opB) <= 0);
211                break;
212            case TPM_EO_BITSET:
213                // All bits SET in B are SET in A. ((A&B)=B)
214            {
215                UINT32 i;
216                for(i = 0; i < size; i++)
217                    if((opA[i] & opB[i]) != opB[i])
218                        return FALSE;
219            }
220            break;
221            case TPM_EO_BITCLEAR:
222                // All bits SET in B are CLEAR in A. ((A&B)=0)
223            {
224                UINT32 i;
225                for(i = 0; i < size; i++)
226                    if((opA[i] & opB[i]) != 0)
```

```
227                         return FALSE;
228             }
229             break;
230             default:
231                 FAIL(FATAL_ERROR_INTERNAL);
232                 break;
233         }
234         return TRUE;
235 }
```

7.5 NV Command Support (NV_spt.c)

7.5.1 Includes

```
1   #include "Tpm.h"
2   #include "NV_spt_fp.h"
```

7.5.2 Functions

7.5.2.1 NvReadAccessChecks()

Common routine for validating a read Used by TPM2_NV_Read(), TPM2_NV_ReadLock() and TPM2_PolicyNV()

Error Returns	Meaning
TPM_RC_NV_AUTHORIZATION	*autHandle* is not allowed to authorize read of the index
TPM_RC_NV_LOCKED	Read locked
TPM_RC_NV_UNINITIALIZED	Try to read an uninitialized index

```
3   TPM_RC
4   NvReadAccessChecks(
5       TPM_HANDLE       authHandle,    // IN: the handle that provided the
6                                       //     authorization
7       TPM_HANDLE       nvHandle,      // IN: the handle of the NV index to be read
8       TPMA_NV          attributes     // IN: the attributes of 'nvHandle'
9       )
10  {
11      // If data is read locked, returns an error
12     if(IS_ATTRIBUTE(attributes, TPMA_NV, READLOCKED))
13         return TPM_RC_NV_LOCKED;
14      // If the authorization was provided by the owner or platform, then check
15      // that the attributes allow the read.  If the authorization handle
16      // is the same as the index, then the checks were made when the authorization
17      // was checked..
18      if(authHandle == TPM_RH_OWNER)
19      {
20          // If Owner provided authorization then ONWERWRITE must be SET
21          if(!IS_ATTRIBUTE(attributes, TPMA_NV, OWNERREAD))
22              return TPM_RC_NV_AUTHORIZATION;
23      }
24      else if(authHandle == TPM_RH_PLATFORM)
25      {
26          // If Platform provided authorization then PPWRITE must be SET
27          if(!IS_ATTRIBUTE(attributes, TPMA_NV, PPREAD))
28              return TPM_RC_NV_AUTHORIZATION;
29      }
30      // If neither Owner nor Platform provided authorization, make sure that it was
31      // provided by this index.
32      else if(authHandle != nvHandle)
33          return TPM_RC_NV_AUTHORIZATION;
34
35  // If the index has not been written, then the value cannot be read
36  // NOTE: This has to come after other access checks to make sure that
37  // the proper authorization is given to TPM2_NV_ReadLock()
38      if(!IS_ATTRIBUTE(attributes, TPMA_NV, WRITTEN))
39          return TPM_RC_NV_UNINITIALIZED;
40
41      return TPM_RC_SUCCESS;
42  }
```

7.5.2.2 NvWriteAccessChecks()

Common routine for validating a write Used by TPM2_NV_Write(), TPM2_NV_Increment(), TPM2_SetBits(), and TPM2_NV_WriteLock()

Error Returns	Meaning
TPM_RC_NV_AUTHORIZATION	Authorization fails
TPM_RC_NV_LOCKED	Write locked

```
43  TPM_RC
44  NvWriteAccessChecks(
45      TPM_HANDLE       authHandle,    // IN: the handle that provided the
46                                      //     authorization
47      TPM_HANDLE       nvHandle,      // IN: the handle of the NV index to be written
48      TPMA_NV          attributes     // IN: the attributes of 'nvHandle'
49      )
50  {
51      // If data is write locked, returns an error
52      if(IS_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED))
53          return TPM_RC_NV_LOCKED;
54          // If the authorization was provided by the owner or platform, then check
55      // that the attributes allow the write.  If the authorization handle
56      // is the same as the index, then the checks were made when the authorization
57      // was checked..
58      if(authHandle == TPM_RH_OWNER)
59      {
60          // If Owner provided authorization then ONWERWRITE must be SET
61          if(!IS_ATTRIBUTE(attributes, TPMA_NV, OWNERWRITE))
62              return TPM_RC_NV_AUTHORIZATION;
63      }
64      else if(authHandle == TPM_RH_PLATFORM)
65      {
66          // If Platform provided authorization then PPWRITE must be SET
67          if(!IS_ATTRIBUTE(attributes, TPMA_NV, PPWRITE))
68              return TPM_RC_NV_AUTHORIZATION;
69      }
70      // If neither Owner nor Platform provided authorization, make sure that it was
71      // provided by this index.
72      else if(authHandle != nvHandle)
73          return TPM_RC_NV_AUTHORIZATION;
74      return TPM_RC_SUCCESS;
75  }
```

7.5.2.3 NvClearOrderly()

This function is used to cause *gp.orderlyState* to be cleared to the non-orderly state.

```
76  TPM_RC
77  NvClearOrderly(
78      void
79      )
80  {
81      if(gp.orderlyState < SU_DA_USED_VALUE)
82          RETURN_IF_NV_IS_NOT_AVAILABLE;
83      g_clearOrderly = TRUE;
84      return TPM_RC_SUCCESS;
85  }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

7.5.2.4 NvIsPinPassIndex()

Function to check to see if an NV index is a PIN Pass Index

Return Value	Meaning
TRUE(1)	is pin pass
FALSE(0)	is not pin pass

```
86  BOOL
87  NvIsPinPassIndex(
88      TPM_HANDLE          index       // IN: Handle to check
89      )
90  {
91      if(HandleGetType(index) == TPM_HT_NV_INDEX)
92      {
93          NV_INDEX                *nvIndex = NvGetIndexInfo(index, NULL);
94
95          return IsNvPinPassIndex(nvIndex->publicArea.attributes);
96      }
97      return FALSE;
98  }
```

 Copyright © TCG 2006-2020

7.6 Object Command Support (Object_spt.c)

7.6.1 Includes

```
1   #include "Tpm.h"
2   #include "Object_spt_fp.h"
```

7.6.2 Local Functions

7.6.2.1 GetIV2BSize()

Get the size of TPM2B_IV in canonical form that will be append to the start of the sensitive data. It includes both size of size field and size of iv data

```
3   static UINT16
4   GetIV2BSize(
5       OBJECT              *protector          // IN: the protector handle
6       )
7   {
8       TPM_ALG_ID          symAlg;
9       UINT16              keyBits;
10
11      // Determine the symmetric algorithm and size of key
12      if(protector == NULL)
13      {
14          // Use the context encryption algorithm and key size
15          symAlg = CONTEXT_ENCRYPT_ALG;
16          keyBits = CONTEXT_ENCRYPT_KEY_BITS;
17      }
18      else
19      {
20          symAlg = protector->publicArea.parameters.asymDetail.symmetric.algorithm;
21          keyBits = protector->publicArea.parameters.asymDetail.symmetric.keyBits.sym;
22      }
23
24      // The IV size is a UINT16 size field plus the block size of the symmetric
25      // algorithm
26      return sizeof(UINT16) + CryptGetSymmetricBlockSize(symAlg, keyBits);
27  }
```

7.6.2.2 ComputeProtectionKeyParms()

This function retrieves the symmetric protection key parameters for the sensitive data The parameters retrieved from this function include encryption algorithm, key size in bit, and a TPM2B_SYM_KEY containing the key material as well as the key size in bytes This function is used for any action that requires encrypting or decrypting of the sensitive area of an object or a credential blob

```
28  static void
29  ComputeProtectionKeyParms(
30      OBJECT          *protector,         // IN: the protector object
31      TPM_ALG_ID       hashAlg,           // IN: hash algorithm for KDFa
32      TPM2B           *name,              // IN: name of the object
33      TPM2B           *seedIn,            // IN: optional seed for duplication blob.
34                                          //     For non duplication blob, this
35                                          //     parameter should be NULL
36      TPM_ALG_ID      *symAlg,            // OUT: the symmetric algorithm
37      UINT16          *keyBits,           // OUT: the symmetric key size in bits
38      TPM2B_SYM_KEY   *symKey             // OUT: the symmetric key
39      )
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
40  {
41      const TPM2B          *seed = seedIn;
42
43      // Determine the algorithms for the KDF and the encryption/decryption
44      // For TPM_RH_NULL, using context settings
45      if(protector == NULL)
46      {
47          // Use the context encryption algorithm and key size
48          *symAlg = CONTEXT_ENCRYPT_ALG;
49          symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
50          *keyBits = CONTEXT_ENCRYPT_KEY_BITS;
51      }
52      else
53      {
54          TPMT_SYM_DEF_OBJECT *symDef;
55          symDef = &protector->publicArea.parameters.asymDetail.symmetric;
56          *symAlg = symDef->algorithm;
57          *keyBits = symDef->keyBits.sym;
58          symKey->t.size = (*keyBits + 7) / 8;
59      }
60      // Get seed for KDF
61      if(seed == NULL)
62          seed = GetSeedForKDF(protector);
63      // KDFa to generate symmetric key and IV value
64      CryptKDFa(hashAlg, seed, STORAGE_KEY, name, NULL,
65                symKey->t.size * 8, symKey->t.buffer, NULL, FALSE);
66      return;
67  }
```

7.6.2.3 ComputeOuterIntegrity()

The sensitive area parameter is a buffer that holds a space for the integrity value and the marshaled sensitive area. The caller should skip over the area set aside for the integrity value and compute the hash of the remainder of the object. The size field of sensitive is in unmarshaled form and the sensitive area contents is an array of bytes.

```
68  static void
69  ComputeOuterIntegrity(
70      TPM2B            *name,            // IN: the name of the object
71      OBJECT           *protector,       // IN: the object that
72                                         //     provides protection. For an object,
73                                         //     it is a parent. For a credential, it
74                                         //     is the encrypt object. For
75                                         //     a Temporary Object, it is NULL
76      TPMI_ALG_HASH     hashAlg,         // IN: algorithm to use for integrity
77      TPM2B            *seedIn,          // IN: an external seed may be provided for
78                                         //     duplication blob. For non duplication
79                                         //     blob, this parameter should be NULL
80      UINT32            sensitiveSize,   // IN: size of the marshaled sensitive data
81      BYTE             *sensitiveData,   // IN: sensitive area
82      TPM2B_DIGEST     *integrity        // OUT: integrity
83      )
84  {
85      HMAC_STATE        hmacState;
86      TPM2B_DIGEST      hmacKey;
87      const TPM2B      *seed = seedIn;
88  //
89      // Get seed for KDF
90      if(seed == NULL)
91          seed = GetSeedForKDF(protector);
92      // Determine the HMAC key bits
93      hmacKey.t.size = CryptHashGetDigestSize(hashAlg);
94
95      // KDFa to generate HMAC key
```

```
 96         CryptKDFa(hashAlg, seed, INTEGRITY_KEY, NULL, NULL,
 97                   hmacKey.t.size * 8, hmacKey.t.buffer, NULL, FALSE);
 98         // Start HMAC and get the size of the digest which will become the integrity
 99         integrity->t.size = CryptHmacStart2B(&hmacState, hashAlg, &hmacKey.b);
100
101         // Adding the marshaled sensitive area to the integrity value
102         CryptDigestUpdate(&hmacState.hashState, sensitiveSize, sensitiveData);
103
104         // Adding name
105         CryptDigestUpdate2B(&hmacState.hashState, name);
106
107         // Compute HMAC
108         CryptHmacEnd2B(&hmacState, &integrity->b);
109
110         return;
111     }
```

7.6.2.4 ComputeInnerIntegrity()

This function computes the integrity of an inner wrap

```
112     static void
113     ComputeInnerIntegrity(
114         TPM_ALG_ID       hashAlg,       // IN: hash algorithm for inner wrap
115         TPM2B           *name,          // IN: the name of the object
116         UINT16           dataSize,      // IN: the size of sensitive data
117         BYTE            *sensitiveData, // IN: sensitive data
118         TPM2B_DIGEST    *integrity      // OUT: inner integrity
119         )
120     {
121         HASH_STATE      hashState;
122     //
123         // Start hash and get the size of the digest which will become the integrity
124         integrity->t.size = CryptHashStart(&hashState, hashAlg);
125
126         // Adding the marshaled sensitive area to the integrity value
127         CryptDigestUpdate(&hashState, dataSize, sensitiveData);
128
129         // Adding name
130         CryptDigestUpdate2B(&hashState, name);
131
132         // Compute hash
133         CryptHashEnd2B(&hashState, &integrity->b);
134
135         return;
136     }
```

7.6.2.5 ProduceInnerIntegrity()

This function produces an inner integrity for regular private, credential or duplication blob It requires the sensitive data being marshaled to the *innerBuffer*, with the leading bytes reserved for integrity hash. It assume the sensitive data starts at address (*innerBuffer* + integrity size). This function integrity at the beginning of the inner buffer It returns the total size of buffer with the inner wrap

```
137     static UINT16
138     ProduceInnerIntegrity(
139         TPM2B           *name,          // IN: the name of the object
140         TPM_ALG_ID       hashAlg,       // IN: hash algorithm for inner wrap
141         UINT16           dataSize,      // IN: the size of sensitive data, excluding the
142                                         //     leading integrity buffer size
143         BYTE            *innerBuffer    // IN/OUT: inner buffer with sensitive data in
144                                         //     it.  At input, the leading bytes of this
```

```
145                                     //     buffer is reserved for integrity
146         )
147     {
148         BYTE            *sensitiveData; // pointer to the sensitive data
149         TPM2B_DIGEST     integrity;
150         UINT16           integritySize;
151         BYTE            *buffer;        // Auxiliary buffer pointer
152     //
153         // sensitiveData points to the beginning of sensitive data in innerBuffer
154         integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
155         sensitiveData = innerBuffer + integritySize;
156
157         ComputeInnerIntegrity(hashAlg, name, dataSize, sensitiveData, &integrity);
158
159         // Add integrity at the beginning of inner buffer
160         buffer = innerBuffer;
161         TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
162
163         return dataSize + integritySize;
164     }
```

7.6.2.6 CheckInnerIntegrity()

This function check integrity of inner blob

Error Returns	Meaning
TPM_RC_INTEGRITY	if the outer blob integrity is bad
errors	unmarshal errors while unmarshaling integrity

```
165     static TPM_RC
166     CheckInnerIntegrity(
167         TPM2B           *name,          // IN: the name of the object
168         TPM_ALG_ID       hashAlg,       // IN: hash algorithm for inner wrap
169         UINT16           dataSize,      // IN: the size of sensitive data, including the
170                                         //     leading integrity buffer size
171         BYTE            *innerBuffer    // IN/OUT: inner buffer with sensitive data in
172                                         //     it
173         )
174     {
175         TPM_RC          result;
176         TPM2B_DIGEST    integrity;
177         TPM2B_DIGEST    integrityToCompare;
178         BYTE            *buffer;                // Auxiliary buffer pointer
179         INT32           size;
180     //
181         // Unmarshal integrity
182         buffer = innerBuffer;
183         size = (INT32)dataSize;
184         result = TPM2B_DIGEST_Unmarshal(&integrity, &buffer, &size);
185         if(result == TPM_RC_SUCCESS)
186         {
187             // Compute integrity to compare
188             ComputeInnerIntegrity(hashAlg, name, (UINT16)size, buffer,
189                                   &integrityToCompare);
190             // Compare outer blob integrity
191             if(!MemoryEqual2B(&integrity.b, &integrityToCompare.b))
192                 result = TPM_RC_INTEGRITY;
193         }
194         return result;
195     }
```

 Copyright © TCG 2006-2020

7.6.3 Public Functions

7.6.3.1 AdjustAuthSize()

This function will validate that the input *authValue* is no larger than the *digestSize* for the *nameAlg*. It will then pad with zeros to the size of the digest.

```
196  BOOL
197  AdjustAuthSize(
198      TPM2B_AUTH          *auth,          // IN/OUT: value to adjust
199      TPMI_ALG_HASH        nameAlg        // IN:
200      )
201  {
202      UINT16               digestSize;
203  //
204      // If there is no nameAlg, then this is a LoadExternal and the authVale can
205      // be any size up to the maximum allowed by the
206      digestSize = (nameAlg == TPM_ALG_NULL) ? sizeof(TPMU_HA)
207          : CryptHashGetDigestSize(nameAlg);
208      if(digestSize < MemoryRemoveTrailingZeros(auth))
209          return FALSE;
210      else if(digestSize > auth->t.size)
211          MemoryPad2B(&auth->b, digestSize);
212      auth->t.size = digestSize;
213
214      return TRUE;
215  }
```

7.6.3.2 AreAttributesForParent()

This function is called by create, load, and import functions.

NOTE: The *isParent* attribute is SET when an object is loaded and it has attributes that are suitable for a parent object.

Return Value	Meaning
TRUE(1)	properties are those of a parent
FALSE(0)	properties are not those of a parent

```
216  BOOL
217  ObjectIsParent(
218      OBJECT          *parentObject    // IN: parent handle
219      )
220  {
221      return parentObject->attributes.isParent;
222  }
```

7.6.3.3 CreateChecks()

Attribute checks that are unique to creation.

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*sensitiveDataOrigin* is not consistent with the object type
other	returns from PublicAttributesValidation()

```
223  TPM_RC
224  CreateChecks(
```

```
225         OBJECT                  *parentObject,
226         TPMT_PUBLIC             *publicArea,
227         UINT16                   sensitiveDataSize
228         )
229     {
230         TPMA_OBJECT              attributes = publicArea->objectAttributes;
231         TPM_RC                   result = TPM_RC_SUCCESS;
232     //
233         // If the caller indicates that they have provided the data, then make sure that
234         // they have provided some data.
235         if((!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
236            && (sensitiveDataSize == 0))
237             return TPM_RCS_ATTRIBUTES;
238         // For an ordinary object, data can only be provided when sensitiveDataOrigin
239         // is CLEAR
240         if((parentObject != NULL)
241            && (IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
242            && (sensitiveDataSize != 0))
243             return TPM_RCS_ATTRIBUTES;
244         switch(publicArea->type)
245         {
246             case ALG_KEYEDHASH_VALUE:
247                 // if this is a data object (sign == decrypt == CLEAR) then the
248                 // TPM cannot be the data source.
249                 if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
250                    && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt)
251                    && IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
252                     result = TPM_RC_ATTRIBUTES;
253                 // comment out the next line in order to prevent a fixedTPM derivation
254                 // parent
255     //            break;
256             case ALG_SYMCIPHER_VALUE:
257                 // A restricted key symmetric key (SYMCIPHER and KEYEDHASH)
258                 // must have sensitiveDataOrigin SET unless it has fixedParent and
259                 // fixedTPM CLEAR.
260                 if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
261                     if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
262                         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent)
263                            || IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
264                             result = TPM_RCS_ATTRIBUTES;
265                 break;
266             default: // Asymmetric keys cannot have the sensitive portion provided
267                 if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
268                     result = TPM_RCS_ATTRIBUTES;
269                 break;
270         }
271         if(TPM_RC_SUCCESS == result)
272         {
273             result = PublicAttributesValidation(parentObject, publicArea);
274         }
275         return result;
276     }
```

7.6.3.4 SchemeChecks

This function is called by TPM2_LoadExternal() and PublicAttributesValidation(). This function validates the schemes in the public area of an object.

 Copyright © TCG 2006-2020

Error Returns	Meaning
TPM_RC_HASH	non-duplicable storage key and its parent have different name algorithm
TPM_RC_KDF	incorrect KDF specified for decrypting keyed hash object
TPM_RC_KEY	invalid key size values in an asymmetric key public area
TPM_RCS_SCHEME	inconsistent attributes *decrypt*, *sign*, *restricted* and key's scheme ID; or hash algorithm is inconsistent with the scheme ID for keyed hash object
TPM_RC_SYMMETRIC	a storage key with no symmetric algorithm specified; or non-storage key with symmetric algorithm different from TPM_ALG_NULL

```
277  TPM_RC
278  SchemeChecks(
279      OBJECT            *parentObject,   // IN: parent (null if primary seed)
280      TPMT_PUBLIC       *publicArea      // IN: public area of the object
281      )
282  {
283      TPMT_SYM_DEF_OBJECT      *symAlgs = NULL;
284      TPM_ALG_ID                scheme = TPM_ALG_NULL;
285      TPMA_OBJECT               attributes = publicArea->objectAttributes;
286      TPMU_PUBLIC_PARMS         *parms = &publicArea->parameters;
287  //
288      switch(publicArea->type)
289      {
290          case ALG_SYMCIPHER_VALUE:
291              symAlgs = &parms->symDetail.sym;
292              // If this is a decrypt key, then only the block cipher modes (not
293              // SMAC) are valid. TPM_ALG_NULL is OK too. If this is a 'sign' key,
294              // then any mode that got through the unmarshaling is OK.
295              if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt)
296                 && !CryptSymModeIsValid(symAlgs->mode.sym, TRUE))
297                  return TPM_RCS_SCHEME;
298              break;
299          case ALG_KEYEDHASH_VALUE:
300              scheme = parms->keyedHashDetail.scheme.scheme;
301              // if both sign and decrypt
302              if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
303                 == IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
304              {
305                  // if both sign and decrypt are set or clear, then need
306                  // TPM_ALG_NULL as scheme
307                  if(scheme != TPM_ALG_NULL)
308                      return TPM_RCS_SCHEME;
309              }
310              else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
311                      && scheme != TPM_ALG_HMAC)
312                  return TPM_RCS_SCHEME;
313              else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
314              {
315                  if(scheme != TPM_ALG_XOR)
316                      return TPM_RCS_SCHEME;
317                  // If this is a derivation parent, then the KDF needs to be
318                  // SP800-108 for this implementation. This is the only derivation
319                  // supported by this implementation. Other implementations could
320                  // support additional schemes. There is no default.
321                  if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
322                  {
323                      if(parms->keyedHashDetail.scheme.details.xor.kdf
324                         != TPM_ALG_KDF1_SP800_108)
325                          return TPM_RCS_SCHEME;
326                      // Must select a digest.
```

 Copyright © TCG 2006-2020

```
327                 if(CryptHashGetDigestSize(
328                     parms->keyedHashDetail.scheme.details.xor.hashAlg) == 0)
329                     return TPM_RCS_HASH;
330             }
331         }
332         break;
333     default: // handling for asymmetric
334         scheme = parms->asymDetail.scheme.scheme;
335         symAlgs = &parms->asymDetail.symmetric;
336         // if the key is both sign and decrypt, then the scheme must be
337         // TPM_ALG_NULL because there is no way to specify both a sign and a
338         // decrypt scheme in the key.
339         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
340            == IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
341         {
342             // scheme must be TPM_ALG_NULL
343             if(scheme != TPM_ALG_NULL)
344                 return TPM_RCS_SCHEME;
345         }
346         else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign))
347         {
348             // If this is a signing key, see if it has a signing scheme
349             if(CryptIsAsymSignScheme(publicArea->type, scheme))
350             {
351                 // if proper signing scheme then it needs a proper hash
352                 if(parms->asymDetail.scheme.details.anySig.hashAlg
353                    == TPM_ALG_NULL)
354                     return TPM_RCS_SCHEME;
355             }
356             else
357             {
358                 // signing key that does not have a proper signing scheme.
359                 // This is OK if the key is not restricted and its scheme
360                 // is TPM_ALG_NULL
361                 if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
362                    || scheme != TPM_ALG_NULL)
363                     return TPM_RCS_SCHEME;
364             }
365         }
366         else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
367         {
368             if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
369             {
370                 // for a restricted decryption key (a parent), scheme
371                 // is required to be TPM_ALG_NULL
372                 if(scheme != TPM_ALG_NULL)
373                     return TPM_RCS_SCHEME;
374             }
375             else
376             {
377                 // For an unrestricted decryption key, the scheme has to
378                 // be a valid scheme or TPM_ALG_NULL
379                 if(scheme != TPM_ALG_NULL &&
380                    !CryptIsAsymDecryptScheme(publicArea->type, scheme))
381                     return TPM_RCS_SCHEME;
382             }
383         }
384         if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
385            || !IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
386         {
387             // For an asymmetric key that is not a parent, the symmetric
388             // algorithms must be TPM_ALG_NULL
389             if(symAlgs->algorithm != TPM_ALG_NULL)
390                 return TPM_RCS_SYMMETRIC;
391         }
392         // Special checks for an ECC key
```

```
393 #if ALG_ECC
394             if(publicArea->type == TPM_ALG_ECC)
395             {
396                 TPM_ECC_CURVE            curveID;
397                 const TPMT_ECC_SCHEME   *curveScheme;
398
399                 curveID = publicArea->parameters.eccDetail.curveID;
400                 curveScheme = CryptGetCurveSignScheme(curveID);
401                 // The curveId must be valid or the unmarshaling is busted.
402                 pAssert(curveScheme != NULL);
403
404                 // If the curveID requires a specific scheme, then the key must
405                 // select the same scheme
406                 if(curveScheme->scheme != TPM_ALG_NULL)
407                 {
408                     TPMS_ECC_PARMS      *ecc = &publicArea->parameters.eccDetail;
409                     if(scheme != curveScheme->scheme)
410                         return TPM_RCS_SCHEME;
411                     // The scheme can allow any hash, or not...
412                     if(curveScheme->details.anySig.hashAlg != TPM_ALG_NULL
413                        && (ecc->scheme.details.anySig.hashAlg
414                            != curveScheme->details.anySig.hashAlg))
415                         return TPM_RCS_SCHEME;
416                 }
417                 // For now, the KDF must be TPM_ALG_NULL
418                 if(publicArea->parameters.eccDetail.kdf.scheme != TPM_ALG_NULL)
419                     return TPM_RCS_KDF;
420             }
421 #endif
422             break;
423     }
424     // If this is a restricted decryption key with symmetric algorithms, then it
425     // is an ordinary parent (not a derivation parent). It needs to specific
426     // symmetric algorithms other than TPM_ALG_NULL
427     if(symAlgs != NULL
428        && IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
429        && IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
430     {
431         if(symAlgs->algorithm == TPM_ALG_NULL)
432             return TPM_RCS_SYMMETRIC;
433 #if 0       //??
434 // This next check is under investigation. Need to see if it will break Windows
435 // before it is enabled. If it does not, then it should be default because a
436 // the mode used with a parent is always CFB and Part 2 indicates as much.
437         if(symAlgs->mode.sym != TPM_ALG_CFB)
438             return TPM_RCS_MODE;
439 #endif
440         // If this parent is not duplicable, then the symmetric algorithms
441         // (encryption and hash) must match those of its parent
442         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent)
443            && (parentObject != NULL))
444         {
445             if(publicArea->nameAlg != parentObject->publicArea.nameAlg)
446                 return TPM_RCS_HASH;
447             if(!MemoryEqual(symAlgs, &parentObject->publicArea.parameters,
448                             sizeof(TPMT_SYM_DEF_OBJECT)))
449                 return TPM_RCS_SYMMETRIC;
450         }
451     }
452     return TPM_RC_SUCCESS;
453 }
```

7.6.3.5 PublicAttributesValidation()

This function validates the values in the public area of an object. This function is used in the processing of TPM2_Create(), TPM2_CreatePrimary(), TPM2_CreateLoaded(), TPM2_Load(), TPM2_Import(), and TPM2_LoadExternal(). For TPM2_Import() this is only used if the new parent has *fixedTPM* SET. For TPM2_LoadExternal(), this is not used for a public-only key

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*fixedTPM*, *fixedParent*, or *encryptedDuplication* attributes are inconsistent between themselves or with those of the parent object; inconsistent *restricted*, *decrypt* and *sign* attributes; attempt to inject sensitive data for an asymmetric key; attempt to create a symmetric cipher key that is not a decryption key
TPM_RC_HASH	*nameAlg* is TPM_ALG_NULL
TPM_RC_SIZE	*authPolicy* size does not match digest size of the name algorithm in *publicArea*
other	returns from SchemeChecks()

```
454  TPM_RC
455  PublicAttributesValidation(
456      OBJECT           *parentObject,   // IN: input parent object
457      TPMT_PUBLIC      *publicArea      // IN: public area of the object
458      )
459  {
460      TPMA_OBJECT       attributes = publicArea->objectAttributes;
461      TPMA_OBJECT       parentAttributes = TPMA_ZERO_INITIALIZER();
462  //
463      if(parentObject != NULL)
464          parentAttributes = parentObject->publicArea.objectAttributes;
465      if(publicArea->nameAlg == TPM_ALG_NULL)
466          return TPM_RCS_HASH;
467      // If there is an authPolicy, it needs to be the size of the digest produced
468      // by the nameAlg of the object
469      if((publicArea->authPolicy.t.size != 0
470          && (publicArea->authPolicy.t.size
471              != CryptHashGetDigestSize(publicArea->nameAlg))))
472          return TPM_RCS_SIZE;
473      // If the parent is fixedTPM (including a Primary Object) the object must have
474      // the same value for fixedTPM and fixedParent
475      if(parentObject == NULL
476         || IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, fixedTPM))
477      {
478          if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent)
479             != IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
480              return TPM_RCS_ATTRIBUTES;
481      }
482      else
483      {
484          // The parent is not fixedTPM so the object can't be fixedTPM
485          if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
486              return  TPM_RCS_ATTRIBUTES;
487      }
488      // See if sign and decrypt are the same
489      if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
490         == IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
491      {
492          // a restricted key cannot have both SET or both CLEAR
493          if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
494              return TPM_RC_ATTRIBUTES;
495          // only a data object may have both sign and decrypt CLEAR
496          // BTW, since we know that decrypt==sign, no need to check both
```

 Copyright © TCG 2006-2020

```
497             if(publicArea->type != TPM_ALG_KEYEDHASH
498                && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign))
499                 return TPM_RC_ATTRIBUTES;
500         }
501         // If the object can't be duplicated (directly or indirectly) then there
502         // is no justification for having encryptedDuplication SET
503         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM)
504            && IS_ATTRIBUTE(attributes, TPMA_OBJECT, encryptedDuplication))
505             return TPM_RCS_ATTRIBUTES;
506         // If a parent object has fixedTPM CLEAR, the child must have the
507         // same encryptedDuplication value as its parent.
508         // Primary objects are considered to have a fixedTPM parent (the seeds).
509         if(parentObject != NULL
510            && !IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, fixedTPM))
511         {
512             if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, encryptedDuplication)
513                != IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, encryptedDuplication))
514                 return TPM_RCS_ATTRIBUTES;
515         }
516         // Special checks for derived objects
517         if((parentObject != NULL) && (parentObject->attributes.derivation == SET))
518         {
519             // A derived object has the same settings for fixedTPM as its parent
520             if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM)
521                != IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, fixedTPM))
522                 return TPM_RCS_ATTRIBUTES;
523             // A derived object is required to be fixedParent
524             if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent))
525                 return TPM_RCS_ATTRIBUTES;
526         }
527         return SchemeChecks(parentObject, publicArea);
528     }
```

7.6.3.6 FillInCreationData()

Fill in creation data for an object.

```
529     void
530     FillInCreationData(
531         TPMI_DH_OBJECT            parentHandle,  // IN: handle of parent
532         TPMI_ALG_HASH             nameHashAlg,   // IN: name hash algorithm
533         TPML_PCR_SELECTION       *creationPCR,   // IN: PCR selection
534         TPM2B_DATA               *outsideData,   // IN: outside data
535         TPM2B_CREATION_DATA      *outCreation,   // OUT: creation data for output
536         TPM2B_DIGEST             *creationDigest // OUT: creation digest
537         )
538     {
539         BYTE                  creationBuffer[sizeof(TPMS_CREATION_DATA)];
540         BYTE                 *buffer;
541         HASH_STATE            hashState;
542     //
543         // Fill in TPMS_CREATION_DATA in outCreation
544
545         // Compute PCR digest
546         PCRComputeCurrentDigest(nameHashAlg, creationPCR,
547                                 &outCreation->creationData.pcrDigest);
548
549         // Put back PCR selection list
550         outCreation->creationData.pcrSelect = *creationPCR;
551
552         // Get locality
553         outCreation->creationData.locality
554             = LocalityGetAttributes(_plat__LocalityGet());
555         outCreation->creationData.parentNameAlg = TPM_ALG_NULL;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
556
557     // If the parent is either a primary seed or TPM_ALG_NULL, then  the Name
558     // and QN of the parent are the parent's handle.
559     if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
560     {
561         buffer = &outCreation->creationData.parentName.t.name[0];
562         outCreation->creationData.parentName.t.size =
563             TPM_HANDLE_Marshal(&parentHandle, &buffer, NULL);
564         // For a primary or temporary object, the parent name (a handle) and the
565         // parent's QN are the same
566         outCreation->creationData.parentQualifiedName
567             = outCreation->creationData.parentName;
568     }
569     else            // Regular object
570     {
571         OBJECT          *parentObject = HandleToObject(parentHandle);
572 //
573         // Set name algorithm
574         outCreation->creationData.parentNameAlg = parentObject->publicArea.nameAlg;
575
576         // Copy parent name
577         outCreation->creationData.parentName = parentObject->name;
578
579         // Copy parent qualified name
580         outCreation->creationData.parentQualifiedName = parentObject->qualifiedName;
581     }
582     // Copy outside information
583     outCreation->creationData.outsideInfo = *outsideData;
584
585     // Marshal creation data to canonical form
586     buffer = creationBuffer;
587     outCreation->size = TPMS_CREATION_DATA_Marshal(&outCreation->creationData,
588                                                    &buffer, NULL);
589     // Compute hash for creation field in public template
590     creationDigest->t.size = CryptHashStart(&hashState, nameHashAlg);
591     CryptDigestUpdate(&hashState, outCreation->size, creationBuffer);
592     CryptHashEnd2B(&hashState, &creationDigest->b);
593
594     return;
595 }
```

7.6.3.7 GetSeedForKDF()

Get a seed for KDF. The KDF for encryption and HMAC key use the same seed.

```
596 const TPM2B *
597 GetSeedForKDF(
598     OBJECT          *protector         // IN: the protector handle
599     )
600 {
601     // Get seed for encryption key.  Use input seed if provided.
602     // Otherwise, using protector object's seedValue.  TPM_RH_NULL is the only
603     // exception that we may not have a loaded object as protector.  In such a
604     // case, use nullProof as seed.
605     if(protector == NULL)
606         return &gr.nullProof.b;
607     else
608         return &protector->sensitive.seedValue.b;
609 }
```

 Copyright © TCG 2006-2020

7.6.3.8 ProduceOuterWrap()

This function produce outer wrap for a buffer containing the sensitive data. It requires the sensitive data being marshaled to the *outerBuffer*, with the leading bytes reserved for integrity hash. If iv is used, iv space should be reserved at the beginning of the buffer. It assumes the sensitive data starts at address (*outerBuffer* + integrity size [+ iv size]). This function performs:

a) Add IV before sensitive area if required

b) encrypt sensitive data, if iv is required, encrypt by iv. otherwise, encrypted by a NULL iv

c) add HMAC integrity at the beginning of the buffer It returns the total size of blob with outer wrap

```
610 UINT16
611 ProduceOuterWrap(
612     OBJECT          *protector,     // IN: The handle of the object that provides
613                                     //     protection.  For object, it is parent
614                                     //     handle. For credential, it is the handle
615                                     //     of encrypt object.
616     TPM2B           *name,          // IN: the name of the object
617     TPM_ALG_ID       hashAlg,       // IN: hash algorithm for outer wrap
618     TPM2B           *seed,          // IN: an external seed may be provided for
619                                     //     duplication blob. For non duplication
620                                     //     blob, this parameter should be NULL
621     BOOL             useIV,         // IN: indicate if an IV is used
622     UINT16           dataSize,      // IN: the size of sensitive data, excluding the
623                                     //     leading integrity buffer size or the
624                                     //     optional iv size
625     BYTE            *outerBuffer    // IN/OUT: outer buffer with sensitive data in
626                                     //     it
627     )
628 {
629     TPM_ALG_ID      symAlg;
630     UINT16          keyBits;
631     TPM2B_SYM_KEY   symKey;
632     TPM2B_IV        ivRNG;          // IV from RNG
633     TPM2B_IV        *iv = NULL;
634     UINT16          ivSize = 0;     // size of iv area, including the size field
635     BYTE            *sensitiveData; // pointer to the sensitive data
636     TPM2B_DIGEST    integrity;
637     UINT16          integritySize;
638     BYTE            *buffer;        // Auxiliary buffer pointer
639 //
640     // Compute the beginning of sensitive data.  The outer integrity should
641     // always exist if this function is called to make an outer wrap
642     integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
643     sensitiveData = outerBuffer + integritySize;
644
645     // If iv is used, adjust the pointer of sensitive data and add iv before it
646     if(useIV)
647     {
648         ivSize = GetIV2BSize(protector);
649
650         // Generate IV from RNG.  The iv data size should be the total IV area
651         // size minus the size of size field
652         ivRNG.t.size = ivSize - sizeof(UINT16);
653         CryptRandomGenerate(ivRNG.t.size, ivRNG.t.buffer);
654
655         // Marshal IV to buffer
656         buffer = sensitiveData;
657         TPM2B_IV_Marshal(&ivRNG, &buffer, NULL);
658
659         // adjust sensitive data starting after IV area
660         sensitiveData += ivSize;
661
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
662             // Use iv for encryption
663             iv = &ivRNG;
664         }
665         // Compute symmetric key parameters for outer buffer encryption
666         ComputeProtectionKeyParms(protector, hashAlg, name, seed,
667                                   &symAlg, &keyBits, &symKey);
668         // Encrypt inner buffer in place
669         CryptSymmetricEncrypt(sensitiveData, symAlg, keyBits,
670                               symKey.t.buffer, iv, TPM_ALG_CFB, dataSize,
671                               sensitiveData);
672         // Compute outer integrity. Integrity computation includes the optional IV
673         // area
674         ComputeOuterIntegrity(name, protector, hashAlg, seed, dataSize + ivSize,
675                               outerBuffer + integritySize, &integrity);
676         // Add integrity at the beginning of outer buffer
677         buffer = outerBuffer;
678         TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
679
680         // return the total size in outer wrap
681         return dataSize + integritySize + ivSize;
682     }
```

7.6.3.9 UnwrapOuter()

This function remove the outer wrap of a blob containing sensitive data This function performs:

a) check integrity of outer blob

b) decrypt outer blob

Error Returns	Meaning
TPM_RCS_INSUFFICIENT	error during sensitive data unmarshaling
TPM_RCS_INTEGRITY	sensitive data integrity is broken
TPM_RCS_SIZE	error during sensitive data unmarshaling
TPM_RCS_VALUE	IV size for CFB does not match the encryption algorithm block size

```
683     TPM_RC
684     UnwrapOuter(
685         OBJECT          *protector,     // IN: The object that provides
686                                         //     protection. For object, it is parent
687                                         //     handle. For credential, it is the
688                                         //     encrypt object.
689         TPM2B           *name,          // IN: the name of the object
690         TPM_ALG_ID       hashAlg,       // IN: hash algorithm for outer wrap
691         TPM2B           *seed,          // IN: an external seed may be provided for
692                                         //     duplication blob. For non duplication
693                                         //     blob, this parameter should be NULL.
694         BOOL             useIV,         // IN: indicates if an IV is used
695         UINT16           dataSize,      // IN: size of sensitive data in outerBuffer,
696                                         //     including the leading integrity buffer
697                                         //     size, and an optional iv area
698         BYTE            *outerBuffer    // IN/OUT: sensitive data
699         )
700     {
701         TPM_RC          result;
702         TPM_ALG_ID      symAlg = TPM_ALG_NULL;
703         TPM2B_SYM_KEY   symKey;
704         UINT16          keyBits = 0;
705         TPM2B_IV        ivIn;               // input IV retrieved from input buffer
706         TPM2B_IV        *iv = NULL;
707         BYTE            *sensitiveData;     // pointer to the sensitive data
```

 Copyright © TCG 2006-2020

```
708         TPM2B_DIGEST     integrityToCompare;
709         TPM2B_DIGEST     integrity;
710         INT32            size;
711     //
712         // Unmarshal integrity
713         sensitiveData = outerBuffer;
714         size = (INT32)dataSize;
715         result = TPM2B_DIGEST_Unmarshal(&integrity, &sensitiveData, &size);
716         if(result == TPM_RC_SUCCESS)
717         {
718             // Compute integrity to compare
719             ComputeOuterIntegrity(name, protector, hashAlg, seed,
720                                   (UINT16)size, sensitiveData,
721                                   &integrityToCompare);
722             // Compare outer blob integrity
723             if(!MemoryEqual2B(&integrity.b, &integrityToCompare.b))
724                 return TPM_RCS_INTEGRITY;
725             // Get the symmetric algorithm parameters used for encryption
726             ComputeProtectionKeyParms(protector, hashAlg, name, seed,
727                                       &symAlg, &keyBits, &symKey);
728             // Retrieve IV if it is used
729             if(useIV)
730             {
731                 result = TPM2B_IV_Unmarshal(&ivIn, &sensitiveData, &size);
732                 if(result == TPM_RC_SUCCESS)
733                 {
734                     // The input iv size for CFB must match the encryption algorithm
735                     // block size
736                     if(ivIn.t.size != CryptGetSymmetricBlockSize(symAlg, keyBits))
737                         result = TPM_RC_VALUE;
738                     else
739                         iv = &ivIn;
740                 }
741             }
742         }
743         // If no errors, decrypt private in place. Since this function uses CFB,
744         // CryptSymmetricDecrypt() will not return any errors. It may fail but it will
745         // not return an error.
746         if(result == TPM_RC_SUCCESS)
747             CryptSymmetricDecrypt(sensitiveData, symAlg, keyBits,
748                                   symKey.t.buffer, iv, TPM_ALG_CFB,
749                                   (UINT16)size, sensitiveData);
750         return result;
751     }
```

7.6.3.10 MarshalSensitive()

This function is used to marshal a sensitive area. Among other things, it adjusts the size of the *authValue* to be no smaller than the digest of *nameAlg* Returns the size of the marshaled area.

```
752     static UINT16
753     MarshalSensitive(
754         OBJECT              *parent,            // IN: the object parent (optional)
755         BYTE                *buffer,            // OUT: receiving buffer
756         TPMT_SENSITIVE      *sensitive,         // IN: the sensitive area to marshal
757         TPMI_ALG_HASH        nameAlg            // IN:
758         )
759     {
760         BYTE                *sizeField = buffer;    // saved so that size can be
761                                                     // marshaled after it is known
762         UINT16               retVal;
763     //
764         // Pad the authValue if needed
765         MemoryPad2B(&sensitive->authValue.b, CryptHashGetDigestSize(nameAlg));
```

 Copyright © TCG 2006-2020

```
766       buffer += 2;
767
768       // Marshal the structure
769   #if ALG_RSA
770       // If the sensitive size is the special case for a prime in the type
771       if((sensitive->sensitive.rsa.t.size & RSA_prime_flag) > 0)
772       {
773           UINT16              sizeSave = sensitive->sensitive.rsa.t.size;
774       //
775           // Turn off the flag that indicates that the sensitive->sensitive contains
776           // the CRT form of the exponent.
777           sensitive->sensitive.rsa.t.size &= ~(RSA_prime_flag);
778           // If the parent isn't fixedTPM, then truncate the sensitive data to be
779           // the size of the prime. Otherwise, leave it at the current size which
780           // is the full CRT size.
781           if(parent == NULL
782              || !IS_ATTRIBUTE(parent->publicArea.objectAttributes,
783                                TPMA_OBJECT, fixedTPM))
784               sensitive->sensitive.rsa.t.size /= 5;
785           retVal = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
786           // Restore the flag and the size.
787           sensitive->sensitive.rsa.t.size = sizeSave;
788       }
789       else
790   #endif
791       retVal = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
792
793       // Marshal the size
794       retVal = (UINT16)(retVal + UINT16_Marshal(&retVal, &sizeField, NULL));
795
796       return retVal;
797   }
```

7.6.3.11 SensitiveToPrivate()

This function prepare the private blob for off the chip storage The operations in this function:

a) marshal TPM2B_SENSITIVE structure into the buffer of TPM2B_PRIVATE

b) apply encryption to the sensitive area.

c) apply outer integrity computation.

```
798   void
799   SensitiveToPrivate(
800       TPMT_SENSITIVE   *sensitive,      // IN: sensitive structure
801       TPM2B_NAME       *name,           // IN: the name of the object
802       OBJECT           *parent,         // IN: The parent object
803       TPM_ALG_ID        nameAlg,        // IN: hash algorithm in public area.  This
804                                         //     parameter is used when parentHandle is
805                                         //     NULL, in which case the object is
806                                         //     temporary.
807       TPM2B_PRIVATE    *outPrivate      // OUT: output private structure
808       )
809   {
810       BYTE                 *sensitiveData;     // pointer to the sensitive data
811       UINT16               dataSize;           // data blob size
812       TPMI_ALG_HASH        hashAlg;            // hash algorithm for integrity
813       UINT16               integritySize;
814       UINT16               ivSize;
815   //
816       pAssert(name != NULL && name->t.size != 0);
817
818       // Find the hash algorithm for integrity computation
819       if(parent == NULL)
```

```
820         {
821             // For Temporary Object, using self name algorithm
822             hashAlg = nameAlg;
823         }
824         else
825         {
826             // Otherwise, using parent's name algorithm
827             hashAlg = parent->publicArea.nameAlg;
828         }
829         // Starting of sensitive data without wrappers
830         sensitiveData = outPrivate->t.buffer;
831
832         // Compute the integrity size
833         integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
834
835         // Reserve space for integrity
836         sensitiveData += integritySize;
837
838         // Get iv size
839         ivSize = GetIV2BSize(parent);
840
841         // Reserve space for iv
842         sensitiveData += ivSize;
843
844         // Marshal the sensitive area including authValue size adjustments.
845         dataSize = MarshalSensitive(parent, sensitiveData, sensitive, nameAlg);
846
847         //Produce outer wrap, including encryption and HMAC
848         outPrivate->t.size = ProduceOuterWrap(parent, &name->b, hashAlg, NULL,
849                                               TRUE, dataSize, outPrivate->t.buffer);
850         return;
851     }
```

7.6.3.12 PrivateToSensitive()

Unwrap a input private area. Check the integrity, decrypt and retrieve data to a sensitive structure. The operations in this function:

a) check the integrity HMAC of the input private area

b) decrypt the private buffer

c) unmarshal TPMT_SENSITIVE structure into the buffer of TPMT_SENSITIVE

Error Returns	Meaning
TPM_RCS_INTEGRITY	if the private area integrity is bad
TPM_RC_SENSITIVE	unmarshal errors while unmarshaling TPMS_ENCRYPT from input private
TPM_RCS_SIZE	error during sensitive data unmarshaling
TPM_RCS_VALUE	outer wrapper does not have an *iV* of the correct size

```
852     TPM_RC
853     PrivateToSensitive(
854         TPM2B           *inPrivate,       // IN: input private structure
855         TPM2B           *name,            // IN: the name of the object
856         OBJECT          *parent,          // IN: parent object
857         TPM_ALG_ID       nameAlg,         // IN: hash algorithm in public area.  It is
858                                           //     passed separately because we only pass
859                                           //     name, rather than the whole public area
860                                           //     of the object.  This parameter is used in
861                                           //     the following two cases: 1. primary
862                                           //     objects. 2. duplication blob with inner
```

```
863                                          //      wrap.  In other cases, this parameter
864                                          //      will be ignored
865     TPMT_SENSITIVE  *sensitive           // OUT: sensitive structure
866     )
867 {
868     TPM_RC          result;
869     BYTE            *buffer;
870     INT32           size;
871     BYTE            *sensitiveData; // pointer to the sensitive data
872     UINT16          dataSize;
873     UINT16          dataSizeInput;
874     TPMI_ALG_HASH   hashAlg;         // hash algorithm for integrity
875     UINT16          integritySize;
876     UINT16          ivSize;
877 //
878     // Make sure that name is provided
879     pAssert(name != NULL && name->size != 0);
880
881     // Find the hash algorithm for integrity computation
882     // For Temporary Object (parent == NULL) use self name algorithm;
883     // Otherwise, using parent's name algorithm
884     hashAlg = (parent == NULL) ? nameAlg : parent->publicArea.nameAlg;
885
886     // unwrap outer
887     result = UnwrapOuter(parent, name, hashAlg, NULL, TRUE,
888                          inPrivate->size, inPrivate->buffer);
889     if(result != TPM_RC_SUCCESS)
890         return result;
891     // Compute the inner integrity size.
892     integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
893
894     // Get iv size
895     ivSize = GetIV2BSize(parent);
896
897     // The starting of sensitive data and data size without outer wrapper
898     sensitiveData = inPrivate->buffer + integritySize + ivSize;
899     dataSize = inPrivate->size - integritySize - ivSize;
900
901     // Unmarshal input data size
902     buffer = sensitiveData;
903     size = (INT32)dataSize;
904     result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
905     if(result == TPM_RC_SUCCESS)
906     {
907         if((dataSizeInput + sizeof(UINT16)) != dataSize)
908             result = TPM_RC_SENSITIVE;
909         else
910         {
911             // Unmarshal sensitive buffer to sensitive structure
912             result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
913             if(result != TPM_RC_SUCCESS || size != 0)
914             {
915                 result = TPM_RC_SENSITIVE;
916             }
917         }
918     }
919     return result;
920 }
```

7.6.3.13 SensitiveToDuplicate()

This function prepare the duplication blob from the sensitive area. The operations in this function:

 Copyright © TCG 2006-2020

a) marshal TPMT_SENSITIVE structure into the buffer of TPM2B_PRIVATE

b) apply inner wrap to the sensitive area if required

c) apply outer wrap if required

```
921  void
922  SensitiveToDuplicate(
923      TPMT_SENSITIVE      *sensitive,      // IN: sensitive structure
924      TPM2B               *name,           // IN: the name of the object
925      OBJECT              *parent,         // IN: The new parent object
926      TPM_ALG_ID           nameAlg,        // IN: hash algorithm in public area. It
927                                           //     is passed separately because we
928                                           //     only pass name, rather than the
929                                           //     whole public area of the object.
930      TPM2B               *seed,           // IN: the external seed. If external
931                                           //     seed is provided with size of 0,
932                                           //     no outer wrap should be applied
933                                           //     to duplication blob.
934      TPMT_SYM_DEF_OBJECT *symDef,         // IN: Symmetric key definition. If the
935                                           //     symmetric key algorithm is NULL,
936                                           //     no inner wrap should be applied.
937      TPM2B_DATA          *innerSymKey,    // IN/OUT: a symmetric key may be
938                                           //     provided to encrypt the inner
939                                           //     wrap of a duplication blob. May
940                                           //     be generated here if needed.
941      TPM2B_PRIVATE       *outPrivate      // OUT: output private structure
942      )
943  {
944      BYTE            *sensitiveData; // pointer to the sensitive data
945      TPMI_ALG_HASH   outerHash = TPM_ALG_NULL;// The hash algorithm for outer wrap
946      TPMI_ALG_HASH   innerHash = TPM_ALG_NULL;// The hash algorithm for inner wrap
947      UINT16          dataSize;       // data blob size
948      BOOL            doInnerWrap = FALSE;
949      BOOL            doOuterWrap = FALSE;
950  //
951      // Make sure that name is provided
952      pAssert(name != NULL && name->size != 0);
953
954      // Make sure symDef and innerSymKey are not NULL
955      pAssert(symDef != NULL && innerSymKey != NULL);
956
957      // Starting of sensitive data without wrappers
958      sensitiveData = outPrivate->t.buffer;
959
960      // Find out if inner wrap is required
961      if(symDef->algorithm != TPM_ALG_NULL)
962      {
963          doInnerWrap = TRUE;
964
965          // Use self nameAlg as inner hash algorithm
966          innerHash = nameAlg;
967
968          // Adjust sensitive data pointer
969          sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(innerHash);
970      }
971      // Find out if outer wrap is required
972      if(seed->size != 0)
973      {
974          doOuterWrap = TRUE;
975
976          // Use parent nameAlg as outer hash algorithm
977          outerHash = parent->publicArea.nameAlg;
978
979          // Adjust sensitive data pointer
980          sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
981         }
982         // Marshal sensitive area
983         dataSize = MarshalSensitive(NULL, sensitiveData, sensitive, nameAlg);
984
985         // Apply inner wrap for duplication blob.  It includes both integrity and
986         // encryption
987         if(doInnerWrap)
988         {
989             BYTE            *innerBuffer = NULL;
990             BOOL            symKeyInput = TRUE;
991             innerBuffer = outPrivate->t.buffer;
992             // Skip outer integrity space
993             if(doOuterWrap)
994                 innerBuffer += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
995             dataSize = ProduceInnerIntegrity(name, innerHash, dataSize,
996                                              innerBuffer);
997             // Generate inner encryption key if needed
998             if(innerSymKey->t.size == 0)
999             {
1000                innerSymKey->t.size = (symDef->keyBits.sym + 7) / 8;
1001                CryptRandomGenerate(innerSymKey->t.size, innerSymKey->t.buffer);
1002
1003                // TPM generates symmetric encryption.  Set the flag to FALSE
1004                symKeyInput = FALSE;
1005            }
1006            else
1007            {
1008                // assume the input key size should matches the symmetric definition
1009                pAssert(innerSymKey->t.size == (symDef->keyBits.sym + 7) / 8);
1010            }
1011
1012            // Encrypt inner buffer in place
1013            CryptSymmetricEncrypt(innerBuffer, symDef->algorithm,
1014                                  symDef->keyBits.sym, innerSymKey->t.buffer, NULL,
1015                                  TPM_ALG_CFB, dataSize, innerBuffer);
1016
1017            // If the symmetric encryption key is imported, clear the buffer for
1018            // output
1019            if(symKeyInput)
1020                innerSymKey->t.size = 0;
1021        }
1022        // Apply outer wrap for duplication blob.  It includes both integrity and
1023        // encryption
1024        if(doOuterWrap)
1025        {
1026            dataSize = ProduceOuterWrap(parent, name, outerHash, seed, FALSE,
1027                                        dataSize, outPrivate->t.buffer);
1028        }
1029        // Data size for output
1030        outPrivate->t.size = dataSize;
1031
1032        return;
1033    }
```

7.6.3.14 DuplicateToSensitive()

Unwrap a duplication blob. Check the integrity, decrypt and retrieve data to a sensitive structure. The operations in this function:

 Copyright © TCG 2006-2020

a) check the integrity HMAC of the input private area
b) decrypt the private buffer
c) unmarshal TPMT_SENSITIVE structure into the buffer of TPMT_SENSITIVE

Error Returns	Meaning
TPM_RC_INSUFFICIENT	unmarshaling sensitive data from *inPrivate* failed
TPM_RC_INTEGRITY	*inPrivate* data integrity is broken
TPM_RC_SIZE	unmarshaling sensitive data from *inPrivate* failed

```
1034 TPM_RC
1035 DuplicateToSensitive(
1036     TPM2B                 *inPrivate,     // IN: input private structure
1037     TPM2B                 *name,          // IN: the name of the object
1038     OBJECT                *parent,        // IN: the parent
1039     TPM_ALG_ID             nameAlg,       // IN: hash algorithm in public area.
1040     TPM2B                 *seed,          // IN: an external seed may be provided.
1041                                           //     If external seed is provided with
1042                                           //     size of 0, no outer wrap is
1043                                           //     applied
1044     TPMT_SYM_DEF_OBJECT *symDef,          // IN: Symmetric key definition. If the
1045                                           //     symmetric key algorithm is NULL,
1046                                           //     no inner wrap is applied
1047     TPM2B                 *innerSymKey,   // IN: a symmetric key may be provided
1048                                           //     to decrypt the inner wrap of a
1049                                           //     duplication blob.
1050     TPMT_SENSITIVE        *sensitive      // OUT: sensitive structure
1051     )
1052 {
1053     TPM_RC                 result;
1054     BYTE                  *buffer;
1055     INT32                  size;
1056     BYTE                  *sensitiveData; // pointer to the sensitive data
1057     UINT16                 dataSize;
1058     UINT16                 dataSizeInput;
1059 //
1060     // Make sure that name is provided
1061     pAssert(name != NULL && name->size != 0);
1062
1063     // Make sure symDef and innerSymKey are not NULL
1064     pAssert(symDef != NULL && innerSymKey != NULL);
1065
1066     // Starting of sensitive data
1067     sensitiveData = inPrivate->buffer;
1068     dataSize = inPrivate->size;
1069
1070     // Find out if outer wrap is applied
1071     if(seed->size != 0)
1072     {
1073         // Use parent nameAlg as outer hash algorithm
1074         TPMI_ALG_HASH    outerHash = parent->publicArea.nameAlg;
1075
1076         result = UnwrapOuter(parent, name, outerHash, seed, FALSE,
1077                              dataSize, sensitiveData);
1078         if(result != TPM_RC_SUCCESS)
1079             return result;
1080         // Adjust sensitive data pointer and size
1081         sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1082         dataSize -= sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1083     }
1084     // Find out if inner wrap is applied
1085     if(symDef->algorithm != TPM_ALG_NULL)
```

```
1086        {
1087            // assume the input key size matches the symmetric definition
1088            pAssert(innerSymKey->size == (symDef->keyBits.sym + 7) / 8);
1089
1090            // Decrypt inner buffer in place
1091            CryptSymmetricDecrypt(sensitiveData, symDef->algorithm,
1092                                  symDef->keyBits.sym, innerSymKey->buffer, NULL,
1093                                  TPM_ALG_CFB, dataSize, sensitiveData);
1094            // Check inner integrity
1095            result = CheckInnerIntegrity(name, nameAlg, dataSize, sensitiveData);
1096            if(result != TPM_RC_SUCCESS)
1097                return result;
1098            // Adjust sensitive data pointer and size
1099            sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(nameAlg);
1100            dataSize -= sizeof(UINT16) + CryptHashGetDigestSize(nameAlg);
1101        }
1102        // Unmarshal input data size
1103        buffer = sensitiveData;
1104        size = (INT32)dataSize;
1105        result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
1106        if(result == TPM_RC_SUCCESS)
1107        {
1108            if((dataSizeInput + sizeof(UINT16)) != dataSize)
1109                result = TPM_RC_SIZE;
1110            else
1111            {
1112                // Unmarshal sensitive buffer to sensitive structure
1113                result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
1114
1115                // if the results is OK make sure that all the data was unmarshaled
1116                if(result == TPM_RC_SUCCESS && size != 0)
1117                    result = TPM_RC_SIZE;
1118            }
1119        }
1120        return result;
1121    }
```

7.6.3.15 SecretToCredential()

This function prepare the credential blob from a secret (a TPM2B_DIGEST) The operations in this function:

a) marshal TPM2B_DIGEST structure into the buffer of TPM2B_ID_OBJECT

b) encrypt the private buffer, excluding the leading integrity HMAC area

c) compute integrity HMAC and append to the beginning of the buffer.

d) Set the total size of TPM2B_ID_OBJECT buffer

```
1122    void
1123    SecretToCredential(
1124        TPM2B_DIGEST        *secret,        // IN: secret information
1125        TPM2B               *name,          // IN: the name of the object
1126        TPM2B               *seed,          // IN: an external seed.
1127        OBJECT              *protector,     // IN: the protector
1128        TPM2B_ID_OBJECT     *outIDObject    // OUT: output credential
1129        )
1130    {
1131        BYTE                *buffer;        // Auxiliary buffer pointer
1132        BYTE                *sensitiveData; // pointer to the sensitive data
1133        TPMI_ALG_HASH        outerHash;     // The hash algorithm for outer wrap
1134        UINT16               dataSize;      // data blob size
1135    //
1136        pAssert(secret != NULL && outIDObject != NULL);
```

 Copyright © TCG 2006-2020

```
1137
1138        // use protector's name algorithm as outer hash ????
1139        outerHash = protector->publicArea.nameAlg;
1140
1141        // Marshal secret area to credential buffer, leave space for integrity
1142        sensitiveData = outIDObject->t.credential
1143            + sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1144    // Marshal secret area
1145        buffer = sensitiveData;
1146        dataSize = TPM2B_DIGEST_Marshal(secret, &buffer, NULL);
1147
1148        // Apply outer wrap
1149        outIDObject->t.size = ProduceOuterWrap(protector, name, outerHash, seed, FALSE,
1150                                               dataSize, outIDObject->t.credential);
1151        return;
1152    }
```

7.6.3.16 CredentialToSecret()

Unwrap a credential. Check the integrity, decrypt and retrieve data to a TPM2B_DIGEST structure. The operations in this function:

a) check the integrity HMAC of the input credential area

b) decrypt the credential buffer

c) unmarshal TPM2B_DIGEST structure into the buffer of TPM2B_DIGEST

Error Returns	Meaning
TPM_RC_INSUFFICIENT	error during credential unmarshaling
TPM_RC_INTEGRITY	credential integrity is broken
TPM_RC_SIZE	error during credential unmarshaling
TPM_RC_VALUE	IV size does not match the encryption algorithm block size

```
1153    TPM_RC
1154    CredentialToSecret(
1155        TPM2B                *inIDObject,     // IN: input credential blob
1156        TPM2B                *name,           // IN: the name of the object
1157        TPM2B                *seed,           // IN: an external seed.
1158        OBJECT               *protector,      // IN: the protector
1159        TPM2B_DIGEST         *secret          // OUT: secret information
1160        )
1161    {
1162        TPM_RC                result;
1163        BYTE                 *buffer;
1164        INT32                 size;
1165        TPMI_ALG_HASH         outerHash;     // The hash algorithm for outer wrap
1166        BYTE                 *sensitiveData; // pointer to the sensitive data
1167        UINT16                dataSize;
1168    //
1169        // use protector's name algorithm as outer hash
1170        outerHash = protector->publicArea.nameAlg;
1171
1172        // Unwrap outer, a TPM_RC_INTEGRITY error may be returned at this point
1173        result = UnwrapOuter(protector, name, outerHash, seed, FALSE,
1174                             inIDObject->size, inIDObject->buffer);
1175        if(result == TPM_RC_SUCCESS)
1176        {
1177            // Compute the beginning of sensitive data
1178            sensitiveData = inIDObject->buffer
1179                + sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1180            dataSize = inIDObject->size
```

```
1181                - (sizeof(UINT16) + CryptHashGetDigestSize(outerHash));
1182            // Unmarshal secret buffer to TPM2B_DIGEST structure
1183            buffer = sensitiveData;
1184            size = (INT32)dataSize;
1185            result = TPM2B_DIGEST_Unmarshal(secret, &buffer, &size);
1186
1187            // If there were no other unmarshaling errors, make sure that the
1188            // expected amount of data was recovered
1189            if(result == TPM_RC_SUCCESS && size != 0)
1190                return TPM_RC_SIZE;
1191        }
1192        return result;
1193    }
```

7.6.3.17 MemoryRemoveTrailingZeros()

This function is used to adjust the length of an authorization value. It adjusts the size of the TPM2B so that it does not include octets at the end of the buffer that contain zero. The function returns the number of non-zero octets in the buffer.

```
1194    UINT16
1195    MemoryRemoveTrailingZeros(
1196        TPM2B_AUTH      *auth           // IN/OUT: value to adjust
1197        )
1198    {
1199        while((auth->t.size > 0) && (auth->t.buffer[auth->t.size - 1] == 0))
1200            auth->t.size--;
1201        return auth->t.size;
1202    }
```

7.6.3.18 SetLabelAndContext()

This function sets the label and context for a derived key. It is possible that *label* or *context* can end up being an Empty Buffer.

```
1203    TPM_RC
1204    SetLabelAndContext(
1205        TPMS_DERIVE             *labelContext,  // IN/OUT: the recovered label and
1206                                                //      context
1207        TPM2B_SENSITIVE_DATA    *sensitive      // IN: the sensitive data
1208        )
1209    {
1210        TPMS_DERIVE              sensitiveValue;
1211        TPM_RC                   result;
1212        INT32                    size;
1213        BYTE                    *buff;
1214    //
1215        // Unmarshal a TPMS_DERIVE from the TPM2B_SENSITIVE_DATA buffer
1216        // If there is something to unmarshal...
1217        if(sensitive->t.size != 0)
1218        {
1219            size = sensitive->t.size;
1220            buff = sensitive->t.buffer;
1221            result = TPMS_DERIVE_Unmarshal(&sensitiveValue, &buff, &size);
1222            if(result != TPM_RC_SUCCESS)
1223                return result;
1224            // If there was a label in the public area leave it there, otherwise, copy
1225            // the new value
1226            if(labelContext->label.t.size == 0)
1227                MemoryCopy2B(&labelContext->label.b, &sensitiveValue.label.b,
1228                             sizeof(labelContext->label.t.buffer));
1229            // if there was a context string in publicArea, it overrides
```

```
1230            if(labelContext->context.t.size == 0)
1231                MemoryCopy2B(&labelContext->context.b, &sensitiveValue.context.b,
1232                             sizeof(labelContext->label.t.buffer));
1233        }
1234        return TPM_RC_SUCCESS;
1235    }
```

7.6.3.19 UnmarshalToPublic()

Support function to unmarshal the template. This is used because the Input may be a TPMT_TEMPLATE and that structure does not have the same size as a TPMT_PUBLIC because of the difference between the *unique* and *seed* fields. If *derive* is not NULL, then the *seed* field is assumed to contain a *label* and *context* that are unmarshaled into *derive*.

```
1236    TPM_RC
1237    UnmarshalToPublic(
1238        TPMT_PUBLIC          *tOut,       // OUT: output
1239        TPM2B_TEMPLATE       *tIn,        // IN:
1240        BOOL                  derivation, // IN: indicates if this is for a derivation
1241        TPMS_DERIVE          *labelContext// OUT: label and context if derivation
1242        )
1243    {
1244        BYTE                 *buffer = tIn->t.buffer;
1245        INT32                 size = tIn->t.size;
1246        TPM_RC                result;
1247    //
1248        // make sure that tOut is zeroed so that there are no remnants from previous
1249        // uses
1250        MemorySet(tOut, 0, sizeof(TPMT_PUBLIC));
1251        // Unmarshal the components of the TPMT_PUBLIC up to the unique field
1252        result = TPMI_ALG_PUBLIC_Unmarshal(&tOut->type, &buffer, &size);
1253        if(result != TPM_RC_SUCCESS)
1254            return result;
1255        result = TPMI_ALG_HASH_Unmarshal(&tOut->nameAlg, &buffer, &size, FALSE);
1256        if(result != TPM_RC_SUCCESS)
1257            return result;
1258        result = TPMA_OBJECT_Unmarshal(&tOut->objectAttributes, &buffer, &size);
1259        if(result != TPM_RC_SUCCESS)
1260            return result;
1261        result = TPM2B_DIGEST_Unmarshal(&tOut->authPolicy, &buffer, &size);
1262        if(result != TPM_RC_SUCCESS)
1263            return result;
1264        result = TPMU_PUBLIC_PARMS_Unmarshal(&tOut->parameters, &buffer, &size,
1265                                             tOut->type);
1266        if(result != TPM_RC_SUCCESS)
1267            return result;
1268        // Now unmarshal a TPMS_DERIVE if this is for derivation
1269        if(derivation)
1270            result = TPMS_DERIVE_Unmarshal(labelContext, &buffer, &size);
1271        else
1272            // otherwise, unmarshal a TPMU_PUBLIC_ID
1273            result = TPMU_PUBLIC_ID_Unmarshal(&tOut->unique, &buffer, &size,
1274                                              tOut->type);
1275        // Make sure the template was used up
1276        if((result == TPM_RC_SUCCESS) && (size != 0))
1277            result = TPM_RC_SIZE;
1278        return result;
1279    }
```

7.6.3.20 ObjectSetExternal()

Set the external attributes for an object.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1280 void
1281 ObjectSetExternal(
1282     OBJECT      *object
1283     )
1284 {
1285     object->attributes.external = SET;
1286 }
```

 Copyright © TCG 2006-2020

7.7 Encrypt Decrypt Support (EncryptDecrypt_spt.c)

```
1   #include "Tpm.h"
2   #include "EncryptDecrypt_fp.h"
3   #include "EncryptDecrypt_spt_fp.h"
4   #if CC_EncryptDecrypt2
```

Error Returns	Meaning
TPM_RC_KEY	is not a symmetric decryption key with both public and private portions loaded
TPM_RC_SIZE	*IvIn* size is incompatible with the block cipher mode; or *inData* size is not an even multiple of the block size for CBC or ECB mode
TPM_RC_VALUE	*keyHandle* is restricted and the argument *mode* does not match the key's mode

```
5   TPM_RC
6   EncryptDecryptShared(
7       TPMI_DH_OBJECT          keyHandleIn,
8       TPMI_YES_NO             decryptIn,
9       TPMI_ALG_SYM_MODE       modeIn,
10      TPM2B_IV                *ivIn,
11      TPM2B_MAX_BUFFER        *inData,
12      EncryptDecrypt_Out      *out
13      )
14  {
15      OBJECT                *symKey;
16      UINT16                 keySize;
17      UINT16                 blockSize;
18      BYTE                  *key;
19      TPM_ALG_ID             alg;
20      TPM_ALG_ID             mode;
21      TPM_RC                 result;
22      BOOL                   OK;
23  // Input Validation
24      symKey = HandleToObject(keyHandleIn);
25      mode = symKey->publicArea.parameters.symDetail.sym.mode.sym;
26
27      // The input key should be a symmetric key
28      if(symKey->publicArea.type != TPM_ALG_SYMCIPHER)
29          return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;
30      // The key must be unrestricted and allow the selected operation
31      OK = !IS_ATTRIBUTE(symKey->publicArea.objectAttributes,
32                         TPMA_OBJECT, restricted);
33      if(YES == decryptIn)
34          OK = OK && IS_ATTRIBUTE(symKey->publicArea.objectAttributes,
35                                  TPMA_OBJECT, decrypt);
36      else
37          OK = OK && IS_ATTRIBUTE(symKey->publicArea.objectAttributes,
38                                  TPMA_OBJECT, sign);
39      if(!OK)
40          return TPM_RCS_ATTRIBUTES + RC_EncryptDecrypt_keyHandle;
41
42      // Make sure that key is an encrypt/decrypt key and not SMAC
43      if(!CryptSymModeIsValid(mode, TRUE))
44         return TPM_RCS_MODE + RC_EncryptDecrypt_keyHandle;
45
46      // If the key mode is not TPM_ALG_NULL...
47      // or TPM_ALG_NULL
48      if(mode != TPM_ALG_NULL)
49      {
50          // then the input mode has to be TPM_ALG_NULL or the same as the key
```

 Copyright © TCG 2006-2020

```
51           if((modeIn != TPM_ALG_NULL) && (modeIn != mode))
52               return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
53       }
54       else
55       {
56           // if the key mode is null, then the input can't be null
57           if(modeIn == TPM_ALG_NULL)
58               return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
59           mode = modeIn;
60       }
61       // The input iv for ECB mode should be an Empty Buffer.  All the other modes
62       // should have an iv size same as encryption block size
63       keySize = symKey->publicArea.parameters.symDetail.sym.keyBits.sym;
64       alg = symKey->publicArea.parameters.symDetail.sym.algorithm;
65       blockSize = CryptGetSymmetricBlockSize(alg, keySize);
66
67       // reverify the algorithm. This is mainly to keep static analysis tools happy
68       if(blockSize == 0)
69           return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;
70
71       // Note: When an algorithm is not supported by a TPM, the TPM_ALG_xxx for that
72       // algorithm is not defined. However, it is assumed that the ALG_xxx_VALUE for
73       // the algorithm is always defined. Both have the same numeric value.
74       // ALG_xxx_VALUE is used here so that the code does not get cluttered with
75       // #ifdef's. Having this check does not mean that the algorithm is supported.
76       // If it was not supported the unmarshaling code would have rejected it before
77       // this function were called. This means that, depending on the implementation,
78       // the check could be redundant but it doesn't hurt.
79       if(((mode == ALG_ECB_VALUE) && (ivIn->t.size != 0))
80          || ((mode != ALG_ECB_VALUE) && (ivIn->t.size != blockSize)))
81           return TPM_RCS_SIZE + RC_EncryptDecrypt_ivIn;
82
83       // The input data size of CBC mode or ECB mode must be an even multiple of
84       // the symmetric algorithm's block size
85       if(((mode == ALG_CBC_VALUE) || (mode == ALG_ECB_VALUE))
86          && ((inData->t.size % blockSize) != 0))
87           return TPM_RCS_SIZE + RC_EncryptDecrypt_inData;
88
89       // Copy IV
90       // Note: This is copied here so that the calls to the encrypt/decrypt functions
91       // will modify the output buffer, not the input buffer
92       out->ivOut = *ivIn;
93
94   // Command Output
95       key = symKey->sensitive.sensitive.sym.t.buffer;
96       // For symmetric encryption, the cipher data size is the same as plain data
97       // size.
98       out->outData.t.size = inData->t.size;
99       if(decryptIn == YES)
100      {
101          // Decrypt data to output
102          result = CryptSymmetricDecrypt(out->outData.t.buffer, alg, keySize, key,
103                                         &(out->ivOut), mode, inData->t.size,
104                                         inData->t.buffer);
105      }
106      else
107      {
108          // Encrypt data to output
109          result = CryptSymmetricEncrypt(out->outData.t.buffer, alg, keySize, key,
110                                         &(out->ivOut), mode, inData->t.size,
111                                         inData->t.buffer);
112      }
113      return result;
114  }
115  #endif // CC_EncryptDecrypt
```

7.8 ACT Support (ACT_spt.c)

7.8.1 Introduction

This code implements the ACT update code. It does not use a mutex. This code uses a platform service (_plat__ACT_UpdateCounter()) that returns *false* if the update is not accepted. If this occurs, then TPM_RC_RETRY should be sent to the caller so that they can retry the operation later. The implementation of this is platform dependent but the reference uses a simple flag to indicate that an update is pending and the only process that can clear that flag is the process that does the actual update.

7.8.2 Includes

```
1     #include "Tpm.h"
2     #include "ACT_spt_fp.h"
3     #include "Platform_fp.h"
```

7.8.3 Functions

7.8.3.1 _ActResume()

This function does the resume processing for an ACT. It updates the saved count and turns signaling back on if necessary.

```
4     static void
5     _ActResume(
6         UINT32              act,            //IN: the act number
7         ACT_STATE          *actData         //IN: pointer to the saved ACT data
8     )
9     {
10        // If the act was non-zero, then restore the counter value.
11        if(actData->remaining > 0)
12            _plat__ACT_UpdateCounter(act, actData->remaining);
13        // if the counter was zero and the ACT signaling, enable the signaling.
14        else if(go.signaledACT & (1 << act))
15            _plat__ACT_SetSignaled(act, TRUE);
16    }
```

7.8.3.2 ActStartup()

This function is called by TPM2_Startup() to initialize the ACT counter values.

```
17    BOOL
18    ActStartup(
19        STARTUP_TYPE        type
20    )
21    {
22        // Reset all the ACT hardware
23        _plat__ACT_Initialize();
24
25        // For TPM_RESET or TPM_RESTART, the ACTs will all be disabled and the output
26        // de-asserted.
27        if(type != SU_RESUME)
28        {
29            go.signaledACT = 0;
30    #define CLEAR_ACT_POLICY(N)                                                     \
31            go.ACT_##N.hashAlg = TPM_ALG_NULL;                                      \
32            go.ACT_##N.authPolicy.b.size = 0;
33
```

 Copyright © TCG 2006-2020

```
34            FOR_EACH_ACT(CLEAR_ACT_POLICY)
35
36        }
37        else
38        {
39            // Resume each of the implemented ACT
40    #define RESUME_ACT(N)    _ActResume(0x##N, &go.ACT_##N);
41
42            FOR_EACH_ACT(RESUME_ACT)
43        }
44        s_ActUpdated = 0;
45        _plat__ACT_EnableTicks(TRUE);
46        return TRUE;
47    }
```

7.8.3.3 _ActSaveState()

Get the counter state and the signaled state for an ACT. If the ACT has not been updated since the last time it was saved, then divide the count by 2.

```
48    static void
49    _ActSaveState(
50        UINT32              act,
51        P_ACT_STATE         actData
52    )
53    {
54        actData->remaining = _plat__ACT_GetRemaining(act);
55        // If the ACT hasn't been updated since the last startup, then it should be
56        // be halved.
57        if((s_ActUpdated & (1 << act)) == 0)
58        {
59            // Don't halve if the count is set to max or if halving would make it zero
60            if((actData->remaining != UINT32_MAX) && (actData->remaining > 1))
61                actData->remaining /= 2;
62        }
63        if(_plat__ACT_GetSignaled(act))
64            go.signaledACT |= (1 << act);
65    }
```

7.8.3.4 ActGetSignaled()

This function returns the state of the signaled flag associated with an ACT.

```
66    BOOL
67    ActGetSignaled(
68        TPM_RH              actHandle
69    )
70    {
71        UINT32              act = actHandle - TPM_RH_ACT_0;
72    //
73        return _plat__ACT_GetSignaled(act);
74    }
```

7.8.3.5 ActShutdown()

This function saves the current state of the counters

```
75    BOOL
76    ActShutdown(
77        TPM_SU              state       //IN: the type of the shutdown.
78    )
```

```
79  {
80      // if this is not shutdown state, then the only type of startup is TPM_RESTART
81      // so the timer values will be cleared. If this is shutdown state, get the current
82      // countdown and signaled values. Plus, if the counter has not been updated
83      // since the last restart, divide the time by 2 so that there is no attack on the
84      // countdown by saving the countdown state early and then not using the TPM.
85      if(state == TPM_SU_STATE)
86      {
87          // This will be populated as each of the ACT is queried
88          go.signaledACT = 0;
89          // Get the current count and the signaled state
90  #define SAVE_ACT_STATE(N) _ActSaveState(0x##N, &go.ACT_##N);
91
92          FOR_EACH_ACT(SAVE_ACT_STATE);
93      }
94      return TRUE;
95  }
```

7.8.3.6 ActIsImplemented()

This function determines if an ACT is implemented in both the TPM and the platform code.

```
96  BOOL
97  ActIsImplemented(
98      UINT32          act
99  )
100 {
101 #define CASE_ACT_
102     // This switch accounts for the TPM implemente values.
103     switch(act)
104     {
105         FOR_EACH_ACT(CASE_ACT_NUMBER)
106             // This ensures that the platorm implementes the values implemented by
107             // the TPM
108             return _plat__ACT_GetImplemented(act);
109         default:
110             break;
111     }
112     return FALSE;
113 }
```

7.8.3.7 ActCounterUpdate()

This function updates the ACT counter. If the counter already has a pending update, it returns TPM_RC_RETRY so that the update can be tried again later.

```
114 TPM_RC
115 ActCounterUpdate(
116     TPM_RH          handle,         //IN: the handle of the act
117     UINT32          newValue        //IN: the value to set in the ACT
118 )
119 {
120     UINT32          act;
121     TPM_RC          result;
122 //
123     act = handle - TPM_RH_ACT_0;
124     // This should never fail, but...
125     if(!_plat__ACT_GetImplemented(act))
126         result = TPM_RC_VALUE;
127     else
128     {
129     // Will need to clear orderly so fail if we are orderly and NV is not available
130         if(NV_IS_ORDERLY)
```

```
131                 RETURN_IF_NV_IS_NOT_AVAILABLE;
132             // if the attempt to update the counter fails, it means that there is an
133             // update pending so wait until it has occurred and then do an update.
134             if(!_plat__ACT_UpdateCounter(act, newValue))
135                 result = TPM_RC_RETRY;
136             else
137             {
138                 // Indicate that the ACT has been updated since last TPM2_Startup().
139                 s_ActUpdated |= (UINT16)(1 << act);
140
141                 // Need to clear the orderly flag
142                 g_clearOrderly = TRUE;
143
144                 result = TPM_RC_SUCCESS;
145             }
146         }
147         return result;
148     }
```

7.8.3.8 ActGetCapabilityData()

This function returns the list of ACT data

Return Value	Meaning
YES	if more ACT data is available
NO	if no more ACT data to

```
149 TPMI_YES_NO
150 ActGetCapabilityData(
151     TPM_HANDLE       actHandle,     // IN: the handle for the starting ACT
152     UINT32           maxCount,      // IN: maximum allowed return values
153     TPML_ACT_DATA   *actList        // OUT: ACT data list
154 )
155 {
156     // Initialize output property list
157     actList->count = 0;
158
159     // Make sure that the starting handle value is in range (again)
160     if((actHandle < TPM_RH_ACT_0) || (actHandle > TPM_RH_ACT_F))
161         return FALSE;
162     // The maximum count of curves we may return is MAX_ECC_CURVES
163     if(maxCount > MAX_ACT_DATA)
164         maxCount = MAX_ACT_DATA;
165     // Scan the ACT data from the starting ACT
166     for(; actHandle <= TPM_RH_ACT_F; actHandle++)
167     {
168         UINT32          act = actHandle - TPM_RH_ACT_0;
169         if(actList->count < maxCount)
170         {
171             if(ActIsImplemented(act))
172             {
173                 TPMS_ACT_DATA    *actData = &actList->actData[actList->count];
174             //
175                 memset(&actData->attributes, 0, sizeof(actData->attributes));
176                 actData->handle = actHandle;
177                 actData->timeout = _plat__ACT_GetRemaining(act);
178                 actData->attributes.signaled = _plat__ACT_GetSignaled(act);
179                 actList->count++;
180             }
181         }
182         else
183         {
```

```
184             if(_plat__ACT_GetImplemented(act))
185                 return YES;
186         }
187     }
188     // If we get here, either all of the ACT values were put in the list, or the list
189     // was filled and there are no more ACT values to return
190     return NO;
191 }
```


 Copyright © TCG 2006-2020

8 Subsystem

8.1 CommandAudit.c

8.1.1 Introduction

This file contains the functions that support command audit.

8.1.2 Includes

```
1   #include "Tpm.h"
```

8.1.3 Functions

8.1.3.1 CommandAuditPreInstall_Init()

This function initializes the command audit list. This function simulates the behavior of manufacturing. A function is used instead of a structure definition because this is easier than figuring out the initialization value for a bit array.

This function would not be implemented outside of a manufacturing or simulation environment.

```
2   void
3   CommandAuditPreInstall_Init(
4       void
5       )
6   {
7       // Clear all the audit commands
8       MemorySet(gp.auditCommands, 0x00, sizeof(gp.auditCommands));
9
10      // TPM_CC_SetCommandCodeAuditStatus always being audited
11      CommandAuditSet(TPM_CC_SetCommandCodeAuditStatus);
12
13      // Set initial command audit hash algorithm to be context integrity hash
14      // algorithm
15      gp.auditHashAlg = CONTEXT_INTEGRITY_HASH_ALG;
16
17      // Set up audit counter to be 0
18      gp.auditCounter = 0;
19
20      // Write command audit persistent data to NV
21      NV_SYNC_PERSISTENT(auditCommands);
22      NV_SYNC_PERSISTENT(auditHashAlg);
23      NV_SYNC_PERSISTENT(auditCounter);
24
25      return;
26  }
```

8.1.3.2 CommandAuditStartup()

This function clears the command audit digest on a TPM Reset.

```
27  BOOL
28  CommandAuditStartup(
29      STARTUP_TYPE     type            // IN: start up type
30      )
31  {
32      if((type != SU_RESTART) && (type != SU_RESUME))
```

 Copyright © TCG 2006-2020

```
33          {
34              // Reset the digest size to initialize the digest
35              gr.commandAuditDigest.t.size = 0;
36          }
37          return TRUE;
38      }
```

8.1.3.3 CommandAuditSet()

This function will SET the audit flag for a command. This function will not SET the audit flag for a command that is not implemented. This ensures that the audit status is not SET when TPM2_GetCapability() is used to read the list of audited commands.

This function is only used by TPM2_SetCommandCodeAuditStatus().

The actions in TPM2_SetCommandCodeAuditStatus() are expected to cause the changes to be saved to NV after it is setting and clearing bits.

Return Value	Meaning
TRUE(1)	command code audit status was changed
FALSE(0)	command code audit status was not changed

```
39      BOOL
40      CommandAuditSet(
41          TPM_CC            commandCode      // IN: command code
42          )
43      {
44          COMMAND_INDEX        commandIndex = CommandCodeToCommandIndex(commandCode);
45
46          // Only SET a bit if the corresponding command is implemented
47          if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
48          {
49              // Can't audit shutdown
50              if(commandCode != TPM_CC_Shutdown)
51              {
52                  if(!TEST_BIT(commandIndex, gp.auditCommands))
53                  {
54                      // Set bit
55                      SET_BIT(commandIndex, gp.auditCommands);
56                      return TRUE;
57                  }
58              }
59          }
60          // No change
61          return FALSE;
62      }
```

8.1.3.4 CommandAuditClear()

This function will CLEAR the audit flag for a command. It will not CLEAR the audit flag for TPM_CC_SetCommandCodeAuditStatus().

This function is only used by TPM2_SetCommandCodeAuditStatus().

The actions in TPM2_SetCommandCodeAuditStatus() are expected to cause the changes to be saved to NV after it is setting and clearing bits.

Return Value	Meaning
TRUE(1)	command code audit status was changed
FALSE(0)	command code audit status was not changed

```
63  BOOL
64  CommandAuditClear(
65      TPM_CC              commandCode     // IN: command code
66      )
67  {
68      COMMAND_INDEX       commandIndex = CommandCodeToCommandIndex(commandCode);
69
70      // Do nothing if the command is not implemented
71      if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
72      {
73          // The bit associated with TPM_CC_SetCommandCodeAuditStatus() cannot be
74          // cleared
75          if(commandCode != TPM_CC_SetCommandCodeAuditStatus)
76          {
77              if(TEST_BIT(commandIndex, gp.auditCommands))
78              {
79                  // Clear bit
80                  CLEAR_BIT(commandIndex, gp.auditCommands);
81                  return TRUE;
82              }
83          }
84      }
85      // No change
86      return FALSE;
87  }
```

8.1.3.5 CommandAuditIsRequired()

This function indicates if the audit flag is SET for a command.

Return Value	Meaning
TRUE(1)	command is audited
FALSE(0)	command is not audited

```
88  BOOL
89  CommandAuditIsRequired(
90      COMMAND_INDEX       commandIndex    // IN: command index
91      )
92  {
93      // Check the bit map.  If the bit is SET, command audit is required
94      return(TEST_BIT(commandIndex, gp.auditCommands));
95  }
```

8.1.3.6 CommandAuditCapGetCCList()

This function returns a list of commands that have their audit bit SET.

The list starts at the input *commandCode*.

 Copyright © TCG 2006-2020

Return Value	Meaning
YES	if there are more command code available
NO	all the available command code has been returned

```
 96  TPMI_YES_NO
 97  CommandAuditCapGetCCList(
 98      TPM_CC           commandCode,    // IN: start command code
 99      UINT32           count,          // IN: count of returned TPM_CC
100      TPML_CC         *commandList     // OUT: list of TPM_CC
101      )
102  {
103      TPMI_YES_NO     more = NO;
104      COMMAND_INDEX   commandIndex;
105
106      // Initialize output handle list
107      commandList->count = 0;
108
109      // The maximum count of command we may return is MAX_CAP_CC
110      if(count > MAX_CAP_CC) count = MAX_CAP_CC;
111
112      // Find the implemented command that has a command code that is the same or
113      // higher than the input
114      // Collect audit commands
115      for(commandIndex = GetClosestCommandIndex(commandCode);
116      commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
117          commandIndex = GetNextCommandIndex(commandIndex))
118      {
119          if(CommandAuditIsRequired(commandIndex))
120          {
121              if(commandList->count < count)
122              {
123                  // If we have not filled up the return list, add this command
124                  // code to its
125                  TPM_CC      cc = GET_ATTRIBUTE(s_ccAttr[commandIndex],
126                                                 TPMA_CC, commandIndex);
127                  if(IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
128                      cc += (1 << 29);
129                  commandList->commandCodes[commandList->count] = cc;
130                  commandList->count++;
131              }
132              else
133              {
134                  // If the return list is full but we still have command
135                  // available, report this and stop iterating
136                  more = YES;
137                  break;
138              }
139          }
140      }
141
142      return more;
143  }
```

8.1.3.7 CommandAuditGetDigest

This command is used to create a digest of the commands being audited. The commands are processed in ascending numeric order with a list of TPM_CC being added to a hash. This operates as if all the audited command codes were concatenated and then hashed.

```
144  void
145  CommandAuditGetDigest(
```

 Copyright © TCG 2006-2020

```
146         TPM2B_DIGEST     *digest          // OUT: command digest
147         )
148     {
149         TPM_CC                          commandCode;
150         COMMAND_INDEX                   commandIndex;
151         HASH_STATE                      hashState;
152
153         // Start hash
154         digest->t.size = CryptHashStart(&hashState, gp.auditHashAlg);
155
156         // Add command code
157         for(commandIndex = 0; commandIndex < COMMAND_COUNT; commandIndex++)
158         {
159             if(CommandAuditIsRequired(commandIndex))
160             {
161                 commandCode = GetCommandCode(commandIndex);
162                 CryptDigestUpdateInt(&hashState, sizeof(commandCode), commandCode);
163             }
164         }
165
166         // Complete hash
167         CryptHashEnd2B(&hashState, &digest->b);
168
169         return;
170     }
```

 Copyright © TCG 2006-2020

8.2 DA.c

8.2.1 Introduction

This file contains the functions and data definitions relating to the dictionary attack logic.

8.2.2 Includes and Data Definitions

```
1   #define DA_C
2   #include "Tpm.h"
```

8.2.3 Functions

8.2.3.1 DAPreInstall_Init()

This function initializes the DA parameters to their manufacturer-default values. The default values are determined by a platform-specific specification.

This function should not be called outside of a manufacturing or simulation environment.

The DA parameters will be restored to these initial values by TPM2_Clear().

```
3   void
4   DAPreInstall_Init(
5       void
6       )
7   {
8       gp.failedTries = 0;
9       gp.maxTries = 3;
10      gp.recoveryTime = 1000;          // in seconds (~16.67 minutes)
11      gp.lockoutRecovery = 1000;       // in seconds
12      gp.lockOutAuthEnabled = TRUE;    // Use of lockoutAuth is enabled
13
14      // Record persistent DA parameter changes to NV
15      NV_SYNC_PERSISTENT(failedTries);
16      NV_SYNC_PERSISTENT(maxTries);
17      NV_SYNC_PERSISTENT(recoveryTime);
18      NV_SYNC_PERSISTENT(lockoutRecovery);
19      NV_SYNC_PERSISTENT(lockOutAuthEnabled);
20
21      return;
22  }
```

8.2.3.2 DAStartup()

This function is called by TPM2_Startup() to initialize the DA parameters. In the case of Startup(CLEAR), use of *lockoutAuth* will be enabled if the lockout recovery time is 0. Otherwise, *lockoutAuth* will not be enabled until the TPM has been continuously powered for the *lockoutRecovery* time.

This function requires that NV be available and not rate limiting.

```
23  BOOL
24  DAStartup(
25      STARTUP_TYPE     type            // IN: startup type
26      )
27  {
28      NOT_REFERENCED(type);
29  #if !ACCUMULATE_SELF_HEAL_TIMER
30      _plat__TimerWasReset();
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
31          s_selfHealTimer = 0;
32          s_lockoutTimer = 0;
33      #else
34          if(_plat__TimerWasReset())
35          {
36              if(!NV_IS_ORDERLY)
37              {
38                  // If shutdown was not orderly, then don't really know if go.time has
39                  // any useful value so reset the timer to 0. This is what the tick
40                  // was reset to
41                  s_selfHealTimer = 0;
42                  s_lockoutTimer = 0;
43              }
44              else
45              {
46                  // If we know how much time was accumulated at the last orderly shutdown
47                  // subtract that from the saved timer values so that they effectively
48                  // have the accumulated values
49                  s_selfHealTimer -= go.time;
50                  s_lockoutTimer -= go.time;
51              }
52          }
53      #endif
54
55          // For any Startup(), if lockoutRecovery is 0, enable use of lockoutAuth.
56          if(gp.lockoutRecovery == 0)
57          {
58              gp.lockOutAuthEnabled = TRUE;
59              // Record the changes to NV
60              NV_SYNC_PERSISTENT(lockOutAuthEnabled);
61          }
62
63          // If DA has not been disabled and the previous shutdown is not orderly
64          // failedTries is not already at its maximum then increment 'failedTries'
65          if(gp.recoveryTime != 0
66             && gp.failedTries < gp.maxTries
67             && !IS_ORDERLY(g_prevOrderlyState))
68          {
69      #if USE_DA_USED
70              gp.failedTries += g_daUsed;
71              g_daUsed = FALSE;
72      #else
73              gp.failedTries++;
74      #endif
75              // Record the change to NV
76              NV_SYNC_PERSISTENT(failedTries);
77          }
78          // Before Startup, the TPM will not do clock updates. At startup, need to
79          // do a time update which will do the DA update.
80          TimeUpdate();
81
82          return TRUE;
83      }
```

8.2.3.3 DARegisterFailure()

This function is called when a authorization failure occurs on an entity that is subject to dictionary-attack protection. When a DA failure is triggered, register the failure by resetting the relevant self-healing timer to the current time.

```
84      void
85      DARegisterFailure(
86          TPM_HANDLE          handle          // IN: handle for failure
87          )
```

```
88  {
89      // Reset the timer associated with lockout if the handle is the lockoutAuth.
90      if(handle == TPM_RH_LOCKOUT)
91          s_lockoutTimer = g_time;
92      else
93          s_selfHealTimer = g_time;
94      return;
95  }
```

8.2.3.4 DASelfHeal()

This function is called to check if sufficient time has passed to allow decrement of *failedTries* or to re-enable use of *lockoutAuth*.

This function should be called when the time interval is updated.

```
96  void
97  DASelfHeal(
98      void
99      )
100 {
101     // Regular authorization self healing logic
102     // If no failed authorization tries, do nothing.  Otherwise, try to
103     // decrease failedTries
104     if(gp.failedTries != 0)
105     {
106         // if recovery time is 0, DA logic has been disabled.  Clear failed tries
107         // immediately
108         if(gp.recoveryTime == 0)
109         {
110             gp.failedTries = 0;
111             // Update NV record
112             NV_SYNC_PERSISTENT(failedTries);
113         }
114         else
115         {
116             UINT64          decreaseCount;
117 #if 0 // Errata eliminates this code
118             // In the unlikely event that failedTries should become larger than
119             // maxTries
120             if(gp.failedTries > gp.maxTries)
121                 gp.failedTries = gp.maxTries;
122 #endif
123             // How much can failedTries be decreased
124
125             // Cast s_selfHealTimer to an int in case it became negative at
126             // startup
127             decreaseCount = ((g_time - (INT64)s_selfHealTimer) / 1000)
128                 / gp.recoveryTime;
129
130             if(gp.failedTries <= (UINT32)decreaseCount)
131                 // should not set failedTries below zero
132                 gp.failedTries = 0;
133             else
134                 gp.failedTries -= (UINT32)decreaseCount;
135
136             // the cast prevents overflow of the product
137             s_selfHealTimer += (decreaseCount * (UINT64)gp.recoveryTime) * 1000;
138             if(decreaseCount != 0)
139                 // If there was a change to the failedTries, record the changes
140                 // to NV
141                 NV_SYNC_PERSISTENT(failedTries);
142         }
143     }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
144
145     // LockoutAuth self healing logic
146     // If lockoutAuth is enabled, do nothing.  Otherwise, try to see if we
147     // may enable it
148     if(!gp.lockOutAuthEnabled)
149     {
150         // if lockout authorization recovery time is 0, a reboot is required to
151         // re-enable use of lockout authorization.  Self-healing would not
152         // apply in this case.
153         if(gp.lockoutRecovery != 0)
154         {
155             if(((g_time - (INT64)s_lockoutTimer) / 1000) >= gp.lockoutRecovery)
156             {
157                 gp.lockOutAuthEnabled = TRUE;
158                 // Record the changes to NV
159                 NV_SYNC_PERSISTENT(lockOutAuthEnabled);
160             }
161         }
162     }
163     return;
164 }
```

8.3 Hierarchy.c

8.3.1 Introduction

This file contains the functions used for managing and accessing the hierarchy-related values.

8.3.2 Includes

```
1   #include "Tpm.h"
```

8.3.3 Functions

8.3.3.1 HierarchyPreInstall()

This function performs the initialization functions for the hierarchy when the TPM is simulated. This function should not be called if the TPM is not in a manufacturing mode at the manufacturer, or in a simulated environment.

```
2   void
3   HierarchyPreInstall_Init(
4       void
5       )
6   {
7       // Allow lockout clear command
8       gp.disableClear = FALSE;
9
10      // Initialize Primary Seeds
11      gp.EPSeed.t.size = sizeof(gp.EPSeed.t.buffer);
12      gp.SPSeed.t.size = sizeof(gp.SPSeed.t.buffer);
13      gp.PPSeed.t.size = sizeof(gp.PPSeed.t.buffer);
14  #if (defined USE_PLATFORM_EPS) && (USE_PLATFORM_EPS != NO)
15      _plat__GetEPS(gp.EPSeed.t.size, gp.EPSeed.t.buffer);
16  #else
17      CryptRandomGenerate(gp.EPSeed.t.size, gp.EPSeed.t.buffer);
18  #endif
19      CryptRandomGenerate(gp.SPSeed.t.size, gp.SPSeed.t.buffer);
20      CryptRandomGenerate(gp.PPSeed.t.size, gp.PPSeed.t.buffer);
21
22      // Initialize owner, endorsement and lockout authorization
23      gp.ownerAuth.t.size = 0;
24      gp.endorsementAuth.t.size = 0;
25      gp.lockoutAuth.t.size = 0;
26
27      // Initialize owner, endorsement, and lockout policy
28      gp.ownerAlg = TPM_ALG_NULL;
29      gp.ownerPolicy.t.size = 0;
30      gp.endorsementAlg = TPM_ALG_NULL;
31      gp.endorsementPolicy.t.size = 0;
32      gp.lockoutAlg = TPM_ALG_NULL;
33      gp.lockoutPolicy.t.size = 0;
34
35      // Initialize ehProof, shProof and phProof
36      gp.phProof.t.size = sizeof(gp.phProof.t.buffer);
37      gp.shProof.t.size = sizeof(gp.shProof.t.buffer);
38      gp.ehProof.t.size = sizeof(gp.ehProof.t.buffer);
39      CryptRandomGenerate(gp.phProof.t.size, gp.phProof.t.buffer);
40      CryptRandomGenerate(gp.shProof.t.size, gp.shProof.t.buffer);
41      CryptRandomGenerate(gp.ehProof.t.size, gp.ehProof.t.buffer);
42
43      // Write hierarchy data to NV
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
44      NV_SYNC_PERSISTENT(disableClear);
45      NV_SYNC_PERSISTENT(EPSeed);
46      NV_SYNC_PERSISTENT(SPSeed);
47      NV_SYNC_PERSISTENT(PPSeed);
48      NV_SYNC_PERSISTENT(ownerAuth);
49      NV_SYNC_PERSISTENT(endorsementAuth);
50      NV_SYNC_PERSISTENT(lockoutAuth);
51      NV_SYNC_PERSISTENT(ownerAlg);
52      NV_SYNC_PERSISTENT(ownerPolicy);
53      NV_SYNC_PERSISTENT(endorsementAlg);
54      NV_SYNC_PERSISTENT(endorsementPolicy);
55      NV_SYNC_PERSISTENT(lockoutAlg);
56      NV_SYNC_PERSISTENT(lockoutPolicy);
57      NV_SYNC_PERSISTENT(phProof);
58      NV_SYNC_PERSISTENT(shProof);
59      NV_SYNC_PERSISTENT(ehProof);
60
61      return;
62  }
```

8.3.3.2 HierarchyStartup()

This function is called at TPM2_Startup() to initialize the hierarchy related values.

```
63  BOOL
64  HierarchyStartup(
65      STARTUP_TYPE     type             // IN: start up type
66      )
67  {
68      // phEnable is SET on any startup
69      g_phEnable = TRUE;
70
71      // Reset platformAuth, platformPolicy; enable SH and EH at TPM_RESET and
72      // TPM_RESTART
73      if(type != SU_RESUME)
74      {
75          gc.platformAuth.t.size = 0;
76          gc.platformPolicy.t.size = 0;
77          gc.platformAlg = TPM_ALG_NULL;
78
79          // enable the storage and endorsement hierarchies and the platformNV
80          gc.shEnable = gc.ehEnable = gc.phEnableNV = TRUE;
81      }
82
83      // nullProof and nullSeed are updated at every TPM_RESET
84      if((type != SU_RESTART) && (type != SU_RESUME))
85      {
86          gr.nullProof.t.size = sizeof(gr.nullProof.t.buffer);
87          CryptRandomGenerate(gr.nullProof.t.size, gr.nullProof.t.buffer);
88          gr.nullSeed.t.size = sizeof(gr.nullSeed.t.buffer);
89          CryptRandomGenerate(gr.nullSeed.t.size, gr.nullSeed.t.buffer);
90      }
91
92      return TRUE;
93  }
```

8.3.3.3 HierarchyGetProof()

This function finds the proof value associated with a hierarchy.It returns a pointer to the proof value.

```
94  TPM2B_PROOF *
95  HierarchyGetProof(
96      TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy constant
```

```
97       )
98   {
99       TPM2B_PROOF          *proof = NULL;
100
101      switch(hierarchy)
102      {
103          case TPM_RH_PLATFORM:
104              // phProof for TPM_RH_PLATFORM
105              proof = &gp.phProof;
106              break;
107          case TPM_RH_ENDORSEMENT:
108              // ehProof for TPM_RH_ENDORSEMENT
109              proof = &gp.ehProof;
110              break;
111          case TPM_RH_OWNER:
112              // shProof for TPM_RH_OWNER
113              proof = &gp.shProof;
114              break;
115          default:
116              // nullProof for TPM_RH_NULL or anything else
117              proof = &gr.nullProof;
118              break;
119      }
120      return proof;
121  }
```

8.3.3.4 HierarchyGetPrimarySeed()

This function returns the primary seed of a hierarchy.

```
122  TPM2B_SEED *
123  HierarchyGetPrimarySeed(
124      TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy
125      )
126  {
127      TPM2B_SEED           *seed = NULL;
128      switch(hierarchy)
129      {
130          case TPM_RH_PLATFORM:
131              seed = &gp.PPSeed;
132              break;
133          case TPM_RH_OWNER:
134              seed = &gp.SPSeed;
135              break;
136          case TPM_RH_ENDORSEMENT:
137              seed = &gp.EPSeed;
138              break;
139           default:
140              seed = &gr.nullSeed;
141              break;
142      }
143      return seed;
144  }
```

8.3.3.5 HierarchyIsEnabled()

This function checks to see if a hierarchy is enabled.

NOTE: The TPM_RH_NULL hierarchy is always enabled.

Return Value	Meaning
TRUE(1)	hierarchy is enabled
FALSE(0)	hierarchy is disabled

```
145  BOOL
146  HierarchyIsEnabled(
147      TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy
148      )
149  {
150      BOOL             enabled = FALSE;
151
152      switch(hierarchy)
153      {
154          case TPM_RH_PLATFORM:
155              enabled = g_phEnable;
156              break;
157          case TPM_RH_OWNER:
158              enabled = gc.shEnable;
159              break;
160          case TPM_RH_ENDORSEMENT:
161              enabled = gc.ehEnable;
162              break;
163          case TPM_RH_NULL:
164              enabled = TRUE;
165              break;
166          default:
167              enabled = FALSE;
168              break;
169      }
170      return enabled;
171  }
```

 Copyright © TCG 2006-2020

8.4 NvDynamic.c

8.4.1 Introduction

The NV memory is divided into two area: dynamic space for user defined NV indexes and evict objects, and reserved space for TPM persistent and state save data.

The entries in dynamic space are a linked list of entries. Each entry has, as its first field, a size. If the size field is zero, it marks the end of the list.

An Index allocation will contain an NV_INDEX structure. If the Index does not have the orderly attribute, the NV_INDEX is followed immediately by the NV data.

An evict object entry contains a handle followed by an OBJECT structure. This results in both the Index and Evict Object having an identifying handle as the first field following the size field.

When an Index has the orderly attribute, the data is kept in RAM. This RAM is saved to backing store in NV memory on any orderly shutdown. The entries in orderly memory are also a linked list using a size field as the first entry.

The attributes of an orderly index are maintained in RAM memory in order to reduce the number of NV writes needed for orderly data. When an orderly index is created, an entry is made in the dynamic NV memory space that holds the Index authorizations (*authPolicy* and *authValue*) and the size of the data. This entry is only modified if the *authValue* of the index is changed. The more volatile data of the index is kept in RAM. When an orderly Index is created or deleted, the RAM data is copied to NV backing store so that the image in the backing store matches the layout of RAM. In normal operation. The RAM data is also copied on any orderly shutdown. In normal operation, the only other reason for writing to the backing store for RAM is when a counter is first written (TPMA_NV_WRITTEN changes from CLEAR to SET) or when a counter "rolls over."

Static space contains items that are individually modifiable. The values are in the *gp* PERSISTEND_DATA structure in RAM and mapped to locations in NV.

8.4.2 Includes, Defines and Data Definitions

```
1    #define NV_C
2    #include "Tpm.h"
```

8.4.3 Local Functions

8.4.3.1 NvNext()

This function provides a method to traverse every data entry in NV dynamic area.

To begin with, parameter *iter* should be initialized to NV_REF_INIT indicating the first element. Every time this function is called, the value in *iter* would be adjusted pointing to the next element in traversal. If there is no next element, *iter* value would be 0. This function returns the address of the 'data entry' pointed by the *iter*. If there is no more element in the set, a 0 value is returned indicating the end of traversal.

```
3    static NV_REF
4    NvNext(
5        NV_REF          *iter,          // IN/OUT: the list iterator
6        TPM_HANDLE      *handle         // OUT: the handle of the next item.
7        )
8    {
9        NV_REF                  currentAddr;
10       NV_ENTRY_HEADER         header;
11   //
12       // If iterator is at the beginning of list
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
13      if(*iter == NV_REF_INIT)
14      {
15          // Initialize iterator
16          *iter = NV_USER_DYNAMIC;
17      }
18      // Step over the size field and point to the handle
19      currentAddr = *iter + sizeof(UINT32);
20
21      // read the header of the next entry
22      NvRead(&header, *iter, sizeof(NV_ENTRY_HEADER));
23
24      // if the size field is zero, then we have hit the end of the list
25      if(header.size == 0)
26          // leave the *iter pointing at the end of the list
27          return 0;
28      // advance the header by the size of the entry
29      *iter += header.size;
30
31      if(handle != NULL)
32          *handle = header.handle;
33      return currentAddr;
34  }
```

8.4.3.2 NvNextByType()

This function returns a reference to the next NV entry of the desired type

Return Value	Meaning
0	end of list
0	the next entry of the indicated type

```
35  static NV_REF
36  NvNextByType(
37      TPM_HANDLE      *handle,        // OUT: the handle of the found type
38      NV_REF          *iter,          // IN: the iterator
39      TPM_HT           type           // IN: the handle type to look for
40      )
41  {
42      NV_REF           addr;
43      TPM_HANDLE       nvHandle;
44  //
45      while((addr = NvNext(iter, &nvHandle)) != 0)
46      {
47          // addr: the address of the location containing the handle of the value
48          // iter: the next location.
49          if(HandleGetType(nvHandle) == type)
50              break;
51      }
52      if(handle != NULL)
53          *handle = nvHandle;
54      return addr;
55  }
```

8.4.3.3 NvNextIndex()

This function returns the reference to the next NV Index entry. A value of 0 indicates the end of the list.

 Copyright © TCG 2006-2020

Return Value	Meaning
0	end of list
0	the next reference

```
56  #define NvNextIndex(handle, iter)                     \
57      NvNextByType(handle, iter, TPM_HT_NV_INDEX)
```

8.4.3.4 NvNextEvict()

This function returns the offset in NV of the next evict object entry. A value of 0 indicates the end of the list.

```
58  #define NvNextEvict(handle, iter)                     \
59      NvNextByType(handle, iter, TPM_HT_PERSISTENT)
```

8.4.3.5 NvGetEnd()

Function to find the end of the NV dynamic data list

```
60  static NV_REF
61  NvGetEnd(
62      void
63      )
64  {
65      NV_REF          iter = NV_REF_INIT;
66      NV_REF          currentAddr;
67  //
68      // Scan until the next address is 0
69      while((currentAddr = NvNext(&iter, NULL)) != 0);
70      return iter;
71  }
```

8.4.3.6 NvGetFreeBytes

This function returns the number of free octets in NV space.

```
72  static UINT32
73  NvGetFreeBytes(
74      void
75      )
76  {
77      // This does not have an overflow issue because NvGetEnd() cannot return a value
78      // that is larger than s_evictNvEnd. This is because there is always a 'stop'
79      // word in the NV memory that terminates the search for the end before the
80      // value can go past s_evictNvEnd.
81      return s_evictNvEnd - NvGetEnd();
82  }
```

8.4.3.7 NvTestSpace()

This function will test if there is enough space to add a new entity.

 Copyright © TCG 2006-2020

Return Value	Meaning
TRUE(1)	space available
FALSE(0)	no enough space

```
83   static BOOL
84   NvTestSpace(
85       UINT32            size,          // IN: size of the entity to be added
86       BOOL              isIndex,       // IN: TRUE if the entity is an index
87       BOOL              isCounter      // IN: TRUE if the index is a counter
88       )
89   {
90       UINT32       remainBytes = NvGetFreeBytes();
91       UINT32       reserved = sizeof(UINT32)       // size of the forward pointer
92           + sizeof(NV_LIST_TERMINATOR);
93   //
94       // Do a compile time sanity check on the setting for NV_MEMORY_SIZE
95   #if NV_MEMORY_SIZE < 1024
96   #error "NV_MEMORY_SIZE probably isn't large enough"
97   #endif
98
99       // For NV Index, need to make sure that we do not allocate an Index if this
100      // would mean that the TPM cannot allocate the minimum number of evict
101      // objects.
102      if(isIndex)
103      {
104          // Get the number of persistent objects allocated
105          UINT32       persistentNum = NvCapGetPersistentNumber();
106
107          // If we have not allocated the requisite number of evict objects, then we
108          // need to reserve space for them.
109          // NOTE: some of this is not written as simply as it might seem because
110          // the values are all unsigned and subtracting needs to be done carefully
111          // so that an underflow doesn't cause problems.
112          if(persistentNum < MIN_EVICT_OBJECTS)
113              reserved += (MIN_EVICT_OBJECTS - persistentNum) * NV_EVICT_OBJECT_SIZE;
114      }
115      // If this is not an index or is not a counter, reserve space for the
116      // required number of counter indexes
117      if(!isIndex || !isCounter)
118      {
119          // Get the number of counters
120          UINT32       counterNum = NvCapGetCounterNumber();
121
122          // If the required number of counters have not been allocated, reserved
123          // space for the extra needed counters
124          if(counterNum < MIN_COUNTER_INDICES)
125              reserved += (MIN_COUNTER_INDICES - counterNum) * NV_INDEX_COUNTER_SIZE;
126      }
127      // Check that the requested allocation will fit after making sure that there
128      // will be no chance of overflow
129      return ((reserved < remainBytes)
130              && (size <= remainBytes)
131              && (size + reserved <= remainBytes));
132  }
```

8.4.3.8 NvWriteNvListEnd()

Function to write the list terminator.

```
133  NV_REF
134  NvWriteNvListEnd(
135      NV_REF            end
```

 Copyright © TCG 2006-2020

```
136     )
137 {
138     // Marker is initialized with zeros
139     BYTE        listEndMarker[sizeof(NV_LIST_TERMINATOR)] = {0};
140     UINT64      maxCount = NvReadMaxCount();
141 //
142     // This is a constant check that can be resolved at compile time.
143     cAssert(sizeof(UINT64) <= sizeof(NV_LIST_TERMINATOR) - sizeof(UINT32));
144
145     // Copy the maxCount value to the marker buffer
146     MemoryCopy(&listEndMarker[sizeof(UINT32)], &maxCount, sizeof(UINT64));
147     pAssert(end + sizeof(NV_LIST_TERMINATOR) <= s_evictNvEnd);
148
149     // Write it to memory
150     NvWrite(end, sizeof(NV_LIST_TERMINATOR), &listEndMarker);
151     return end + sizeof(NV_LIST_TERMINATOR);
152 }
```

8.4.3.9 NvAdd()

This function adds a new entity to NV.

This function requires that there is enough space to add a new entity (i.e., that NvTestSpace() has been called and the available space is at least as large as the required space).

The *totalSize* will be the size of *entity*. If a handle is added, this function will increase the size accordingly.

```
153 static TPM_RC
154 NvAdd(
155     UINT32           totalSize,     // IN: total size needed for this entity For
156                                     //     evict object, totalSize is the same as
157                                     //     bufferSize.  For NV Index, totalSize is
158                                     //     bufferSize plus index data size
159     UINT32           bufferSize,    // IN: size of initial buffer
160     TPM_HANDLE       handle,        // IN: optional handle
161     BYTE            *entity         // IN: initial buffer
162     )
163 {
164     NV_REF          newAddr;        // IN: where the new entity will start
165     NV_REF          nextAddr;
166 //
167     RETURN_IF_NV_IS_NOT_AVAILABLE;
168
169     // Get the end of data list
170     newAddr = NvGetEnd();
171
172     // Step over the forward pointer
173     nextAddr = newAddr + sizeof(UINT32);
174
175     // Optionally write the handle. For indexes, the handle is TPM_RH_UNASSIGNED
176     // so that the handle in the nvIndex is used instead of writing this value
177     if(handle != TPM_RH_UNASSIGNED)
178     {
179         NvWrite((UINT32)nextAddr, sizeof(TPM_HANDLE), &handle);
180         nextAddr += sizeof(TPM_HANDLE);
181     }
182     // Write entity data
183     NvWrite((UINT32)nextAddr, bufferSize, entity);
184
185     // Advance the pointer by the amount of the total
186     nextAddr += totalSize;
187
188     // Finish by writing the link value
189
190     // Write the next offset (relative addressing)
```

```
191         totalSize = nextAddr - newAddr;
192
193         // Write link value
194         NvWrite((UINT32)newAddr, sizeof(UINT32), &totalSize);
195
196         // Write the list terminator
197         NvWriteNvListEnd(nextAddr);
198
199         return TPM_RC_SUCCESS;
200     }
```

8.4.3.10 NvDelete()

This function is used to delete an NV Index or persistent object from NV memory.

```
201     static TPM_RC
202     NvDelete(
203         NV_REF            entityRef       // IN: reference to entity to be deleted
204         )
205     {
206         UINT32           entrySize;
207         // adjust entityAddr to back up and point to the forward pointer
208         NV_REF           entryRef = entityRef - sizeof(UINT32);
209         NV_REF           endRef = NvGetEnd();
210         NV_REF           nextAddr; // address of the next entry
211     //
212         RETURN_IF_NV_IS_NOT_AVAILABLE;
213
214         // Get the offset of the next entry. That is, back up and point to the size
215         // field of the entry
216         NvRead(&entrySize, entryRef, sizeof(UINT32));
217
218         // The next entry after the one being deleted is at a relative offset
219         // from the current entry
220         nextAddr = entryRef + entrySize;
221
222         // If this is not the last entry, move everything up
223         if(nextAddr < endRef)
224         {
225             pAssert(nextAddr > entryRef);
226             _plat__NvMemoryMove(nextAddr,
227                                 entryRef,
228                                 (endRef - nextAddr));
229         }
230         // The end of the used space is now moved up by the amount of space we just
231         // reclaimed
232         endRef -= entrySize;
233
234         // Write the end marker, and make the new end equal to the first byte after
235         // the just added end value. This will automatically update the NV value for
236         // maxCounter.
237         // NOTE: This is the call that sets flag to cause NV to be updated
238         endRef = NvWriteNvListEnd(endRef);
239
240         // Clear the reclaimed memory
241         _plat__NvMemoryClear(endRef, entrySize);
242
243         return TPM_RC_SUCCESS;
244     }
```

 Copyright © TCG 2006-2020

8.4.4 RAM-based NV Index Data Access Functions

8.4.4.1 Introduction

The data layout in ram buffer is {size of(NV_handle + attributes + data NV_handle, attributes, data} for each NV Index data stored in RAM.

NV storage associated with orderly data is updated when a NV Index is added but NOT when the data or attributes are changed. Orderly data is only updated to NV on an orderly shutdown (TPM2_Shutdown())

8.4.4.2 NvRamNext()

This function is used to iterate trough the list of Ram Index values. *iter needs to be initialized by calling

```
245 static NV_RAM_REF
246 NvRamNext(
247     NV_RAM_REF      *iter,          // IN/OUT: the list iterator
248     TPM_HANDLE      *handle         // OUT: the handle of the next item.
249     )
250 {
251     NV_RAM_REF           currentAddr;
252     NV_RAM_HEADER        header;
253 //
254     // If iterator is at the beginning of list
255     if(*iter == NV_RAM_REF_INIT)
256     {
257         // Initialize iterator
258         *iter = &s_indexOrderlyRam[0];
259     }
260     // if we are going to return what the iter is currently pointing to...
261     currentAddr = *iter;
262
263     // If iterator reaches the end of NV space, then don't advance and return
264     // that we are at the end of the list. The end of the list occurs when
265     // we don't have space for a size and a handle
266     if(currentAddr + sizeof(NV_RAM_HEADER) > RAM_ORDERLY_END)
267         return NULL;
268     // read the header of the next entry
269     MemoryCopy(&header, currentAddr, sizeof(NV_RAM_HEADER));
270
271     // if the size field is zero, then we have hit the end of the list
272     if(header.size == 0)
273         // leave the *iter pointing at the end of the list
274         return NULL;
275     // advance the header by the size of the entry
276     *iter = currentAddr + header.size;
277
278 //    pAssert(*iter <= RAM_ORDERLY_END);
279     if(handle != NULL)
280         *handle = header.handle;
281     return currentAddr;
282 }
```

8.4.4.3 NvRamGetEnd()

This routine performs the same function as NvGetEnd() but for the RAM data.

```
283 static NV_RAM_REF
284 NvRamGetEnd(
285     void
286     )
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
287 {
288     NV_RAM_REF          iter = NV_RAM_REF_INIT;
289     NV_RAM_REF          currentAddr;
290 //
291     // Scan until the next address is 0
292     while((currentAddr = NvRamNext(&iter, NULL)) != 0);
293     return iter;
294 }
```

8.4.4.4 NvRamTestSpaceIndex()

This function indicates if there is enough RAM space to add a data for a new NV Index.

Return Value	Meaning
TRUE(1)	space available
FALSE(0)	no enough space

```
295 static BOOL
296 NvRamTestSpaceIndex(
297     UINT32            size            // IN: size of the data to be added to RAM
298     )
299 {
300     UINT32           remaining = (UINT32)(RAM_ORDERLY_END - NvRamGetEnd());
301     UINT32           needed = sizeof(NV_RAM_HEADER) + size;
302 //
303     // NvRamGetEnd points to the next available byte.
304     return remaining >= needed;
305 }
```

8.4.4.5 NvRamGetIndex()

This function returns the offset of NV data in the RAM buffer

This function requires that NV Index is in RAM. That is, the index must be known to exist.

```
306 static NV_RAM_REF
307 NvRamGetIndex(
308     TPMI_RH_NV_INDEX      handle          // IN: NV handle
309     )
310 {
311     NV_RAM_REF          iter = NV_RAM_REF_INIT;
312     NV_RAM_REF          currentAddr;
313     TPM_HANDLE          foundHandle;
314 //
315     while((currentAddr = NvRamNext(&iter, &foundHandle)) != 0)
316     {
317         if(handle == foundHandle)
318             break;
319     }
320     return currentAddr;
321 }
```

8.4.4.6 NvUpdateIndexOrderlyData()

This function is used to cause an update of the orderly data to the NV backing store.

```
322 void
323 NvUpdateIndexOrderlyData(
324     void
325     )
```

```
326 {
327     // Write reserved RAM space to NV
328     NvWrite(NV_INDEX_RAM_DATA, sizeof(s_indexOrderlyRam), s_indexOrderlyRam);
329 }
```

8.4.4.7 NvAddRAM()

This function adds a new data area to RAM.

This function requires that enough free RAM space is available to add the new data.

This function should be called after the NV Index space has been updated and the index removed. This insures that NV is available so that checking for NV availability is not required during this function.

```
330 static void
331 NvAddRAM(
332     TPMS_NV_PUBLIC  *index          // IN: the index descriptor
333     )
334 {
335     NV_RAM_HEADER       header;
336     NV_RAM_REF          end = NvRamGetEnd();
337 //
338     header.size = sizeof(NV_RAM_HEADER) + index->dataSize;
339     header.handle = index->nvIndex;
340     MemoryCopy(&header.attributes, &index->attributes, sizeof(TPMA_NV));
341
342     pAssert(ORDERLY_RAM_ADDRESS_OK(end, header.size));
343
344     // Copy the header to the memory
345     MemoryCopy(end, &header, sizeof(NV_RAM_HEADER));
346
347     // Clear the data area (just in case)
348     MemorySet(end + sizeof(NV_RAM_HEADER), 0, index->dataSize);
349
350     // Step over this new entry
351     end += header.size;
352
353     // If the end marker will fit, add it
354     if(end + sizeof(UINT32) < RAM_ORDERLY_END)
355         MemorySet(end, 0, sizeof(UINT32));
356     // Write reserved RAM space to NV to reflect the newly added NV Index
357     SET_NV_UPDATE(UT_ORDERLY);
358
359     return;
360 }
```

8.4.4.8 NvDeleteRAM()

This function is used to delete a RAM-backed NV Index data area. The space used by the entry are overwritten by the contents of the Index data that comes after (the data is moved up to fill the hole left by removing this index. The reclaimed space is cleared to zeros. This function assumes the data of NV Index exists in RAM.

This function should be called after the NV Index space has been updated and the index removed. This insures that NV is available so that checking for NV availability is not required during this function.

```
361 static void
362 NvDeleteRAM(
363     TPMI_RH_NV_INDEX     handle          // IN: NV handle
364     )
365 {
366     NV_RAM_REF           nodeAddress;
367     NV_RAM_REF           nextNode;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
368     UINT32                size;
369     NV_RAM_REF            lastUsed = NvRamGetEnd();
370 //
371     nodeAddress = NvRamGetIndex(handle);
372
373     pAssert(nodeAddress != 0);
374
375     // Get node size
376     MemoryCopy(&size, nodeAddress, sizeof(size));
377
378     // Get the offset of next node
379     nextNode = nodeAddress + size;
380
381     // Copy the data
382     MemoryCopy(nodeAddress, nextNode, (int)(lastUsed - nextNode));
383
384     // Clear out the reclaimed space
385     MemorySet(lastUsed - size, 0, size);
386
387     // Write reserved RAM space to NV to reflect the newly delete NV Index
388     SET_NV_UPDATE(UT_ORDERLY);
389
390     return;
391 }
```

8.4.4.9 NvReadIndex()

This function is used to read the NV Index NV_INDEX. This is used so that the index information can be compressed and only this function would be needed to decompress it. Mostly, compression would only be able to save the space needed by the policy.

```
392 void
393 NvReadNvIndexInfo(
394     NV_REF           ref,            // IN: points to NV where index is located
395     NV_INDEX        *nvIndex         // OUT: place to receive index data
396     )
397 {
398     pAssert(nvIndex != NULL);
399     NvRead(nvIndex, ref, sizeof(NV_INDEX));
400     return;
401 }
```

8.4.4.10 NvReadObject()

This function is used to read a persistent object. This is used so that the object information can be compressed and only this function would be needed to uncompress it.

```
402 void
403 NvReadObject(
404     NV_REF           ref,            // IN: points to NV where index is located
405     OBJECT          *object          // OUT: place to receive the object data
406     )
407 {
408     NvRead(object, (ref + sizeof(TPM_HANDLE)), sizeof(OBJECT));
409     return;
410 }
```

8.4.4.11 NvFindEvict()

This function will return the NV offset of an evict object

Return Value	Meaning
0	evict object not found
0	offset of evict object

```
411 static NV_REF
412 NvFindEvict(
413     TPM_HANDLE       nvHandle,
414     OBJECT          *object
415     )
416 {
417     NV_REF          found = NvFindHandle(nvHandle);
418 //
419     // If we found the handle and the request included an object pointer, fill it in
420     if(found != 0 && object != NULL)
421         NvReadObject(found, object);
422     return found;
423 }
```

8.4.4.12 NvIndexIsDefined()

See if an index is already defined

```
424 BOOL
425 NvIndexIsDefined(
426     TPM_HANDLE       nvHandle       // IN: Index to look for
427     )
428 {
429     return (NvFindHandle(nvHandle) != 0);
430 }
```

8.4.4.13 NvConditionallyWrite()

Function to check if the data to be written has changed and write it if it has

Error Returns	Meaning
TPM_RC_NV_RATE	NV is unavailable because of rate limit
TPM_RC_NV_UNAVAILABLE	NV is inaccessible

```
431 static TPM_RC
432 NvConditionallyWrite(
433     NV_REF           entryAddr,     // IN: stating address
434     UINT32           size,          // IN: size of the data to write
435     void            *data           // IN: the data to write
436     )
437 {
438     // If the index data is actually changed, then a write to NV is required
439     if(_plat__NvIsDifferent(entryAddr, size, data))
440     {
441         // Write the data if NV is available
442         if(g_NvStatus == TPM_RC_SUCCESS)
443         {
444             NvWrite(entryAddr, size, data);
445         }
446         return g_NvStatus;
447     }
448     return TPM_RC_SUCCESS;
449 }
```

8.4.4.14 NvReadNvIndexAttributes()

This function returns the attributes of an NV Index.

```
450  static TPMA_NV
451  NvReadNvIndexAttributes(
452      NV_REF          locator         // IN: reference to an NV index
453      )
454  {
455      TPMA_NV                 attributes;
456  //
457      NvRead(&attributes,
458             locator + offsetof(NV_INDEX, publicArea.attributes),
459             sizeof(TPMA_NV));
460      return attributes;
461  }
```

8.4.4.15 NvReadRamIndexAttributes()

This function returns the attributes from the RAM header structure. This function is used to deal with the fact that the header structure is only byte aligned.

```
462  static TPMA_NV
463  NvReadRamIndexAttributes(
464      NV_RAM_REF      ref             // IN: pointer to a NV_RAM_HEADER
465      )
466  {
467      TPMA_NV         attributes;
468  //
469      MemoryCopy(&attributes, ref + offsetof(NV_RAM_HEADER, attributes),
470                 sizeof(TPMA_NV));
471      return attributes;
472  }
```

8.4.4.16 NvWriteNvIndexAttributes()

This function is used to write just the attributes of an index to NV.

Error Returns	Meaning
TPM_RC_NV_RATE	NV is rate limiting so retry
TPM_RC_NV_UNAVAILABLE	NV is not available

```
473  static TPM_RC
474  NvWriteNvIndexAttributes(
475      NV_REF          locator,        // IN: location of the index
476      TPMA_NV         attributes      // IN: attributes to write
477      )
478  {
479      return NvConditionallyWrite(
480          locator + offsetof(NV_INDEX, publicArea.attributes),
481          sizeof(TPMA_NV),
482          &attributes);
483  }
```

8.4.4.17 NvWriteRamIndexAttributes()

This function is used to write the index attributes into an unaligned structure

```
484 static void
485 NvWriteRamIndexAttributes(
486     NV_RAM_REF      ref,          // IN: address of the header
487     TPMA_NV         attributes    // IN: the attributes to write
488     )
489 {
490     MemoryCopy(ref + offsetof(NV_RAM_HEADER, attributes), &attributes,
491                sizeof(TPMA_NV));
492     return;
493 }
```

8.4.5 Externally Accessible Functions

8.4.5.1 NvIsPlatformPersistentHandle()

This function indicates if a handle references a persistent object in the range belonging to the platform.

Return Value	Meaning
TRUE(1)	handle references a platform persistent object and may reference an owner persistent object either
FALSE(0)	handle does not reference platform persistent object

```
494 BOOL
495 NvIsPlatformPersistentHandle(
496     TPM_HANDLE      handle        // IN: handle
497     )
498 {
499     return (handle >= PLATFORM_PERSISTENT && handle <= PERSISTENT_LAST);
500 }
```

8.4.5.2 NvIsOwnerPersistentHandle()

This function indicates if a handle references a persistent object in the range belonging to the owner.

Return Value	Meaning
TRUE(1)	handle is owner persistent handle
FALSE(0)	handle is not owner persistent handle and may not be a persistent handle at all

```
501 BOOL
502 NvIsOwnerPersistentHandle(
503     TPM_HANDLE      handle        // IN: handle
504     )
505 {
506     return (handle >= PERSISTENT_FIRST && handle < PLATFORM_PERSISTENT);
507 }
```

8.4.5.3 NvIndexIsAccessible()

This function validates that a handle references a defined NV Index and that the Index is currently accessible.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

Error Returns	Meaning
TPM_RC_HANDLE	the handle points to an undefined NV Index If *shEnable* is CLEAR, this would include an index created using *ownerAuth*. If *phEnableNV* is CLEAR, this would include and index created using *platformAuth*
TPM_RC_NV_READLOCKED	Index is present but locked for reading and command does not write to the index
TPM_RC_NV_WRITELOCKED	Index is present but locked for writing and command writes to the index

```
508 TPM_RC
509 NvIndexIsAccessible(
510     TPMI_RH_NV_INDEX     handle         // IN: handle
511     )
512 {
513     NV_INDEX             *nvIndex = NvGetIndexInfo(handle, NULL);
514 //
515     if(nvIndex == NULL)
516     // If index is not found, return TPM_RC_HANDLE
517         return TPM_RC_HANDLE;
518     if(gc.shEnable == FALSE || gc.phEnableNV == FALSE)
519     {
520         // if shEnable is CLEAR, an ownerCreate NV Index should not be
521         // indicated as present
522         if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, PLATFORMCREATE))
523         {
524             if(gc.shEnable == FALSE)
525                 return TPM_RC_HANDLE;
526         }
527         // if phEnableNV is CLEAR, a platform created Index should not
528         // be visible
529         else if(gc.phEnableNV == FALSE)
530             return TPM_RC_HANDLE;
531     }
532 #if 0 // Writelock test for debug
533     // If the Index is write locked and this is an NV Write operation...
534     if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITELOCKED)
535        &&  IsWriteOperation(commandIndex))
536     {
537         // then return a locked indication unless the command is TPM2_NV_WriteLock
538         if(GetCommandCode(commandIndex) != TPM_CC_NV_WriteLock)
539             return TPM_RC_NV_LOCKED;
540         return TPM_RC_SUCCESS;
541     }
542 #endif
543 #if 0   // Readlock Test for debug
544     // If the Index is read locked and this is an NV Read operation...
545     if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, READLOCKED)
546        && IsReadOperation(commandIndex))
547     {
548         // then return a locked indication unless the command is TPM2_NV_ReadLock
549         if(GetCommandCode(commandIndex) != TPM_CC_NV_ReadLock)
550             return TPM_RC_NV_LOCKED;
551     }
552 #endif
553     // NV Index is accessible
554     return TPM_RC_SUCCESS;
555 }
```

8.4.5.4 NvGetEvictObject()

This function is used to dereference an evict object handle and get a pointer to the object.

 Copyright © TCG 2006-2020

Error Returns	Meaning
TPM_RC_HANDLE	the handle does not point to an existing persistent object

```
556  TPM_RC
557  NvGetEvictObject(
558      TPM_HANDLE        handle,        // IN: handle
559      OBJECT           *object         // OUT: object data
560      )
561  {
562      NV_REF           entityAddr;         // offset points to the entity
563  //
564      // Find the address of evict object and copy to object
565      entityAddr = NvFindEvict(handle, object);
566
567      // whether there is an error or not, make sure that the evict
568      // status of the object is set so that the slot will get freed on exit
569      // Must do this after NvFindEvict loads the object
570      object->attributes.evict = SET;
571
572      // If handle is not found, return an error
573      if(entityAddr == 0)
574          return TPM_RC_HANDLE;
575      return TPM_RC_SUCCESS;
576  }
```

8.4.5.5 NvIndexCacheInit()

Function to initialize the Index cache

```
577  void
578  NvIndexCacheInit(
579      void
580      )
581  {
582      s_cachedNvRef = NV_REF_INIT;
583      s_cachedNvRamRef = NV_RAM_REF_INIT;
584      s_cachedNvIndex.publicArea.nvIndex = TPM_RH_UNASSIGNED;
585      return;
586  }
```

8.4.5.6 NvGetIndexData()

This function is used to access the data in an NV Index. The data is returned as a byte sequence.

This function requires that the NV Index be defined, and that the required data is within the data range. It also requires that TPMA_NV_WRITTEN of the Index is SET.

```
587  void
588  NvGetIndexData(
589      NV_INDEX             *nvIndex,       // IN: the in RAM index descriptor
590      NV_REF                locator,       // IN: where the data is located
591      UINT32                offset,        // IN: offset of NV data
592      UINT16                size,          // IN: number of octets of NV data to read
593      void                 *data           // OUT: data buffer
594      )
595  {
596      TPMA_NV              nvAttributes;
597  //
598      pAssert(nvIndex != NULL);
599
600      nvAttributes = nvIndex->publicArea.attributes;
```

```
601
602     pAssert(IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN));
603
604     if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, ORDERLY))
605     {
606         // Get data from RAM buffer
607         NV_RAM_REF          ramAddr = NvRamGetIndex(nvIndex->publicArea.nvIndex);
608         pAssert(ramAddr != 0 && (size <=
609                 ((NV_RAM_HEADER *)ramAddr)->size - sizeof(NV_RAM_HEADER) - offset));
610         MemoryCopy(data, ramAddr + sizeof(NV_RAM_HEADER) + offset, size);
611     }
612     else
613     {
614         // Validate that read falls within range of the index
615         pAssert(offset <= nvIndex->publicArea.dataSize
616                 &&  size <= (nvIndex->publicArea.dataSize - offset));
617         NvRead(data, locator + sizeof(NV_INDEX) + offset, size);
618     }
619     return;
620 }
```

8.4.5.7 NvHashIndexData()

This function adds Index data to a hash. It does this in parts to avoid large stack buffers.

```
621 void
622 NvHashIndexData(
623     HASH_STATE          *hashState,      // IN: Initialized hash state
624     NV_INDEX            *nvIndex,        // IN: Index
625     NV_REF               locator,        // IN: where the data is located
626     UINT32               offset,         // IN: starting offset
627     UINT16               size            // IN: amount to hash
628 )
629 {
630 #define BUFFER_SIZE     64
631     BYTE                 buffer[BUFFER_SIZE];
632     if (offset > nvIndex->publicArea.dataSize)
633         return;
634     // Make sure that we don't try to read off the end.
635     if ((offset + size) > nvIndex->publicArea.dataSize)
636         size = nvIndex->publicArea.dataSize - (UINT16)offset;
637 #if BUFFER_SIZE >= MAX_NV_INDEX_SIZE
638     NvGetIndexData(nvIndex, locator, offset, size, buffer);
639     CryptDigestUpdate(hashState, size, buffer);
640 #else
641     {
642         INT16               i;
643         UINT16              readSize;
644         //
645         for (i = size; i > 0; offset += readSize, i -= readSize)
646         {
647             readSize = (i < BUFFER_SIZE) ? i : BUFFER_SIZE;
648             NvGetIndexData(nvIndex, locator, offset, readSize, buffer);
649             CryptDigestUpdate(hashState, readSize, buffer);
650         }
651     }
652 #endif // BUFFER_SIZE >= MAX_NV_INDEX_SIZE
653 #undef  BUFFER_SIZE
654 }
```

8.4.5.8 NvGetUINT64Data()

Get data in integer format of a bit or counter NV Index.

This function requires that the NV Index is defined and that the NV Index previously has been written.

```
655 UINT64
656 NvGetUINT64Data(
657     NV_INDEX            *nvIndex,       // IN: the in RAM index descriptor
658     NV_REF               locator        // IN: where index exists in NV
659     )
660 {
661     UINT64                intVal;
662 //
663     // Read the value and convert it to internal format
664     NvGetIndexData(nvIndex, locator, 0, 8, &intVal);
665     return BYTE_ARRAY_TO_UINT64(((BYTE *)&intVal));
666 }
```

8.4.5.9 NvWriteIndexAttributes()

This function is used to write just the attributes of an index.

Error Returns	Meaning
TPM_RC_NV_RATE	NV is rate limiting so retry
TPM_RC_NV_UNAVAILABLE	NV is not available

```
667 TPM_RC
668 NvWriteIndexAttributes(
669     TPM_HANDLE       handle,
670     NV_REF           locator,       // IN: location of the index
671     TPMA_NV          attributes     // IN: attributes to write
672     )
673 {
674     TPM_RC              result;
675 //
676     if(IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY))
677     {
678         NV_RAM_REF      ram = NvRamGetIndex(handle);
679         NvWriteRamIndexAttributes(ram, attributes);
680         result = TPM_RC_SUCCESS;
681     }
682     else
683     {
684         result = NvWriteNvIndexAttributes(locator, attributes);
685     }
686     return result;
687 }
```

8.4.5.10 NvWriteIndexAuth()

This function is used to write the *authValue* of an index. It is used by TPM2_NV_ChangeAuth()

Error Returns	Meaning
TPM_RC_NV_RATE	NV is rate limiting so retry
TPM_RC_NV_UNAVAILABLE	NV is not available

```
688 TPM_RC
689 NvWriteIndexAuth(
690     NV_REF           locator,       // IN: location of the index
691     TPM2B_AUTH      *authValue      // IN: the authValue to write
692     )
```

 Copyright © TCG 2006-2020

```
693  {
694      TPM_RC              result;
695  //
696      // If the locator is pointing to the cached index value...
697      if(locator == s_cachedNvRef)
698      {
699          // copy the authValue to the cached index so it will be there if we
700          // look for it. This is a safety thing.
701          MemoryCopy2B(&s_cachedNvIndex.authValue.b, &authValue->b,
702                       sizeof(s_cachedNvIndex.authValue.t.buffer));
703      }
704      result = NvConditionallyWrite(
705          locator + offsetof(NV_INDEX, authValue),
706          sizeof(UINT16) + authValue->t.size,
707          authValue);
708      return result;
709  }
```

8.4.5.11 NvGetIndexInfo()

This function loads the *nvIndex* Info into the NV cache and returns a pointer to the NV_INDEX. If the returned value is zero, the index was not found. The *locator* parameter, if not NULL, will be set to the offset in NV of the Index (the location of the handle of the Index).

This function will set the index cache. If the index is orderly, the attributes from RAM are substituted for the attributes in the cached index

```
710  NV_INDEX *
711  NvGetIndexInfo(
712      TPM_HANDLE       nvHandle,      // IN: the index handle
713      NV_REF          *locator        // OUT: location of the index
714      )
715  {
716      if(s_cachedNvIndex.publicArea.nvIndex != nvHandle)
717      {
718          s_cachedNvIndex.publicArea.nvIndex = TPM_RH_UNASSIGNED;
719          s_cachedNvRamRef = 0;
720          s_cachedNvRef = NvFindHandle(nvHandle);
721          if(s_cachedNvRef == 0)
722              return NULL;
723          NvReadNvIndexInfo(s_cachedNvRef, &s_cachedNvIndex);
724          if(IS_ATTRIBUTE(s_cachedNvIndex.publicArea.attributes, TPMA_NV, ORDERLY))
725          {
726              s_cachedNvRamRef = NvRamGetIndex(nvHandle);
727              s_cachedNvIndex.publicArea.attributes =
728                  NvReadRamIndexAttributes(s_cachedNvRamRef);
729          }
730      }
731      if(locator != NULL)
732          *locator = s_cachedNvRef;
733      return &s_cachedNvIndex;
734  }
```

8.4.5.12 NvWriteIndexData()

This function is used to write NV index data. It is intended to be used to update the data associated with the default index.

This function requires that the NV Index is defined, and the data is within the defined data range for the index.

Index data is only written due to a command that modifies the data in a single index. There is no case where changes are made to multiple indexes data at the same time. Multiple attributes may be change

but not multiple index data. This is important because we will normally be handling the index for which we have the cached pointer values.

Error Returns	Meaning
TPM_RC_NV_RATE	NV is rate limiting so retry
TPM_RC_NV_UNAVAILABLE	NV is not available

```
735 TPM_RC
736 NvWriteIndexData(
737     NV_INDEX        *nvIndex,       // IN: the description of the index
738     UINT32           offset,        // IN: offset of NV data
739     UINT32           size,          // IN: size of NV data
740     void            *data           // IN: data buffer
741     )
742 {
743     TPM_RC                result = TPM_RC_SUCCESS;
744 //
745     pAssert(nvIndex != NULL);
746     // Make sure that this is dealing with the 'default' index.
747     // Note: it is tempting to change the calling sequence so that the 'default' is
748     // presumed.
749     pAssert(nvIndex->publicArea.nvIndex == s_cachedNvIndex.publicArea.nvIndex);
750
751     // Validate that write falls within range of the index
752     pAssert(offset <= nvIndex->publicArea.dataSize
753             &&  size <= (nvIndex->publicArea.dataSize - offset));
754
755     // Update TPMA_NV_WRITTEN bit if necessary
756     if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
757     {
758         // Update the in memory version of the attributes
759         SET_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN);
760
761         // If this is not orderly, then update the NV version of
762         // the attributes
763         if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
764         {
765             result = NvWriteNvIndexAttributes(s_cachedNvRef,
766                                               nvIndex->publicArea.attributes);
767             if(result != TPM_RC_SUCCESS)
768                 return result;
769             // If this is a partial write of an ordinary index, clear the whole
770             // index.
771             if(IsNvOrdinaryIndex(nvIndex->publicArea.attributes)
772                && (nvIndex->publicArea.dataSize > size))
773                 _plat__NvMemoryClear(s_cachedNvRef + sizeof(NV_INDEX),
774                                      nvIndex->publicArea.dataSize);
775         }
776         else
777         {
778             // This is orderly so update the RAM version
779             MemoryCopy(s_cachedNvRamRef + offsetof(NV_RAM_HEADER, attributes),
780                        &nvIndex->publicArea.attributes, sizeof(TPMA_NV));
781             // If setting WRITTEN for an orderly counter, make sure that the
782             // state saved version of the counter is saved
783             if(IsNvCounterIndex(nvIndex->publicArea.attributes))
784                 SET_NV_UPDATE(UT_ORDERLY);
785             // If setting the written attribute on an ordinary index, make sure that
786             // the data is all cleared out in case there is a partial write. This
787             // is only necessary for ordinary indexes because all of the other types
788             // are always written in total.
789             else if(IsNvOrdinaryIndex(nvIndex->publicArea.attributes))
790                 MemorySet(s_cachedNvRamRef + sizeof(NV_RAM_HEADER),
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
791                             0, nvIndex->publicArea.dataSize);
792             }
793         }
794         // If this is orderly data, write it to RAM
795         if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
796         {
797             // Note: if this is the first write to a counter, the code above will queue
798             // the write to NV of the RAM data in order to update TPMA_NV_WRITTEN. In
799             // process of doing that write, it will also write the initial counter value
800
801             // Update RAM
802             MemoryCopy(s_cachedNvRamRef + sizeof(NV_RAM_HEADER) + offset, data, size);
803
804             // And indicate that the TPM is no longer orderly
805             g_clearOrderly = TRUE;
806         }
807         else
808         {
809             // Offset into the index to the first byte of the data to be written to NV
810             result = NvConditionallyWrite(s_cachedNvRef + sizeof(NV_INDEX) + offset,
811                                           size, data);
812         }
813         return result;
814     }
```

8.4.5.13 NvWriteUINT64Data()

This function to write back a UINT64 value. The various UINT64 values (bits, counters, and PINs) are kept in canonical format but manipulate in native format. This takes a native format value converts it and saves it back as in canonical format.

This function will return the value from NV or RAM depending on the type of the index (orderly or not)

```
815     TPM_RC
816     NvWriteUINT64Data(
817         NV_INDEX        *nvIndex,       // IN: the description of the index
818         UINT64           intValue       // IN: the value to write
819         )
820     {
821         BYTE            bytes[8];
822         UINT64_TO_BYTE_ARRAY(intValue, bytes);
823     //
824         return NvWriteIndexData(nvIndex, 0, 8, &bytes);
825     }
```

8.4.5.14 NvGetIndexName()

This function computes the Name of an index The *name* buffer receives the bytes of the Name and the return value is the number of octets in the Name.

This function requires that the NV Index is defined.

```
826     TPM2B_NAME *
827     NvGetIndexName(
828         NV_INDEX        *nvIndex,       // IN: the index over which the name is to be
829                                         //     computed
830         TPM2B_NAME      *name           // OUT: name of the index
831         )
832     {
833         UINT16              dataSize, digestSize;
834         BYTE                marshalBuffer[sizeof(TPMS_NV_PUBLIC)];
835         BYTE               *buffer;
836         HASH_STATE          hashState;
```

```
837 //
838     // Marshal public area
839     buffer = marshalBuffer;
840     dataSize = TPMS_NV_PUBLIC_Marshal(&nvIndex->publicArea, &buffer, NULL);
841
842     // hash public area
843     digestSize = CryptHashStart(&hashState, nvIndex->publicArea.nameAlg);
844     CryptDigestUpdate(&hashState, dataSize, marshalBuffer);
845
846     // Complete digest leaving room for the nameAlg
847     CryptHashEnd(&hashState, digestSize, &name->b.buffer[2]);
848
849     // Include the nameAlg
850     UINT16_TO_BYTE_ARRAY(nvIndex->publicArea.nameAlg, name->b.buffer);
851     name->t.size = digestSize + 2;
852     return name;
853 }
```

8.4.5.15 NvGetNameByIndexHandle()

This function is used to compute the Name of an NV Index referenced by handle.

The *name* buffer receives the bytes of the Name and the return value is the number of octets in the Name.

This function requires that the NV Index is defined.

```
854 TPM2B_NAME *
855 NvGetNameByIndexHandle(
856     TPMI_RH_NV_INDEX     handle,         // IN: handle of the index
857     TPM2B_NAME          *name            // OUT: name of the index
858     )
859 {
860     NV_INDEX             *nvIndex = NvGetIndexInfo(handle, NULL);
861 //
862     return NvGetIndexName(nvIndex, name);
863 }
```

8.4.5.16 NvDefineIndex()

This function is used to assign NV memory to an NV Index.

Error Returns	Meaning
TPM_RC_NV_SPACE	insufficient NV space

```
864 TPM_RC
865 NvDefineIndex(
866     TPMS_NV_PUBLIC  *publicArea,    // IN: A template for an area to create.
867     TPM2B_AUTH      *authValue      // IN: The initial authorization value
868     )
869 {
870     // The buffer to be written to NV memory
871     NV_INDEX        nvIndex;            // the index data
872     UINT16          entrySize;          // size of entry
873     TPM_RC          result;
874 //
875     entrySize = sizeof(NV_INDEX);
876
877     // only allocate data space for indexes that are going to be written to NV.
878     // Orderly indexes don't need space.
879     if(!IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY))
880         entrySize += publicArea->dataSize;
```

```
881     // Check if we have enough space to create the NV Index
882     // In this implementation, the only resource limitation is the available NV
883     // space (and possibly RAM space.)  Other implementation may have other
884     // limitation on counter or on NV slots
885     if(!NvTestSpace(entrySize, TRUE, IsNvCounterIndex(publicArea->attributes)))
886         return TPM_RC_NV_SPACE;
887
888     // if the index to be defined is RAM backed, check RAM space availability
889     // as well
890     if(IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY)
891        &&  !NvRamTestSpaceIndex(publicArea->dataSize))
892         return TPM_RC_NV_SPACE;
893     // Copy input value to nvBuffer
894     nvIndex.publicArea = *publicArea;
895
896     // Copy the authValue
897     nvIndex.authValue = *authValue;
898
899     // Add index to NV memory
900     result = NvAdd(entrySize, sizeof(NV_INDEX), TPM_RH_UNASSIGNED,
901                    (BYTE *)&nvIndex);
902     if(result == TPM_RC_SUCCESS)
903     {
904     // If the data of NV Index is RAM backed, add the data area in RAM as well
905         if(IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY))
906             NvAddRAM(publicArea);
907     }
908     return result;
909 }
```

8.4.5.17 NvAddEvictObject()

This function is used to assign NV memory to a persistent object.

Error Returns	Meaning
TPM_RC_NV_HANDLE	the requested handle is already in use
TPM_RC_NV_SPACE	insufficient NV space

```
910 TPM_RC
911 NvAddEvictObject(
912     TPMI_DH_OBJECT    evictHandle,    // IN: new evict handle
913     OBJECT           *object          // IN: object to be added
914     )
915 {
916     TPM_HANDLE        temp = object->evictHandle;
917     TPM_RC            result;
918 //
919     // Check if we have enough space to add the evict object
920     // An evict object needs 8 bytes in index table + sizeof OBJECT
921     // In this implementation, the only resource limitation is the available NV
922     // space.  Other implementation may have other limitation on evict object
923     // handle space
924     if(!NvTestSpace(sizeof(OBJECT) + sizeof(TPM_HANDLE), FALSE, FALSE))
925         return TPM_RC_NV_SPACE;
926
927     // Set evict attribute and handle
928     object->attributes.evict = SET;
929     object->evictHandle = evictHandle;
930
931     // Now put this in NV
932     result = NvAdd(sizeof(OBJECT), sizeof(OBJECT), evictHandle, (BYTE *)object);
933
```

```
934         // Put things back the way they were
935         object->attributes.evict = CLEAR;
936         object->evictHandle = temp;
937
938         return result;
939     }
```

8.4.5.18 NvDeleteIndex()

This function is used to delete an NV Index.

Error Returns	Meaning
TPM_RC_NV_UNAVAILABLE	NV is not accessible
TPM_RC_NV_RATE	NV is rate limiting

```
940     TPM_RC
941     NvDeleteIndex(
942         NV_INDEX        *nvIndex,       // IN: an in RAM index descriptor
943         NV_REF           entityAddr     // IN: location in NV
944         )
945     {
946         TPM_RC           result;
947     //
948         if(nvIndex != NULL)
949         {
950             // Whenever a counter is deleted, make sure that the MaxCounter value is
951             // updated to reflect the value
952             if(IsNvCounterIndex(nvIndex->publicArea.attributes)
953                && IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
954                 NvUpdateMaxCount(NvGetUINT64Data(nvIndex, entityAddr));
955             result = NvDelete(entityAddr);
956             if(result != TPM_RC_SUCCESS)
957                 return result;
958             // If the NV Index is RAM backed, delete the RAM data as well
959             if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
960                 NvDeleteRAM(nvIndex->publicArea.nvIndex);
961             NvIndexCacheInit();
962         }
963         return TPM_RC_SUCCESS;
964     }
```

8.4.5.19 NvDeleteEvict()

This function will delete a NV evict object. Will return success if object deleted or if it does not exist

```
965     TPM_RC
966     NvDeleteEvict(
967         TPM_HANDLE       handle         // IN: handle of entity to be deleted
968         )
969     {
970         NV_REF      entityAddr = NvFindEvict(handle, NULL);     // pointer to entity
971         TPM_RC      result = TPM_RC_SUCCESS;
972     //
973         if(entityAddr != 0)
974             result = NvDelete(entityAddr);
975         return result;
976     }
```

8.4.5.20 NvFlushHierarchy()

This function will delete persistent objects belonging to the indicated hierarchy. If the storage hierarchy is selected, the function will also delete any NV Index defined using *ownerAuth*.

Error Returns	Meaning
TPM_RC_NV_RATE	NV is unavailable because of rate limit
TPM_RC_NV_UNAVAILABLE	NV is inaccessible

```
977   TPM_RC
978   NvFlushHierarchy(
979       TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy to be flushed.
980       )
981   {
982       NV_REF           iter = NV_REF_INIT;
983       NV_REF           currentAddr;
984       TPM_HANDLE       entityHandle;
985       TPM_RC           result = TPM_RC_SUCCESS;
986   //
987       while((currentAddr = NvNext(&iter, &entityHandle)) != 0)
988       {
989           if(HandleGetType(entityHandle) == TPM_HT_NV_INDEX)
990           {
991               NV_INDEX        nvIndex;
992   //
993               // If flush endorsement or platform hierarchy, no NV Index would be
994               // flushed
995               if(hierarchy == TPM_RH_ENDORSEMENT || hierarchy == TPM_RH_PLATFORM)
996                   continue;
997               // Get the index information
998               NvReadNvIndexInfo(currentAddr, &nvIndex);
999
1000              // For storage hierarchy, flush OwnerCreated index
1001              if(!IS_ATTRIBUTE(nvIndex.publicArea.attributes, TPMA_NV,
1002                               PLATFORMCREATE))
1003              {
1004                  // Delete the index (including RAM for orderly)
1005                  result = NvDeleteIndex(&nvIndex, currentAddr);
1006                  if(result != TPM_RC_SUCCESS)
1007                      break;
1008                  // Re-iterate from beginning after a delete
1009                  iter = NV_REF_INIT;
1010              }
1011          }
1012          else if(HandleGetType(entityHandle) == TPM_HT_PERSISTENT)
1013          {
1014              OBJECT_ATTRIBUTES           attributes;
1015  //
1016              NvRead(&attributes,
1017                     (UINT32)(currentAddr
1018                              + sizeof(TPM_HANDLE)
1019                              + offsetof(OBJECT, attributes)),
1020                     sizeof(OBJECT_ATTRIBUTES));
1021              // If the evict object belongs to the hierarchy to be flushed...
1022              if((hierarchy == TPM_RH_PLATFORM && attributes.ppsHierarchy == SET)
1023                 || (hierarchy == TPM_RH_OWNER && attributes.spsHierarchy == SET)
1024                 || (hierarchy == TPM_RH_ENDORSEMENT
1025                     && attributes.epsHierarchy == SET))
1026              {
1027                  // ...then delete the evict object
1028                  result = NvDelete(currentAddr);
1029                  if(result != TPM_RC_SUCCESS)
1030                      break;
```

 Copyright © TCG 2006-2020

```
1031                // Re-iterate from beginning after a delete
1032                iter = NV_REF_INIT;
1033            }
1034        }
1035        else
1036        {
1037            FAIL(FATAL_ERROR_INTERNAL);
1038        }
1039    }
1040    return result;
1041 }
```

8.4.5.21 NvSetGlobalLock()

This function is used to SET the TPMA_NV_WRITELOCKED attribute for all NV indexes that have TPMA_NV_GLOBALLOCK SET. This function is use by TPM2_NV_GlobalWriteLock().

Error Returns	Meaning
TPM_RC_NV_RATE	NV is unavailable because of rate limit
TPM_RC_NV_UNAVAILABLE	NV is inaccessible

```
1042 TPM_RC
1043 NvSetGlobalLock(
1044     void
1045     )
1046 {
1047     NV_REF              iter = NV_REF_INIT;
1048     NV_RAM_REF          ramIter = NV_RAM_REF_INIT;
1049     NV_REF              currentAddr;
1050     NV_RAM_REF          currentRamAddr;
1051     TPM_RC              result = TPM_RC_SUCCESS;
1052 //
1053     // Check all normal indexes
1054     while((currentAddr = NvNextIndex(NULL, &iter)) != 0)
1055     {
1056         TPMA_NV         attributes = NvReadNvIndexAttributes(currentAddr);
1057 //
1058         // See if it should be locked
1059         if(!IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY)
1060            &&  IS_ATTRIBUTE(attributes, TPMA_NV, GLOBALLOCK))
1061         {
1062             SET_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
1063             result = NvWriteNvIndexAttributes(currentAddr, attributes);
1064             if(result != TPM_RC_SUCCESS)
1065                 return result;
1066         }
1067     }
1068     // Now search all the orderly attributes
1069     while((currentRamAddr = NvRamNext(&ramIter, NULL)) != 0)
1070     {
1071         // See if it should be locked
1072         TPMA_NV         attributes = NvReadRamIndexAttributes(currentRamAddr);
1073         if(IS_ATTRIBUTE(attributes, TPMA_NV, GLOBALLOCK))
1074         {
1075             SET_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
1076             NvWriteRamIndexAttributes(currentRamAddr, attributes);
1077         }
1078     }
1079     return result;
1080 }
```

8.4.5.22 InsertSort()

Sort a handle into handle list in ascending order. The total handle number in the list should not exceed MAX_CAP_HANDLES

```
1081 static void
1082 InsertSort(
1083     TPML_HANDLE     *handleList,    // IN/OUT: sorted handle list
1084     UINT32           count,         // IN: maximum count in the handle list
1085     TPM_HANDLE       entityHandle   // IN: handle to be inserted
1086     )
1087 {
1088     UINT32          i, j;
1089     UINT32          originalCount;
1090 //
1091     // For a corner case that the maximum count is 0, do nothing
1092     if(count == 0)
1093         return;
1094     // For empty list, add the handle at the beginning and return
1095     if(handleList->count == 0)
1096     {
1097         handleList->handle[0] = entityHandle;
1098         handleList->count++;
1099         return;
1100     }
1101     // Check if the maximum of the list has been reached
1102     originalCount = handleList->count;
1103     if(originalCount < count)
1104         handleList->count++;
1105     // Insert the handle to the list
1106     for(i = 0; i < originalCount; i++)
1107     {
1108         if(handleList->handle[i] > entityHandle)
1109         {
1110             for(j = handleList->count - 1; j > i; j--)
1111             {
1112                 handleList->handle[j] = handleList->handle[j - 1];
1113             }
1114             break;
1115         }
1116     }
1117     // If a slot was found, insert the handle in this position
1118     if(i < originalCount || handleList->count > originalCount)
1119         handleList->handle[i] = entityHandle;
1120     return;
1121 }
```

8.4.5.23 NvCapGetPersistent()

This function is used to get a list of handles of the persistent objects, starting at *handle*.

Handle must be in valid persistent object handle range, but does not have to reference an existing persistent object.

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
1122 TPMI_YES_NO
1123 NvCapGetPersistent(
1124     TPMI_DH_OBJECT   handle,        // IN: start handle
1125     UINT32           count,         // IN: maximum number of returned handles
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
1126        TPML_HANDLE      *handleList      // OUT: list of handle
1127        )
1128    {
1129        TPMI_YES_NO                 more = NO;
1130        NV_REF                      iter = NV_REF_INIT;
1131        NV_REF                      currentAddr;
1132        TPM_HANDLE                  entityHandle;
1133    //
1134        pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
1135
1136        // Initialize output handle list
1137        handleList->count = 0;
1138
1139        // The maximum count of handles we may return is MAX_CAP_HANDLES
1140        if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1141
1142        while((currentAddr = NvNextEvict(&entityHandle, &iter)) != 0)
1143        {
1144            // Ignore persistent handles that have values less than the input handle
1145            if(entityHandle < handle)
1146                continue;
1147            // if the handles in the list have reached the requested count, and there
1148            // are still handles need to be inserted, indicate that there are more.
1149            if(handleList->count == count)
1150                more = YES;
1151            // A handle with a value larger than start handle is a candidate
1152            // for return. Insert sort it to the return list.  Insert sort algorithm
1153            // is chosen here for simplicity based on the assumption that the total
1154            // number of NV indexes is small.  For an implementation that may allow
1155            // large number of NV indexes, a more efficient sorting algorithm may be
1156            // used here.
1157            InsertSort(handleList, count, entityHandle);
1158        }
1159        return more;
1160    }
```

8.4.5.24 NvCapGetIndex()

This function returns a list of handles of NV indexes, starting from *handle*. *Handle* must be in the range of NV indexes, but does not have to reference an existing NV Index.

Return Value	Meaning
YES	if there are more handles to report
NO	all the available handles has been reported

```
1161    TPMI_YES_NO
1162    NvCapGetIndex(
1163        TPMI_DH_OBJECT    handle,         // IN: start handle
1164        UINT32            count,          // IN: max number of returned handles
1165        TPML_HANDLE      *handleList      // OUT: list of handle
1166        )
1167    {
1168        TPMI_YES_NO                 more = NO;
1169        NV_REF                      iter = NV_REF_INIT;
1170        NV_REF                      currentAddr;
1171        TPM_HANDLE                  nvHandle;
1172    //
1173        pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
1174
1175        // Initialize output handle list
1176        handleList->count = 0;
1177
```

 Copyright © TCG 2006-2020

```
1178        // The maximum count of handles we may return is MAX_CAP_HANDLES
1179        if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1180
1181        while((currentAddr = NvNextIndex(&nvHandle, &iter)) != 0)
1182        {
1183            // Ignore index handles that have values less than the 'handle'
1184            if(nvHandle < handle)
1185                continue;
1186            // if the count of handles in the list has reached the requested count,
1187            // and there are still handles to report, set more.
1188            if(handleList->count == count)
1189                more = YES;
1190            // A handle with a value larger than start handle is a candidate
1191            // for return. Insert sort it to the return list.  Insert sort algorithm
1192            // is chosen here for simplicity based on the assumption that the total
1193            // number of NV indexes is small.  For an implementation that may allow
1194            // large number of NV indexes, a more efficient sorting algorithm may be
1195            // used here.
1196            InsertSort(handleList, count, nvHandle);
1197        }
1198        return more;
1199    }
```

8.4.5.25 NvCapGetIndexNumber()

This function returns the count of NV Indexes currently defined.

```
1200    UINT32
1201    NvCapGetIndexNumber(
1202        void
1203        )
1204    {
1205        UINT32           num = 0;
1206        NV_REF           iter = NV_REF_INIT;
1207    //
1208        while(NvNextIndex(NULL, &iter) != 0)
1209            num++;
1210        return num;
1211    }
```

8.4.5.26 NvCapGetPersistentNumber()

Function returns the count of persistent objects currently in NV memory.

```
1212    UINT32
1213    NvCapGetPersistentNumber(
1214        void
1215        )
1216    {
1217        UINT32          num = 0;
1218        NV_REF         iter = NV_REF_INIT;
1219        TPM_HANDLE      handle;
1220    //
1221        while(NvNextEvict(&handle, &iter) != 0)
1222            num++;
1223        return num;
1224    }
```

8.4.5.27 NvCapGetPersistentAvail()

This function returns an estimate of the number of additional persistent objects that could be loaded into NV memory.

```
1225  UINT32
1226  NvCapGetPersistentAvail(
1227      void
1228      )
1229  {
1230      UINT32          availNVSpace;
1231      UINT32          counterNum = NvCapGetCounterNumber();
1232      UINT32          reserved = sizeof(NV_LIST_TERMINATOR);
1233  //
1234      // Get the available space in NV storage
1235      availNVSpace = NvGetFreeBytes();
1236
1237      if(counterNum < MIN_COUNTER_INDICES)
1238      {
1239          // Some space has to be reserved for counter objects.
1240          reserved += (MIN_COUNTER_INDICES - counterNum) * NV_INDEX_COUNTER_SIZE;
1241          if(reserved > availNVSpace)
1242              availNVSpace = 0;
1243          else
1244              availNVSpace -= reserved;
1245      }
1246      return availNVSpace / NV_EVICT_OBJECT_SIZE;
1247  }
```

8.4.5.28 NvCapGetCounterNumber()

Get the number of defined NV Indexes that are counter indexes.

```
1248  UINT32
1249  NvCapGetCounterNumber(
1250      void
1251      )
1252  {
1253      NV_REF           iter = NV_REF_INIT;
1254      NV_REF           currentAddr;
1255      UINT32           num = 0;
1256  //
1257      while((currentAddr = NvNextIndex(NULL, &iter)) != 0)
1258      {
1259          TPMA_NV             attributes = NvReadNvIndexAttributes(currentAddr);
1260          if(IsNvCounterIndex(attributes))
1261              num++;
1262      }
1263      return num;
1264  }
```

8.4.5.29 NvSetStartupAttributes()

Local function to set the attributes of an Index at TPM Reset and TPM Restart.

```
1265  static TPMA_NV
1266  NvSetStartupAttributes(
1267      TPMA_NV         attributes,         // IN: attributes to change
1268      STARTUP_TYPE     type               // IN: start up type
1269      )
1270  {
1271      // Clear read lock
```

```
1272        CLEAR_ATTRIBUTE(attributes, TPMA_NV, READLOCKED);
1273
1274        // Will change a non counter index to the unwritten state if:
1275        // a) TPMA_NV_CLEAR_STCLEAR is SET
1276        // b) orderly and TPM Reset
1277        if(!IsNvCounterIndex(attributes))
1278        {
1279            if(IS_ATTRIBUTE(attributes, TPMA_NV, CLEAR_STCLEAR)
1280               || (IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY)
1281                   && (type == SU_RESET)))
1282                CLEAR_ATTRIBUTE(attributes, TPMA_NV, WRITTEN);
1283        }
1284        // Unlock any index that is not written or that does not have
1285        // TPMA_NV_WRITEDEFINE SET.
1286        if(!IS_ATTRIBUTE(attributes, TPMA_NV, WRITTEN)
1287           || !IS_ATTRIBUTE(attributes, TPMA_NV, WRITEDEFINE))
1288            CLEAR_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
1289        return attributes;
1290    }
```

8.4.5.30 NvEntityStartup()

This function is called at TPM_Startup(). If the startup completes a TPM Resume cycle, no action is taken. If the startup is a TPM Reset or a TPM Restart, then this function will:

a) clear read/write lock;

b) reset NV Index data that has TPMA_NV_CLEAR_STCLEAR SET; and

c) set the lower bits in orderly counters to 1 for a non-orderly startup

It is a prerequisite that NV be available for writing before this function is called.

```
1291    BOOL
1292    NvEntityStartup(
1293        STARTUP_TYPE     type           // IN: start up type
1294        )
1295    {
1296        NV_REF                iter = NV_REF_INIT;
1297        NV_RAM_REF            ramIter = NV_RAM_REF_INIT;
1298        NV_REF                currentAddr;        // offset points to the current entity
1299        NV_RAM_REF            currentRamAddr;
1300        TPM_HANDLE            nvHandle;
1301        TPMA_NV               attributes;
1302    //
1303        // Restore RAM index data
1304        NvRead(s_indexOrderlyRam, NV_INDEX_RAM_DATA, sizeof(s_indexOrderlyRam));
1305
1306        // Initialize the max NV counter value
1307        NvSetMaxCount(NvGetMaxCount());
1308
1309        // If recovering from state save, do nothing else
1310        if(type == SU_RESUME)
1311            return TRUE;
1312        // Iterate all the NV Index to clear the locks
1313        while((currentAddr = NvNextIndex(&nvHandle, &iter)) != 0)
1314        {
1315            attributes = NvReadNvIndexAttributes(currentAddr);
1316
1317            // If this is an orderly index, defer processing until loop below
1318            if(IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY))
1319                continue;
1320            // Set the attributes appropriate for this startup type
1321            attributes = NvSetStartupAttributes(attributes, type);
1322            NvWriteNvIndexAttributes(currentAddr, attributes);
```

```
1323        }
1324        // Iterate all the orderly indexes to clear the locks and initialize counters
1325        while((currentRamAddr = NvRamNext(&ramIter, NULL)) != 0)
1326        {
1327            attributes = NvReadRamIndexAttributes(currentRamAddr);
1328
1329            attributes = NvSetStartupAttributes(attributes, type);
1330
1331            // update attributes in RAM
1332            NvWriteRamIndexAttributes(currentRamAddr, attributes);
1333
1334            // Set the lower bits in an orderly counter to 1 for a non-orderly startup
1335            if(IsNvCounterIndex(attributes)
1336               && (g_prevOrderlyState == SU_NONE_VALUE))
1337            {
1338                UINT64      counter;
1339    //
1340                // Read the counter value last saved to NV.
1341                counter = BYTE_ARRAY_TO_UINT64(currentRamAddr + sizeof(NV_RAM_HEADER));
1342
1343                // Set the lower bits of counter to 1's
1344                counter |= MAX_ORDERLY_COUNT;
1345
1346                // Write back to RAM
1347                // NOTE: Do not want to force a write to NV here. The counter value will
1348                // stay in RAM until the next shutdown or rollover.
1349                UINT64_TO_BYTE_ARRAY(counter, currentRamAddr + sizeof(NV_RAM_HEADER));
1350            }
1351        }
1352        return TRUE;
1353    }
```

8.4.5.31 NvCapGetCounterAvail()

This function returns an estimate of the number of additional counter type NV indexes that can be defined.

```
1354    UINT32
1355    NvCapGetCounterAvail(
1356        void
1357        )
1358    {
1359        UINT32          availNVSpace;
1360        UINT32          availRAMSpace;
1361        UINT32          persistentNum = NvCapGetPersistentNumber();
1362        UINT32          reserved = sizeof(NV_LIST_TERMINATOR);
1363    //
1364        // Get the available space in NV storage
1365        availNVSpace = NvGetFreeBytes();
1366
1367        if(persistentNum < MIN_EVICT_OBJECTS)
1368        {
1369            // Some space has to be reserved for evict object. Adjust availNVSpace.
1370            reserved += (MIN_EVICT_OBJECTS - persistentNum) * NV_EVICT_OBJECT_SIZE;
1371            if(reserved > availNVSpace)
1372                availNVSpace = 0;
1373            else
1374                availNVSpace -= reserved;
1375        }
1376        // Compute the available space in RAM
1377        availRAMSpace = (int)(RAM_ORDERLY_END - NvRamGetEnd());
1378
1379        // Return the min of counter number in NV and in RAM
1380        if(availNVSpace / NV_INDEX_COUNTER_SIZE
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1381            > availRAMSpace / NV_RAM_INDEX_COUNTER_SIZE)
1382            return availRAMSpace / NV_RAM_INDEX_COUNTER_SIZE;
1383        else
1384            return availNVSpace / NV_INDEX_COUNTER_SIZE;
1385    }
```

8.4.5.32 NvFindHandle()

this function returns the offset in NV memory of the entity associated with the input handle. A value of zero indicates that handle does not exist reference an existing persistent object or defined NV Index.

```
1386    NV_REF
1387    NvFindHandle(
1388        TPM_HANDLE       handle
1389        )
1390    {
1391        NV_REF           addr;
1392        NV_REF           iter = NV_REF_INIT;
1393        TPM_HANDLE       nextHandle;
1394    //
1395        while((addr = NvNext(&iter, &nextHandle)) != 0)
1396        {
1397            if(nextHandle == handle)
1398                break;
1399        }
1400        return addr;
1401    }
```

8.4.6 NV Max Counter

8.4.6.1 Introduction

The TPM keeps track of the highest value of a deleted counter index. When an index is deleted, this value is updated if the deleted counter index is greater than the previous value. When a new index is created and first incremented, it will get a value that is at least one greater than any other index than any previously deleted index. This insures that it is not possible to roll back an index.

The highest counter value is keep in NV in a special end-of-list marker. This marker is only updated when an index is deleted. Otherwise it just moves.

When the TPM starts up, it searches NV for the end of list marker and initializes an in memory value (*s_maxCounter*).

8.4.6.2 NvReadMaxCount()

This function returns the max NV counter value.

```
1402    UINT64
1403    NvReadMaxCount(
1404        void
1405        )
1406    {
1407        return s_maxCounter;
1408    }
```

8.4.6.3 NvUpdateMaxCount()

This function updates the max counter value to NV memory. This is just staging for the actual write that will occur when the NV index memory is modified.

 Copyright © TCG 2006-2020

```
1409 void
1410 NvUpdateMaxCount(
1411     UINT64          count
1412     )
1413 {
1414     if(count > s_maxCounter)
1415         s_maxCounter = count;
1416 }
```

8.4.6.4 NvSetMaxCount()

This function is used at NV initialization time to set the initial value of the maximum counter.

```
1417 void
1418 NvSetMaxCount(
1419     UINT64          value
1420     )
1421 {
1422     s_maxCounter = value;
1423 }
```

8.4.6.5 NvGetMaxCount()

Function to get the NV max counter value from the end-of-list marker

```
1424 UINT64
1425 NvGetMaxCount(
1426     void
1427     )
1428 {
1429     NV_REF                 iter = NV_REF_INIT;
1430     NV_REF                 currentAddr;
1431     UINT64                 maxCount;
1432 //
1433     // Find the end of list marker and initialize the NV Max Counter value.
1434     while((currentAddr = NvNext(&iter, NULL )) != 0);
1435     // 'iter' should be pointing at the end of list marker so read in the current
1436     // value of the s_maxCounter.
1437     NvRead(&maxCount, iter + sizeof(UINT32), sizeof(maxCount));
1438
1439     return maxCount;
1440 }
```

8.5 NvReserved.c

8.5.1 Introduction

The NV memory is divided into two areas: dynamic space for user defined NV Indices and evict objects, and reserved space for TPM persistent and state save data.

The entries in dynamic space are a linked list of entries. Each entry has, as its first field, a size. If the size field is zero, it marks the end of the list.

An allocation of an Index or evict object may use almost all of the remaining NV space such that the size field will not fit. The functions that search the list are aware of this and will terminate the search if they either find a zero size or recognize that there is insufficient space for the size field.

An Index allocation will contain an NV_INDEX structure. If the Index does not have the orderly attribute, the NV_INDEX is followed immediately by the NV data.

An evict object entry contains a handle followed by an OBJECT structure. This results in both the Index and Evict Object having an identifying handle as the first field following the size field.

When an Index has the orderly attribute, the data is kept in RAM. This RAM is saved to backing store in NV memory on any orderly shutdown. The entries in orderly memory are also a linked list using a size field as the first entry. As with the NV memory, the list is terminated by a zero size field or when the last entry leaves insufficient space for the terminating size field.

The attributes of an orderly index are maintained in RAM memory in order to reduce the number of NV writes needed for orderly data. When an orderly index is created, an entry is made in the dynamic NV memory space that holds the Index authorizations (*authPolicy* and *authValue*) and the size of the data. This entry is only modified if the *authValue* of the index is changed. The more volatile data of the index is kept in RAM. When an orderly Index is created or deleted, the RAM data is copied to NV backing store so that the image in the backing store matches the layout of RAM. In normal operation. The RAM data is also copied on any orderly shutdown. In normal operation, the only other reason for writing to the backing store for RAM is when a counter is first written (TPMA_NV_WRITTEN changes from CLEAR to SET) or when a counter "rolls over."

Static space contains items that are individually modifiable. The values are in the *gp* PERSISTEND_DATA structure in RAM and mapped to locations in NV.

8.5.2 Includes, Defines

```
1    #define NV_C
2    #include    "Tpm.h"
```

8.5.3 Functions

8.5.3.1 NvInitStatic()

This function initializes the static variables used in the NV subsystem.

```
3    static void
4    NvInitStatic(
5        void
6        )
7    {
8        // In some implementations, the end of NV is variable and is set at boot time.
9        // This value will be the same for each boot, but is not necessarily known
10       // at compile time.
11       s_evictNvEnd = (NV_REF)NV_MEMORY_SIZE;
12       return;
```

```
13 }
```

8.5.3.2 NvCheckState()

Function to check the NV state by accessing the platform-specific function to get the NV state. The result state is registered in *s_NvIsAvailable* that will be reported by NvIsAvailable().

This function is called at the beginning of ExecuteCommand() before any potential check of *g_NvStatus*.

```
14 void
15 NvCheckState(
16     void
17     )
18 {
19     int      func_return;
20 //
21     func_return = _plat__IsNvAvailable();
22     if(func_return == 0)
23         g_NvStatus = TPM_RC_SUCCESS;
24     else if(func_return == 1)
25         g_NvStatus = TPM_RC_NV_UNAVAILABLE;
26     else
27         g_NvStatus = TPM_RC_NV_RATE;
28     return;
29 }
```

8.5.3.3 NvCommit

This is a wrapper for the platform function to commit pending NV writes.

```
30 BOOL
31 NvCommit(
32     void
33     )
34 {
35     return (_plat__NvCommit() == 0);
36 }
```

8.5.3.4 NvPowerOn()

This function is called at _TPM_Init() to initialize the NV environment.

Return Value	Meaning
TRUE(1)	all NV was initialized
FALSE(0)	the NV containing saved state had an error and TPM2_Startup(CLEAR) is required

```
37 BOOL
38 NvPowerOn(
39     void
40     )
41 {
42     int          nvError = 0;
43     // If power was lost, need to re-establish the RAM data that is loaded from
44     // NV and initialize the static variables
45     if(g_powerWasLost)
46     {
47         if((nvError = _plat__NVEnable(0)) < 0)
48             FAIL(FATAL_ERROR_NV_UNRECOVERABLE);
```

 Copyright © TCG 2006-2020

```
49          NvInitStatic();
50      }
51      return nvError == 0;
52  }
```

8.5.3.5 NvManufacture()

This function initializes the NV system at pre-install time.

This function should only be called in a manufacturing environment or in a simulation.

The layout of NV memory space is an implementation choice.

```
53  void
54  NvManufacture(
55      void
56      )
57  {
58  #if SIMULATION
59      // Simulate the NV memory being in the erased state.
60      _plat__NvMemoryClear(0, NV_MEMORY_SIZE);
61  #endif
62      // Initialize static variables
63      NvInitStatic();
64      // Clear the RAM used for Orderly Index data
65      MemorySet(s_indexOrderlyRam, 0, RAM_INDEX_SPACE);
66      // Write that Orderly Index data to NV
67      NvUpdateIndexOrderlyData();
68      // Initialize the next offset of the first entry in evict/index list to 0 (the
69      // end of list marker) and the initial s_maxCounterValue;
70      NvSetMaxCount(0);
71      // Put the end of list marker at the end of memory. This contains the MaxCount
72      // value as well as the end marker.
73      NvWriteNvListEnd(NV_USER_DYNAMIC);
74      return;
75  }
```

8.5.3.6 NvRead()

This function is used to move reserved data from NV memory to RAM.

```
76  void
77  NvRead(
78      void            *outBuffer,     // OUT: buffer to receive data
79      UINT32           nvOffset,      // IN: offset in NV of value
80      UINT32           size           // IN: size of the value to read
81      )
82  {
83      // Input type should be valid
84      pAssert(nvOffset + size < NV_MEMORY_SIZE);
85      _plat__NvMemoryRead(nvOffset, size, outBuffer);
86      return;
87  }
```

8.5.3.7 NvWrite()

This function is used to post reserved data for writing to NV memory. Before the TPM completes the operation, the value will be written.

```
88  BOOL
89  NvWrite(
90      UINT32           nvOffset,      // IN: location in NV to receive data
```

```
 91         UINT32            size,          // IN: size of the data to move
 92         void             *inBuffer       // IN: location containing data to write
 93         )
 94     {
 95         // Input type should be valid
 96         if(nvOffset + size <= NV_MEMORY_SIZE)
 97         {
 98             // Set the flag that a NV write happened
 99             SET_NV_UPDATE(UT_NV);
100             return _plat__NvMemoryWrite(nvOffset, size, inBuffer);
101         }
102         return FALSE;
103     }
```

8.5.3.8 NvUpdatePersistent()

This function is used to update a value in the PERSISTENT_DATA structure and commits the value to NV.

```
104     void
105     NvUpdatePersistent(
106         UINT32            offset,        // IN: location in PERMANENT_DATA to be updated
107         UINT32            size,          // IN: size of the value
108         void             *buffer         // IN: the new data
109         )
110     {
111         pAssert(offset + size <= sizeof(gp));
112         MemoryCopy(&gp + offset, buffer, size);
113         NvWrite(offset, size, buffer);
114     }
```

8.5.3.9 NvClearPersistent()

This function is used to clear a persistent data entry and commit it to NV

```
115     void
116     NvClearPersistent(
117         UINT32            offset,        // IN: the offset in the PERMANENT_DATA
118                                          //     structure to be cleared (zeroed)
119         UINT32            size           // IN: number of bytes to clear
120         )
121     {
122         pAssert(offset + size <= sizeof(gp));
123         MemorySet((&gp) + offset, 0, size);
124         NvWrite(offset, size, (&gp) + offset);
125     }
```

8.5.3.10 NvReadPersistent()

This function reads persistent data to the RAM copy of the *gp* structure.

```
126     void
127     NvReadPersistent(
128         void
129         )
130     {
131         NvRead(&gp, NV_PERSISTENT_DATA, sizeof(gp));
132         return;
133     }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

8.6 Object.c

8.6.1 Introduction

This file contains the functions that manage the object store of the TPM.

8.6.2 Includes and Data Definitions

```
1    #define OBJECT_C
2    #include "Tpm.h"
```

8.6.3 Functions

8.6.3.1 ObjectFlush()

This function marks an object slot as available. Since there is no checking of the input parameters, it should be used judiciously.

NOTE: This could be converted to a macro.

```
3    void
4    ObjectFlush(
5        OBJECT          *object
6        )
7    {
8        object->attributes.occupied = CLEAR;
9    }
```

8.6.3.2 ObjectSetInUse()

This access function sets the occupied attribute of an object slot.

```
10   void
11   ObjectSetInUse(
12       OBJECT          *object
13       )
14   {
15       object->attributes.occupied = SET;
16   }
```

8.6.3.3 ObjectStartup()

This function is called at TPM2_Startup() to initialize the object subsystem.

```
17   BOOL
18   ObjectStartup(
19       void
20       )
21   {
22       UINT32      i;
23   //
24       // object slots initialization
25       for(i = 0; i < MAX_LOADED_OBJECTS; i++)
26       {
27           //Set the slot to not occupied
28           ObjectFlush(&s_objects[i]);
29       }
```

```
30      return TRUE;
31  }
```

8.6.3.4 ObjectCleanupEvict()

In this implementation, a persistent object is moved from NV into an object slot for processing. It is flushed after command execution. This function is called from ExecuteCommand().

```
32  void
33  ObjectCleanupEvict(
34      void
35      )
36  {
37      UINT32      i;
38  //
39      // This has to be iterated because a command may have two handles
40      // and they may both be persistent.
41      // This could be made to be more efficient so that a search is not needed.
42      for(i = 0; i < MAX_LOADED_OBJECTS; i++)
43      {
44          // If an object is a temporary evict object, flush it from slot
45          OBJECT      *object = &s_objects[i];
46          if(object->attributes.evict == SET)
47              ObjectFlush(object);
48      }
49      return;
50  }
```

8.6.3.5 IsObjectPresent()

This function checks to see if a transient handle references a loaded object. This routine should not be called if the handle is not a transient handle. The function validates that the handle is in the implementation-dependent allowed in range for loaded transient objects.

Return Value	Meaning
TRUE(1)	handle references a loaded object
FALSE(0)	handle is not an object handle, or it does not reference to a loaded object

```
51  BOOL
52  IsObjectPresent(
53      TPMI_DH_OBJECT   handle         // IN: handle to be checked
54      )
55  {
56      UINT32          slotIndex = handle - TRANSIENT_FIRST;
57      // Since the handle is just an index into the array that is zero based, any
58      // handle value outsize of the range of:
59      //    TRANSIENT_FIRST -- (TRANSIENT_FIRST + MAX_LOADED_OBJECT - 1)
60      // will now be greater than or equal to MAX_LOADED_OBJECTS
61      if(slotIndex >= MAX_LOADED_OBJECTS)
62          return FALSE;
63      // Indicate if the slot is occupied
64      return (s_objects[slotIndex].attributes.occupied == TRUE);
65  }
```

8.6.3.6 ObjectIsSequence()

This function is used to check if the object is a sequence object. This function should not be called if the handle does not reference a loaded object.

Return Value	Meaning
TRUE(1)	object is an HMAC, hash, or event sequence object
FALSE(0)	object is not an HMAC, hash, or event sequence object

```
66  BOOL
67  ObjectIsSequence(
68      OBJECT          *object         // IN: handle to be checked
69      )
70  {
71      pAssert(object != NULL);
72      return (object->attributes.hmacSeq == SET
73              || object->attributes.hashSeq == SET
74              || object->attributes.eventSeq == SET);
75  }
```

8.6.3.7 HandleToObject()

This function is used to find the object structure associated with a handle.

This function requires that *handle* references a loaded object or a permanent handle.

```
76  OBJECT*
77  HandleToObject(
78      TPMI_DH_OBJECT   handle         // IN: handle of the object
79      )
80  {
81      UINT32              index;
82  //
83      // Return NULL if the handle references a permanent handle because there is no
84      // associated OBJECT.
85      if(HandleGetType(handle) == TPM_HT_PERMANENT)
86          return NULL;
87      // In this implementation, the handle is determined by the slot occupied by the
88      // object.
89      index = handle - TRANSIENT_FIRST;
90      pAssert(index < MAX_LOADED_OBJECTS);
91      pAssert(s_objects[index].attributes.occupied);
92      return &s_objects[index];
93  }
```

8.6.3.8 GetQualifiedName()

This function returns the Qualified Name of the object. In this implementation, the Qualified Name is computed when the object is loaded and is saved in the internal representation of the object. The alternative would be to retain the Name of the parent and compute the QN when needed. This would take the same amount of space so it is not recommended that the alternate be used.

This function requires that *handle* references a loaded object.

```
94  void
95  GetQualifiedName(
96      TPMI_DH_OBJECT   handle,        // IN: handle of the object
97      TPM2B_NAME      *qualifiedName  // OUT: qualified name of the object
98      )
99  {
100     OBJECT      *object;
101 //
102     switch(HandleGetType(handle))
103     {
104         case TPM_HT_PERMANENT:
```

```
105                 qualifiedName->t.size = sizeof(TPM_HANDLE);
106                 UINT32_TO_BYTE_ARRAY(handle, qualifiedName->t.name);
107                 break;
108             case TPM_HT_TRANSIENT:
109                 object = HandleToObject(handle);
110                 if(object == NULL || object->publicArea.nameAlg == TPM_ALG_NULL)
111                     qualifiedName->t.size = 0;
112                 else
113                     // Copy the name
114                     *qualifiedName = object->qualifiedName;
115                 break;
116             default:
117                 FAIL(FATAL_ERROR_INTERNAL);
118         }
119         return;
120     }
```

8.6.3.9 ObjectGetHierarchy()

This function returns the handle for the hierarchy of an object.

```
121     TPMI_RH_HIERARCHY
122     ObjectGetHierarchy(
123         OBJECT          *object         // IN :object
124         )
125     {
126         if(object->attributes.spsHierarchy)
127         {
128             return TPM_RH_OWNER;
129         }
130         else if(object->attributes.epsHierarchy)
131         {
132             return TPM_RH_ENDORSEMENT;
133         }
134         else if(object->attributes.ppsHierarchy)
135         {
136             return TPM_RH_PLATFORM;
137         }
138         else
139         {
140             return TPM_RH_NULL;
141         }
142     }
```

8.6.3.10 GetHeriarchy()

This function returns the handle of the hierarchy to which a handle belongs. This function is similar to ObjectGetHierarchy() but this routine takes a handle while ObjectGetHierarchy() takes an pointer to an object.

This function requires that *handle* references a loaded object.

```
143     TPMI_RH_HIERARCHY
144     GetHeriarchy(
145         TPMI_DH_OBJECT   handle         // IN :object handle
146         )
147     {
148         OBJECT          *object = HandleToObject(handle);
149     //
150         return ObjectGetHierarchy(object);
151     }
```

 Copyright © TCG 2006-2020

8.6.3.11 FindEmptyObjectSlot()

This function finds an open object slot, if any. It will clear the attributes but will not set the occupied attribute. This is so that a slot may be used and discarded if everything does not go as planned.

Return Value	Meaning
NULL	no open slot found
NULL	pointer to available slot

```
152 OBJECT *
153 FindEmptyObjectSlot(
154     TPMI_DH_OBJECT   *handle          // OUT: (optional)
155     )
156 {
157     UINT32               i;
158     OBJECT              *object;
159 //
160     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
161     {
162         object = &s_objects[i];
163         if(object->attributes.occupied == CLEAR)
164         {
165             if(handle)
166                 *handle = i + TRANSIENT_FIRST;
167             // Initialize the object attributes
168             MemorySet(&object->attributes, 0, sizeof(OBJECT_ATTRIBUTES));
169             return object;
170         }
171     }
172     return NULL;
173 }
```

8.6.3.12 ObjectAllocateSlot()

This function is used to allocate a slot in internal object array.

```
174 OBJECT *
175 ObjectAllocateSlot(
176     TPMI_DH_OBJECT   *handle         // OUT: handle of allocated object
177     )
178 {
179     OBJECT           *object = FindEmptyObjectSlot(handle);
180 //
181     if(object != NULL)
182     {
183         // if found, mark as occupied
184         ObjectSetInUse(object);
185     }
186     return object;
187 }
```

8.6.3.13 ObjectSetLoadedAttributes()

This function sets the internal attributes for a loaded object. It is called to finalize the OBJECT attributes (not the TPMA_OBJECT attributes) for a loaded object.

```
188 void
189 ObjectSetLoadedAttributes(
190     OBJECT           *object,         // IN: object attributes to finalize
191     TPM_HANDLE        parentHandle    // IN: the parent handle
```

 Copyright © TCG 2006-2020

```
192         )
193     {
194         OBJECT                *parent = HandleToObject(parentHandle);
195         TPMA_OBJECT            objectAttributes = object->publicArea.objectAttributes;
196     //
197         // Copy the stClear attribute from the public area. This could be overwritten
198         // if the parent has stClear SET
199         object->attributes.stClear =
200             IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, stClear);
201         // If parent handle is a permanent handle, it is a primary (unless it is NULL
202         if(parent == NULL)
203         {
204             object->attributes.primary = SET;
205             switch(parentHandle)
206             {
207                 case TPM_RH_ENDORSEMENT:
208                     object->attributes.epsHierarchy = SET;
209                     break;
210                 case TPM_RH_OWNER:
211                     object->attributes.spsHierarchy = SET;
212                     break;
213                 case TPM_RH_PLATFORM:
214                     object->attributes.ppsHierarchy = SET;
215                     break;
216                 default:
217                     // Treat the temporary attribute as a hierarchy
218                     object->attributes.temporary = SET;
219                     object->attributes.primary = CLEAR;
220                     break;
221             }
222         }
223         else
224         {
225             // is this a stClear object
226             object->attributes.stClear =
227                 (IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, stClear)
228                  || (parent->attributes.stClear == SET));
229             object->attributes.epsHierarchy = parent->attributes.epsHierarchy;
230             object->attributes.spsHierarchy = parent->attributes.spsHierarchy;
231             object->attributes.ppsHierarchy = parent->attributes.ppsHierarchy;
232             // An object is temporary if its parent is temporary or if the object
233             // is external
234             object->attributes.temporary = parent->attributes.temporary
235                 || object->attributes.external;
236         }
237         // If this is an external object, set the QN == name but don't SET other
238         // key properties ('parent' or 'derived')
239         if(object->attributes.external)
240             object->qualifiedName = object->name;
241         else
242         {
243             // check attributes for different types of parents
244             if(IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, restricted)
245                && !object->attributes.publicOnly
246                && IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, decrypt)
247                && object->publicArea.nameAlg != TPM_ALG_NULL)
248             {
249                 // This is a parent. If it is not a KEYEDHASH, it is an ordinary parent.
250                 // Otherwise, it is a derivation parent.
251                 if(object->publicArea.type == TPM_ALG_KEYEDHASH)
252                     object->attributes.derivation = SET;
253                 else
254                     object->attributes.isParent = SET;
255             }
256             ComputeQualifiedName(parentHandle, object->publicArea.nameAlg,
257                                  &object->name, &object->qualifiedName);
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
258         }
259         // Set slot occupied
260         ObjectSetInUse(object);
261         return;
262     }
```

8.6.3.14 ObjectLoad()

Common function to load an object. A loaded object has its public area validated (unless its *nameAlg* is TPM_ALG_NULL). If a sensitive part is loaded, it is verified to be correct and if both public and sensitive parts are loaded, then the cryptographic binding between the objects is validated. This function does not cause the allocated slot to be marked as in use.

```
263     TPM_RC
264     ObjectLoad(
265         OBJECT           *object,         // IN: pointer to object slot
266                                           //     object
267         OBJECT           *parent,         // IN: (optional) the parent object
268         TPMT_PUBLIC      *publicArea,     // IN: public area to be installed in the object
269         TPMT_SENSITIVE   *sensitive,      // IN: (optional) sensitive area to be
270                                           //      installed in the object
271         TPM_RC            blamePublic,    // IN: parameter number to associate with the
272                                           //     publicArea errors
273         TPM_RC            blameSensitive,// IN: parameter number to associate with the
274                                           //     sensitive area errors
275         TPM2B_NAME       *name            // IN: (optional)
276     )
277     {
278         TPM_RC            result = TPM_RC_SUCCESS;
279     //
280     // Do validations of public area object descriptions
281         pAssert(publicArea != NULL);
282
283         // Is this public only or a no-name object?
284         if(sensitive == NULL || publicArea->nameAlg == TPM_ALG_NULL)
285         {
286             // Need to have schemes checked so that we do the right thing with the
287             // public key.
288             result = SchemeChecks(NULL, publicArea);
289         }
290         else
291         {
292             // For any sensitive area, make sure that the seedSize is no larger than the
293             // digest size of nameAlg
294             if(sensitive->seedValue.t.size > CryptHashGetDigestSize(publicArea->nameAlg))
295                 return TPM_RCS_KEY_SIZE + blameSensitive;
296             // Check attributes and schemes for consistency
297             result = PublicAttributesValidation(parent, publicArea);
298         }
299         if(result != TPM_RC_SUCCESS)
300             return RcSafeAddToResult(result, blamePublic);
301
302     // Sensitive area and binding checks
303
304         // On load, check nothing if the parent is fixedTPM. For all other cases, validate
305         // the keys.
306         if((parent == NULL)
307            || ((parent != NULL) && !IS_ATTRIBUTE(parent->publicArea.objectAttributes,
308                                                  TPMA_OBJECT, fixedTPM)))
309         {
310             // Do the cryptographic key validation
311             result = CryptValidateKeys(publicArea, sensitive, blamePublic,
312                                        blameSensitive);
313             if(result != TPM_RC_SUCCESS)
```

```
314                 return result;
315         }
316     #if ALG_RSA
317         // If this is an RSA key, then expand the private exponent.
318         // Note: ObjectLoad() is only called by TPM2_Import() if the parent is fixedTPM.
319         // For any key that does not have a fixedTPM parent, the exponent is computed
320         // whenever it is loaded
321         if((publicArea->type == TPM_ALG_RSA) && (sensitive != NULL))
322         {
323             result = CryptRsaLoadPrivateExponent(publicArea, sensitive);
324             if(result != TPM_RC_SUCCESS)
325                 return result;
326         }
327     #endif // ALG_RSA
328         // See if there is an object to populate
329         if((result == TPM_RC_SUCCESS) && (object != NULL))
330         {
331             // Initialize public
332             object->publicArea = *publicArea;
333             // Copy sensitive if there is one
334             if(sensitive == NULL)
335                 object->attributes.publicOnly = SET;
336             else
337                 object->sensitive = *sensitive;
338             // Set the name, if one was provided
339             if(name != NULL)
340                 object->name = *name;
341             else
342                 object->name.t.size = 0;
343         }
344         return result;
345     }
```

8.6.3.15 AllocateSequenceSlot()

This function allocates a sequence slot and initializes the parts that are used by the normal objects so that a sequence object is not inadvertently used for an operation that is not appropriate for a sequence.

```
346     static HASH_OBJECT *
347     AllocateSequenceSlot(
348         TPM_HANDLE      *newHandle,     // OUT: receives the allocated handle
349         TPM2B_AUTH      *auth           // IN: the authValue for the slot
350         )
351     {
352         HASH_OBJECT      *object = (HASH_OBJECT *)ObjectAllocateSlot(newHandle);
353     //
354         // Validate that the proper location of the hash state data relative to the
355         // object state data. It would be good if this could have been done at compile
356         // time but it can't so do it in something that can be removed after debug.
357         cAssert(offsetof(HASH_OBJECT, auth) == offsetof(OBJECT, publicArea.authPolicy));
358
359         if(object != NULL)
360         {
361
362         // Set the common values that a sequence object shares with an ordinary object
363             // First, clear all attributes
364             MemorySet(&object->objectAttributes, 0, sizeof(TPMA_OBJECT));
365
366             // The type is TPM_ALG_NULL
367             object->type = TPM_ALG_NULL;
368
369             // This has no name algorithm and the name is the Empty Buffer
370             object->nameAlg = TPM_ALG_NULL;
371
```

```
372         // A sequence object is considered to be in the NULL hierarchy so it should
373         // be marked as temporary so that it can't be persisted
374         object->attributes.temporary = SET;
375
376         // A sequence object is DA exempt.
377         SET_ATTRIBUTE(object->objectAttributes, TPMA_OBJECT, noDA);
378
379         // Copy the authorization value
380         if(auth != NULL)
381             object->auth = *auth;
382         else
383             object->auth.t.size = 0;
384     }
385     return object;
386 }
387 #if CC_HMAC_Start || CC_MAC_Start
```

8.6.3.16 ObjectCreateHMACSequence()

This function creates an internal HMAC sequence object.

Error Returns	Meaning
TPM_RC_OBJECT_MEMORY	if there is no free slot for an object

```
388 TPM_RC
389 ObjectCreateHMACSequence(
390     TPMI_ALG_HASH     hashAlg,        // IN: hash algorithm
391     OBJECT           *keyObject,      // IN: the object containing the HMAC key
392     TPM2B_AUTH       *auth,           // IN: authValue
393     TPMI_DH_OBJECT   *newHandle       // OUT: HMAC sequence object handle
394     )
395 {
396     HASH_OBJECT          *hmacObject;
397 //
398     // Try to allocate a slot for new object
399     hmacObject = AllocateSequenceSlot(newHandle, auth);
400
401     if(hmacObject == NULL)
402         return TPM_RC_OBJECT_MEMORY;
403     // Set HMAC sequence bit
404     hmacObject->attributes.hmacSeq = SET;
405
406 #if !SMAC_IMPLEMENTED
407     if(CryptHmacStart(&hmacObject->state.hmacState, hashAlg,
408                    keyObject->sensitive.sensitive.bits.b.size,
409                    keyObject->sensitive.sensitive.bits.b.buffer) == 0)
410 #else
411     if(CryptMacStart(&hmacObject->state.hmacState,
412                      &keyObject->publicArea.parameters,
413                      hashAlg, &keyObject->sensitive.sensitive.any.b) == 0)
414 #endif // SMAC_IMPLEMENTED
415         return TPM_RC_FAILURE;
416     return TPM_RC_SUCCESS;
417 }
418 #endif
```

8.6.3.17 ObjectCreateHashSequence()

This function creates a hash sequence object.

 Copyright © TCG 2006-2020

Error Returns	Meaning
TPM_RC_OBJECT_MEMORY	if there is no free slot for an object

```
419  TPM_RC
420  ObjectCreateHashSequence(
421      TPMI_ALG_HASH    hashAlg,          // IN: hash algorithm
422      TPM2B_AUTH      *auth,             // IN: authValue
423      TPMI_DH_OBJECT  *newHandle         // OUT: sequence object handle
424      )
425  {
426      HASH_OBJECT          *hashObject = AllocateSequenceSlot(newHandle, auth);
427  //
428      // See if slot allocated
429      if(hashObject == NULL)
430          return TPM_RC_OBJECT_MEMORY;
431      // Set hash sequence bit
432      hashObject->attributes.hashSeq = SET;
433
434      // Start hash for hash sequence
435      CryptHashStart(&hashObject->state.hashState[0], hashAlg);
436
437      return TPM_RC_SUCCESS;
438  }
```

8.6.3.18 ObjectCreateEventSequence()

This function creates an event sequence object.

Error Returns	Meaning
TPM_RC_OBJECT_MEMORY	if there is no free slot for an object

```
439  TPM_RC
440  ObjectCreateEventSequence(
441      TPM2B_AUTH      *auth,             // IN: authValue
442      TPMI_DH_OBJECT  *newHandle         // OUT: sequence object handle
443      )
444  {
445      HASH_OBJECT          *hashObject = AllocateSequenceSlot(newHandle, auth);
446      UINT32                count;
447      TPM_ALG_ID            hash;
448  //
449      // See if slot allocated
450      if(hashObject == NULL)
451          return TPM_RC_OBJECT_MEMORY;
452      // Set the event sequence attribute
453      hashObject->attributes.eventSeq = SET;
454
455      // Initialize hash states for each implemented PCR algorithms
456      for(count = 0; (hash = CryptHashGetAlgByIndex(count)) != TPM_ALG_NULL; count++)
457          CryptHashStart(&hashObject->state.hashState[count], hash);
458      return TPM_RC_SUCCESS;
459  }
```

8.6.3.19 ObjectTerminateEvent()

This function is called to close out the event sequence and clean up the hash context states.

```
460  void
461  ObjectTerminateEvent(
462      void
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
463         )
464     {
465         HASH_OBJECT          *hashObject;
466         int                   count;
467         BYTE                  buffer[MAX_DIGEST_SIZE];
468     //
469         hashObject = (HASH_OBJECT *)HandleToObject(g_DRTMHandle);
470
471         // Don't assume that this is a proper sequence object
472         if(hashObject->attributes.eventSeq)
473         {
474             // If it is, close any open hash contexts. This is done in case
475             // the cryptographic implementation has some context values that need to be
476             // cleaned up (hygiene).
477             //
478             for(count = 0; CryptHashGetAlgByIndex(count) != TPM_ALG_NULL; count++)
479             {
480                 CryptHashEnd(&hashObject->state.hashState[count], 0, buffer);
481             }
482             // Flush sequence object
483             FlushObject(g_DRTMHandle);
484         }
485         g_DRTMHandle = TPM_RH_UNASSIGNED;
486     }
```

8.6.3.20 ObjectContextLoad()

This function loads an object from a saved object context.

Return Value	Meaning
NULL	if there is no free slot for an object
NULL	points to the loaded object

```
487     OBJECT *
488     ObjectContextLoad(
489         ANY_OBJECT_BUFFER   *object,         // IN: pointer to object structure in saved
490                                              //     context
491         TPMI_DH_OBJECT      *handle          // OUT: object handle
492         )
493     {
494         OBJECT      *newObject = ObjectAllocateSlot(handle);
495     //
496         // Try to allocate a slot for new object
497         if(newObject != NULL)
498         {
499             // Copy the first part of the object
500             MemoryCopy(newObject, object, offsetof(HASH_OBJECT, state));
501             // See if this is a sequence object
502             if(ObjectIsSequence(newObject))
503             {
504                 // If this is a sequence object, import the data
505                 SequenceDataImport((HASH_OBJECT *)newObject,
506                                    (HASH_OBJECT_BUFFER *)object);
507             }
508             else
509             {
510                 // Copy input object data to internal structure
511                 MemoryCopy(newObject, object, sizeof(OBJECT));
512             }
513         }
514         return newObject;
515     }
```

 Copyright © TCG 2006-2020

8.6.3.21 FlushObject()

This function frees an object slot.

This function requires that the object is loaded.

```
516 void
517 FlushObject(
518     TPMI_DH_OBJECT    handle          // IN: handle to be freed
519     )
520 {
521     UINT32       index = handle - TRANSIENT_FIRST;
522 //
523     pAssert(index < MAX_LOADED_OBJECTS);
524     // Clear all the object attributes
525     MemorySet((BYTE*)&(s_objects[index].attributes),
526               0, sizeof(OBJECT_ATTRIBUTES));
527     return;
528 }
```

8.6.3.22 ObjectFlushHierarchy()

This function is called to flush all the loaded transient objects associated with a hierarchy when the hierarchy is disabled.

```
529 void
530 ObjectFlushHierarchy(
531     TPMI_RH_HIERARCHY     hierarchy      // IN: hierarchy to be flush
532     )
533 {
534     UINT16          i;
535 //
536     // iterate object slots
537     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
538     {
539         if(s_objects[i].attributes.occupied)          // If found an occupied slot
540         {
541             switch(hierarchy)
542             {
543                 case TPM_RH_PLATFORM:
544                     if(s_objects[i].attributes.ppsHierarchy == SET)
545                         s_objects[i].attributes.occupied = FALSE;
546                     break;
547                 case TPM_RH_OWNER:
548                     if(s_objects[i].attributes.spsHierarchy == SET)
549                         s_objects[i].attributes.occupied = FALSE;
550                     break;
551                 case TPM_RH_ENDORSEMENT:
552                     if(s_objects[i].attributes.epsHierarchy == SET)
553                         s_objects[i].attributes.occupied = FALSE;
554                     break;
555                 default:
556                     FAIL(FATAL_ERROR_INTERNAL);
557                     break;
558             }
559         }
560     }
561
562     return;
563 }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

8.6.3.23 ObjectLoadEvict()

This function loads a persistent object into a transient object slot.

This function requires that *handle* is associated with a persistent object.

Error Returns	Meaning
TPM_RC_HANDLE	the persistent object does not exist or the associated hierarchy is disabled.
TPM_RC_OBJECT_MEMORY	no object slot

```
564 TPM_RC
565 ObjectLoadEvict(
566     TPM_HANDLE      *handle,        // IN:OUT: evict object handle.  If success, it
567                                     // will be replace by the loaded object handle
568     COMMAND_INDEX    commandIndex   // IN: the command being processed
569     )
570 {
571     TPM_RC          result;
572     TPM_HANDLE      evictHandle = *handle;   // Save the evict handle
573     OBJECT          *object;
574 //
575     // If this is an index that references a persistent object created by
576     // the platform, then return TPM_RH_HANDLE if the phEnable is FALSE
577     if(*handle >= PLATFORM_PERSISTENT)
578     {
579         // belongs to platform
580         if(g_phEnable == CLEAR)
581             return TPM_RC_HANDLE;
582     }
583     // belongs to owner
584     else if(gc.shEnable == CLEAR)
585         return TPM_RC_HANDLE;
586     // Try to allocate a slot for an object
587     object = ObjectAllocateSlot(handle);
588     if(object == NULL)
589         return TPM_RC_OBJECT_MEMORY;
590     // Copy persistent object to transient object slot.  A TPM_RC_HANDLE
591     // may be returned at this point. This will mark the slot as containing
592     // a transient object so that it will be flushed at the end of the
593     // command
594     result = NvGetEvictObject(evictHandle, object);
595
596     // Bail out if this failed
597     if(result != TPM_RC_SUCCESS)
598         return result;
599     // check the object to see if it is in the endorsement hierarchy
600     // if it is and this is not a TPM2_EvictControl() command, indicate
601     // that the hierarchy is disabled.
602     // If the associated hierarchy is disabled, make it look like the
603     // handle is not defined
604     if(ObjectGetHierarchy(object) == TPM_RH_ENDORSEMENT
605        && gc.ehEnable == CLEAR
606        && GetCommandCode(commandIndex) != TPM_CC_EvictControl)
607         return TPM_RC_HANDLE;
608
609     return result;
610 }
```

8.6.3.24 ObjectComputeName()

This does the name computation from a public area (can be marshaled or not).

```
611 TPM2B_NAME *
612 ObjectComputeName(
613     UINT32           size,          // IN: the size of the area to digest
614     BYTE            *publicArea,    // IN: the public area to digest
615     TPM_ALG_ID       nameAlg,       // IN: the hash algorithm to use
616     TPM2B_NAME      *name           // OUT: Computed name
617     )
618 {
619     // Hash the publicArea into the name buffer leaving room for the nameAlg
620     name->t.size = CryptHashBlock(nameAlg, size, publicArea,
621                                   sizeof(name->t.name) - 2,
622                                   &name->t.name[2]);
623     // set the nameAlg
624     UINT16_TO_BYTE_ARRAY(nameAlg, name->t.name);
625     name->t.size += 2;
626     return name;
627 }
```

8.6.3.25 PublicMarshalAndComputeName()

This function computes the Name of an object from its public area.

```
628 TPM2B_NAME *
629 PublicMarshalAndComputeName(
630     TPMT_PUBLIC     *publicArea,    // IN: public area of an object
631     TPM2B_NAME      *name           // OUT: name of the object
632     )
633 {
634     // Will marshal a public area into a template. This is because the internal
635     // format for a TPM2B_PUBLIC is a structure and not a simple BYTE buffer.
636     TPM2B_TEMPLATE       marshaled;     // this is big enough to hold a
637                                         //  marshaled TPMT_PUBLIC
638     BYTE                *buffer = (BYTE *)&marshaled.t.buffer;
639 //
640     // if the nameAlg is NULL then there is no name.
641     if(publicArea->nameAlg == TPM_ALG_NULL)
642         name->t.size = 0;
643     else
644     {
645         // Marshal the public area into its canonical form
646         marshaled.t.size = TPMT_PUBLIC_Marshal(publicArea, &buffer, NULL);
647         // and compute the name
648         ObjectComputeName(marshaled.t.size, marshaled.t.buffer,
649                           publicArea->nameAlg, name);
650     }
651     return name;
652 }
```

8.6.3.26 ComputeQualifiedName()

This function computes the qualified name of an object.

```
653 void
654 ComputeQualifiedName(
655     TPM_HANDLE       parentHandle,  // IN: parent's handle
656     TPM_ALG_ID       nameAlg,       // IN: name hash
657     TPM2B_NAME      *name,          // IN: name of the object
658     TPM2B_NAME      *qualifiedName  // OUT: qualified name of the object
659     )
660 {
661     HASH_STATE      hashState;   // hash state
662     TPM2B_NAME      parentName;
663 //
```


 Copyright © TCG 2006-2020

```
664         if(parentHandle == TPM_RH_UNASSIGNED)
665         {
666             MemoryCopy2B(&qualifiedName->b, &name->b, sizeof(qualifiedName->t.name));
667             *qualifiedName = *name;
668         }
669         else
670         {
671             GetQualifiedName(parentHandle, &parentName);
672
673             //      QN_A = hash_A (QN of parent || NAME_A)
674
675             // Start hash
676             qualifiedName->t.size = CryptHashStart(&hashState, nameAlg);
677
678             // Add parent's qualified name
679             CryptDigestUpdate2B(&hashState, &parentName.b);
680
681             // Add self name
682             CryptDigestUpdate2B(&hashState, &name->b);
683
684             // Complete hash leaving room for the name algorithm
685             CryptHashEnd(&hashState, qualifiedName->t.size,
686                          &qualifiedName->t.name[2]);
687             UINT16_TO_BYTE_ARRAY(nameAlg, qualifiedName->t.name);
688             qualifiedName->t.size += 2;
689         }
690         return;
691     }
```

8.6.3.27 ObjectIsStorage()

This function determines if an object has the attributes associated with a parent. A parent is an asymmetric or symmetric block cipher key that has its *restricted* and *decrypt* attributes SET, and *sign* CLEAR.

Return Value	Meaning
TRUE(1)	object is a storage key
FALSE(0)	object is not a storage key

```
692     BOOL
693     ObjectIsStorage(
694         TPMI_DH_OBJECT    handle         // IN: object handle
695         )
696     {
697         OBJECT           *object = HandleToObject(handle);
698         TPMT_PUBLIC      *publicArea = ((object != NULL) ? &object->publicArea : NULL);
699     //
700         return (publicArea != NULL
701                 && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
702                 && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
703                 && !IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign)
704                 && (object->publicArea.type == ALG_RSA_VALUE
705                     || object->publicArea.type == ALG_ECC_VALUE));
706     }
```

8.6.3.28 ObjectCapGetLoaded()

This function returns a a list of handles of loaded object, starting from *handle*. *Handle* must be in the range of valid transient object handles, but does not have to be the handle of a loaded transient object.

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
707  TPMI_YES_NO
708  ObjectCapGetLoaded(
709      TPMI_DH_OBJECT    handle,        // IN: start handle
710      UINT32            count,         // IN: count of returned handles
711      TPML_HANDLE      *handleList     // OUT: list of handle
712      )
713  {
714      TPMI_YES_NO          more = NO;
715      UINT32               i;
716  //
717      pAssert(HandleGetType(handle) == TPM_HT_TRANSIENT);
718
719      // Initialize output handle list
720      handleList->count = 0;
721
722      // The maximum count of handles we may return is MAX_CAP_HANDLES
723      if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
724
725      // Iterate object slots to get loaded object handles
726      for(i = handle - TRANSIENT_FIRST; i < MAX_LOADED_OBJECTS; i++)
727      {
728          if(s_objects[i].attributes.occupied == TRUE)
729          {
730              // A valid transient object can not be the copy of a persistent object
731              pAssert(s_objects[i].attributes.evict == CLEAR);
732
733              if(handleList->count < count)
734              {
735                  // If we have not filled up the return list, add this object
736                  // handle to it
737                  handleList->handle[handleList->count] = i + TRANSIENT_FIRST;
738                  handleList->count++;
739              }
740              else
741              {
742                  // If the return list is full but we still have loaded object
743                  // available, report this and stop iterating
744                  more = YES;
745                  break;
746              }
747          }
748      }
749
750      return more;
751  }
```

8.6.3.29 ObjectCapGetTransientAvail()

This function returns an estimate of the number of additional transient objects that could be loaded into the TPM.

```
752  UINT32
753  ObjectCapGetTransientAvail(
754      void
755      )
756  {
757      UINT32       i;
758      UINT32       num = 0;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
759 //
760     // Iterate object slot to get the number of unoccupied slots
761     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
762     {
763         if(s_objects[i].attributes.occupied == FALSE) num++;
764     }
765
766     return num;
767 }
```

8.6.3.30 ObjectGetPublicAttributes()

Returns the attributes associated with an object handles.

```
768 TPMA_OBJECT
769 ObjectGetPublicAttributes(
770     TPM_HANDLE       handle
771     )
772 {
773     return HandleToObject(handle)->publicArea.objectAttributes;
774 }
775 OBJECT_ATTRIBUTES
776 ObjectGetProperties(
777     TPM_HANDLE       handle
778     )
779 {
780     return HandleToObject(handle)->attributes;
781 }
```

 Copyright © TCG 2006-2020

8.7 PCR.c

8.7.1 Introduction

This function contains the functions needed for PCR access and manipulation.

This implementation uses a static allocation for the PCR. The amount of memory is allocated based on the number of PCR in the implementation and the number of implemented hash algorithms. This is not the expected implementation. PCR SPACE DEFINITIONS.

In the definitions below, the *g_hashPcrMap* is a bit array that indicates which of the PCR are implemented. The *g_hashPcr* array is an array of digests. In this implementation, the space is allocated whether the PCR is implemented or not.

8.7.2 Includes, Defines, and Data Definitions

```
1  #define PCR_C
2  #include "Tpm.h"
```

The initial value of PCR attributes. The value of these fields should be consistent with PC Client specification In this implementation, we assume the total number of implemented PCR is 24.

```
3  static const PCR_Attributes s_initAttributes[] =
4  {
5      // PCR 0 - 15, static RTM
6      {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
7      {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
8      {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
9      {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
10
11     {0, 0x0F, 0x1F},        // PCR 16, Debug
12     {0, 0x10, 0x1C},        // PCR 17, Locality 4
13     {0, 0x10, 0x1C},        // PCR 18, Locality 3
14     {0, 0x10, 0x0C},        // PCR 19, Locality 2
15     {0, 0x14, 0x0E},        // PCR 20, Locality 1
16     {0, 0x14, 0x04},        // PCR 21, Dynamic OS
17     {0, 0x14, 0x04},        // PCR 22, Dynamic OS
18     {0, 0x0F, 0x1F},        // PCR 23, Application specific
19     {0, 0x0F, 0x1F}         // PCR 24, testing policy
20 };
21
22 //** Functions
23
24 //*** PCRBelongsAuthGroup()
25 // This function indicates if a PCR belongs to a group that requires an authValue
26 // in order to modify the PCR.  If it does, 'groupIndex' is set to value of
27 // the group index.  This feature of PCR is decided by the platform specification.
```

Return Value	Meaning
TRUE(1)	PCR belongs to an authorization group
FALSE(0)	PCR belongs to an authorization group

```
28 BOOL
29 PCRBelongsAuthGroup(
30     TPMI_DH_PCR      handle,         // IN: handle of PCR
31     UINT32          *groupIndex      // OUT: group index if PCR belongs a
32                                      //      group that allows authValue.  If PCR
33                                      //      does not belong to an authorization
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
34                                      //      group, the value in this parameter is
35                                      //      invalid
36      )
37  {
38  #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
39      // Platform specification determines to which authorization group a PCR belongs
40      // (if any). In this implementation, we assume there is only
41      // one authorization group which contains PCR[20-22].  If the platform
42      // specification requires differently, the implementation should be changed
43      // accordingly
44      if(handle >= 20 && handle <= 22)
45      {
46          *groupIndex = 0;
47          return TRUE;
48      }
49
50  #endif
51      return FALSE;
52  }
```

8.7.2.1 PCRBelongsPolicyGroup()

This function indicates if a PCR belongs to a group that requires a policy authorization in order to modify the PCR. If it does, *groupIndex* is set to value of the group index. This feature of PCR is decided by the platform specification.

Return Value	Meaning
TRUE(1)	PCR belongs to a policy group
FALSE(0)	PCR does not belong to a policy group

```
53  BOOL
54  PCRBelongsPolicyGroup(
55      TPMI_DH_PCR      handle,        // IN: handle of PCR
56      UINT32          *groupIndex     // OUT: group index if PCR belongs a group that
57                                      //     allows policy.  If PCR does not belong to
58                                      //     a policy group, the value in this
59                                      //     parameter is invalid
60      )
61  {
62  #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
63      // Platform specification decides if a PCR belongs to a policy group and
64      // belongs to which group.  In this implementation, we assume there is only
65      // one policy group which contains PCR20-22.  If the platform specification
66      // requires differently, the implementation should be changed accordingly
67      if(handle >= 20 && handle <= 22)
68      {
69          *groupIndex = 0;
70          return TRUE;
71      }
72  #endif
73      return FALSE;
74  }
```

8.7.2.2 PCRBelongsTCBGroup()

This function indicates if a PCR belongs to the TCB group.

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

Return Value	Meaning
TRUE(1)	PCR belongs to a TCB group
FALSE(0)	PCR does not belong to a TCB group

```
75   static BOOL
76   PCRBelongsTCBGroup(
77       TPMI_DH_PCR      handle          // IN: handle of PCR
78       )
79   {
80   #if ENABLE_PCR_NO_INCREMENT == YES
81       // Platform specification decides if a PCR belongs to a TCB group.  In this
82       // implementation, we assume PCR[20-22] belong to TCB group.  If the platform
83       // specification requires differently, the implementation should be
84       // changed accordingly
85       if(handle >= 20 && handle <= 22)
86           return TRUE;
87
88   #endif
89       return FALSE;
90   }
```

8.7.2.3 PCRPolicyIsAvailable()

This function indicates if a policy is available for a PCR.

Return Value	Meaning
TRUE(1)	the PCR may be authorized by policy
FALSE(0)	the PCR does not allow policy

```
91   BOOL
92   PCRPolicyIsAvailable(
93       TPMI_DH_PCR      handle          // IN: PCR handle
94       )
95   {
96       UINT32          groupIndex;
97
98       return PCRBelongsPolicyGroup(handle, &groupIndex);
99   }
```

8.7.2.4 PCRGetAuthValue()

This function is used to access the *authValue* of a PCR. If PCR does not belong to an *authValue* group, an EmptyAuth() will be returned.

```
100  TPM2B_AUTH *
101  PCRGetAuthValue(
102      TPMI_DH_PCR      handle          // IN: PCR handle
103      )
104  {
105      UINT32      groupIndex;
106
107      if(PCRBelongsAuthGroup(handle, &groupIndex))
108      {
109          return &gc.pcrAuthValues.auth[groupIndex];
110      }
111      else
112      {
113          return NULL;
```

```
114         }
115     }
```

8.7.2.5 PCRGetAuthPolicy()

This function is used to access the authorization policy of a PCR. It sets *policy* to the authorization policy and returns the hash algorithm for policy If the PCR does not allow a policy, TPM_ALG_NULL is returned.

```
116     TPMI_ALG_HASH
117     PCRGetAuthPolicy(
118         TPMI_DH_PCR      handle,        // IN: PCR handle
119         TPM2B_DIGEST    *policy         // OUT: policy of PCR
120         )
121     {
122         UINT32           groupIndex;
123
124         if(PCRBelongsPolicyGroup(handle, &groupIndex))
125         {
126             *policy = gp.pcrPolicies.policy[groupIndex];
127             return gp.pcrPolicies.hashAlg[groupIndex];
128         }
129         else
130         {
131             policy->t.size = 0;
132             return TPM_ALG_NULL;
133         }
134     }
```

8.7.2.6 PCRSimStart()

This function is used to initialize the policies when a TPM is manufactured. This function would only be called in a manufacturing environment or in a TPM simulator.

```
135     void
136     PCRSimStart(
137         void
138         )
139     {
140         UINT32  i;
141     #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
142         for(i = 0; i < NUM_POLICY_PCR_GROUP; i++)
143         {
144             gp.pcrPolicies.hashAlg[i] = TPM_ALG_NULL;
145             gp.pcrPolicies.policy[i].t.size = 0;
146         }
147     #endif
148     #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
149         for(i = 0; i < NUM_AUTHVALUE_PCR_GROUP; i++)
150         {
151             gc.pcrAuthValues.auth[i].t.size = 0;
152         }
153     #endif
154         // We need to give an initial configuration on allocated PCR before
155         // receiving any TPM2_PCR_Allocate command to change this configuration
156         // When the simulation environment starts, we allocate all the PCRs
157         for(gp.pcrAllocated.count = 0; gp.pcrAllocated.count < HASH_COUNT;
158         gp.pcrAllocated.count++)
159         {
160             gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].hash
161                 = CryptHashGetAlgByIndex(gp.pcrAllocated.count);
162
163             gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].sizeofSelect
```

```
164                 = PCR_SELECT_MAX;
165             for(i = 0; i < PCR_SELECT_MAX; i++)
166                 gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].pcrSelect[i]
167                 = 0xFF;
168         }
169
170         // Store the initial configuration to NV
171         NV_SYNC_PERSISTENT(pcrPolicies);
172         NV_SYNC_PERSISTENT(pcrAllocated);
173
174         return;
175     }
```

8.7.2.7 GetSavedPcrPointer()

This function returns the address of an array of state saved PCR based on the hash algorithm.

Return Value	Meaning
NULL	no such algorithm
NULL	pointer to the 0th byte of the 0th PCR

```
176     static BYTE *
177     GetSavedPcrPointer(
178         TPM_ALG_ID       alg,           // IN: algorithm for bank
179         UINT32           pcrIndex       // IN: PCR index in PCR_SAVE
180         )
181     {
182         BYTE            *retVal;
183         switch(alg)
184         {
185     #if ALG_SHA1
186             case ALG_SHA1_VALUE:
187                 retVal = gc.pcrSave.sha1[pcrIndex];
188                 break;
189     #endif
190     #if ALG_SHA256
191             case ALG_SHA256_VALUE:
192                 retVal = gc.pcrSave.sha256[pcrIndex];
193                 break;
194     #endif
195     #if ALG_SHA384
196             case ALG_SHA384_VALUE:
197                 retVal = gc.pcrSave.sha384[pcrIndex];
198                 break;
199     #endif
200
201     #if ALG_SHA512
202             case ALG_SHA512_VALUE:
203                 retVal = gc.pcrSave.sha512[pcrIndex];
204                 break;
205     #endif
206     #if ALG_SM3_256
207             case ALG_SM3_256_VALUE:
208                 retVal = gc.pcrSave.sm3_256[pcrIndex];
209                 break;
210     #endif
211             default:
212                 FAIL(FATAL_ERROR_INTERNAL);
213         }
214         return retVal;
215     }
```

 Copyright © TCG 2006-2020

8.7.2.8 PcrIsAllocated()

This function indicates if a PCR number for the particular hash algorithm is allocated.

Return Value	Meaning
TRUE(1)	PCR is allocated
FALSE(0)	PCR is not allocated

```
216 BOOL
217 PcrIsAllocated(
218     UINT32          pcr,            // IN: The number of the PCR
219     TPMI_ALG_HASH   hashAlg         // IN: The PCR algorithm
220     )
221 {
222     UINT32              i;
223     BOOL                allocated = FALSE;
224
225     if(pcr < IMPLEMENTATION_PCR)
226     {
227         for(i = 0; i < gp.pcrAllocated.count; i++)
228         {
229             if(gp.pcrAllocated.pcrSelections[i].hash == hashAlg)
230             {
231                 if(((gp.pcrAllocated.pcrSelections[i].pcrSelect[pcr / 8])
232                     & (1 << (pcr % 8))) != 0)
233                     allocated = TRUE;
234                 else
235                     allocated = FALSE;
236                 break;
237             }
238         }
239     }
240     return allocated;
241 }
```

8.7.2.9 GetPcrPointer()

This function returns the address of an array of PCR based on the hash algorithm.

Return Value	Meaning
NULL	no such algorithm
NULL	pointer to the 0th byte of the 0th PCR

```
242 static BYTE *
243 GetPcrPointer(
244     TPM_ALG_ID      alg,            // IN: algorithm for bank
245     UINT32          pcrNumber       // IN: PCR number
246     )
247 {
248     static BYTE     *pcr = NULL;
249
250     if(!PcrIsAllocated(pcrNumber, alg))
251         return NULL;
252
253     switch(alg)
254     {
255 #if ALG_SHA1
256         case ALG_SHA1_VALUE:
257             pcr = s_pcrs[pcrNumber].sha1Pcr;
258             break;
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
259 #endif
260 #if ALG_SHA256
261         case ALG_SHA256_VALUE:
262             pcr = s_pcrs[pcrNumber].sha256Pcr;
263             break;
264 #endif
265 #if ALG_SHA384
266         case ALG_SHA384_VALUE:
267             pcr = s_pcrs[pcrNumber].sha384Pcr;
268             break;
269 #endif
270 #if ALG_SHA512
271         case ALG_SHA512_VALUE:
272             pcr = s_pcrs[pcrNumber].sha512Pcr;
273             break;
274 #endif
275 #if ALG_SM3_256
276         case ALG_SM3_256_VALUE:
277             pcr = s_pcrs[pcrNumber].sm3_256Pcr;
278             break;
279 #endif
280         default:
281             FAIL(FATAL_ERROR_INTERNAL);
282             break;
283     }
284     return pcr;
285 }
```

8.7.2.10 IsPcrSelected()

This function indicates if an indicated PCR number is selected by the bit map in *selection*.

Return Value	Meaning
TRUE(1)	PCR is selected
FALSE(0)	PCR is not selected

```
286 static BOOL
287 IsPcrSelected(
288     UINT32              pcr,            // IN: The number of the PCR
289     TPMS_PCR_SELECTION  *selection      // IN: The selection structure
290     )
291 {
292     BOOL                selected;
293     selected = (pcr < IMPLEMENTATION_PCR
294                 && ((selection->pcrSelect[pcr / 8]) & (1 << (pcr % 8))) != 0);
295     return selected;
296 }
```

8.7.2.11 FilterPcr()

This function modifies a PCR selection array based on the implemented PCR.

```
297 static void
298 FilterPcr(
299     TPMS_PCR_SELECTION  *selection      // IN: input PCR selection
300     )
301 {
302     UINT32      i;
303     TPMS_PCR_SELECTION      *allocated = NULL;
304
305     // If size of select is less than PCR_SELECT_MAX, zero the unspecified PCR
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
306     for(i = selection->sizeofSelect; i < PCR_SELECT_MAX; i++)
307         selection->pcrSelect[i] = 0;
308
309     // Find the internal configuration for the bank
310     for(i = 0; i < gp.pcrAllocated.count; i++)
311     {
312         if(gp.pcrAllocated.pcrSelections[i].hash == selection->hash)
313         {
314             allocated = &gp.pcrAllocated.pcrSelections[i];
315             break;
316         }
317     }
318
319     for(i = 0; i < selection->sizeofSelect; i++)
320     {
321         if(allocated == NULL)
322         {
323             // If the required bank does not exist, clear input selection
324             selection->pcrSelect[i] = 0;
325         }
326         else
327             selection->pcrSelect[i] &= allocated->pcrSelect[i];
328     }
329
330     return;
331 }
```

8.7.2.12 PcrDrtm()

This function does the DRTM and H-CRTM processing it is called from _TPM_Hash_End().

```
332 void
333 PcrDrtm(
334     const TPMI_DH_PCR        pcrHandle,      // IN: the index of the PCR to be
335                                              //     modified
336     const TPMI_ALG_HASH      hash,           // IN: the bank identifier
337     const TPM2B_DIGEST      *digest          // IN: the digest to modify the PCR
338     )
339 {
340     BYTE         *pcrData = GetPcrPointer(hash, pcrHandle);
341
342     if(pcrData != NULL)
343     {
344         // Rest the PCR to zeros
345         MemorySet(pcrData, 0, digest->t.size);
346
347         // if the TPM has not started, then set the PCR to 0...04 and then extend
348         if(!TPMIsStarted())
349         {
350             pcrData[digest->t.size - 1] = 4;
351         }
352         // Now, extend the value
353         PCRExtend(pcrHandle, hash, digest->t.size, (BYTE *)digest->t.buffer);
354     }
355 }
```

8.7.2.13 PCR_ClearAuth()

This function is used to reset the PCR authorization values. It is called on TPM2_Startup(CLEAR) and TPM2_Clear().

```
356 void
357 PCR_ClearAuth(
```

```
358      void
359      )
360  {
361  #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
362      int          j;
363      for(j = 0; j < NUM_AUTHVALUE_PCR_GROUP; j++)
364      {
365          gc.pcrAuthValues.auth[j].t.size = 0;
366      }
367  #endif
368  }
```

8.7.2.14 PCRStartup()

This function initializes the PCR subsystem at TPM2_Startup().

```
369  BOOL
370  PCRStartup(
371      STARTUP_TYPE     type,          // IN: startup type
372      BYTE             locality       // IN: startup locality
373      )
374  {
375      UINT32              pcr, j;
376      UINT32              saveIndex = 0;
377
378      g_pcrReConfig = FALSE;
379
380      // Don't test for SU_RESET because that should be the default when nothing
381      // else is selected
382      if(type != SU_RESUME && type != SU_RESTART)
383      {
384          // PCR generation counter is cleared at TPM_RESET
385          gr.pcrCounter = 0;
386      }
387
388      // Initialize/Restore PCR values
389      for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
390      {
391          // On resume, need to know if this PCR had its state saved or not
392          UINT32      stateSaved;
393
394          if(type == SU_RESUME
395             &&  s_initAttributes[pcr].stateSave == SET)
396          {
397              stateSaved = 1;
398          }
399          else
400          {
401              stateSaved = 0;
402              PCRChanged(pcr);
403          }
404
405          // If this is the H-CRTM PCR and we are not doing a resume and we
406          // had an H-CRTM event, then we don't change this PCR
407          if(pcr == HCRTM_PCR && type != SU_RESUME && g_DrtmPreStartup == TRUE)
408              continue;
409
410          // Iterate each hash algorithm bank
411          for(j = 0; j < gp.pcrAllocated.count; j++)
412          {
413              TPMI_ALG_HASH    hash = gp.pcrAllocated.pcrSelections[j].hash;
414              BYTE            *pcrData = GetPcrPointer(hash, pcr);
415              UINT16           pcrSize = CryptHashGetDigestSize(hash);
416
```

 Copyright © TCG 2006-2020

```
417                if(pcrData != NULL)
418                {
419                    // if state was saved
420                    if(stateSaved == 1)
421                    {
422                        // Restore saved PCR value
423                        BYTE    *pcrSavedData;
424                        pcrSavedData = GetSavedPcrPointer(
425                            gp.pcrAllocated.pcrSelections[j].hash,
426                            saveIndex);
427                        if(pcrSavedData == NULL)
428                            return FALSE;
429                        MemoryCopy(pcrData, pcrSavedData, pcrSize);
430                    }
431                    else
432                        // PCR was not restored by state save
433                    {
434                        // If the reset locality of the PCR is 4, then
435                        // the reset value is all one's, otherwise it is
436                        // all zero.
437                        if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
438                            MemorySet(pcrData, 0xFF, pcrSize);
439                        else
440                        {
441                            MemorySet(pcrData, 0, pcrSize);
442                            if(pcr == HCRTM_PCR)
443                                pcrData[pcrSize - 1] = locality;
444                        }
445                    }
446                }
447            }
448            saveIndex += stateSaved;
449        }
450        // Reset authValues on TPM2_Startup(CLEAR)
451        if(type != SU_RESUME)
452            PCR_ClearAuth();
453        return TRUE;
454    }
```

8.7.2.15 PCRStateSave()

This function is used to save the PCR values that will be restored on TPM Resume.

```
455    void
456    PCRStateSave(
457        TPM_SU            type            // IN: startup type
458        )
459    {
460        UINT32            pcr, j;
461        UINT32            saveIndex = 0;
462
463        // if state save CLEAR, nothing to be done.  Return here
464        if(type == TPM_SU_CLEAR)
465            return;
466
467        // Copy PCR values to the structure that should be saved to NV
468        for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
469        {
470            UINT32  stateSaved = (s_initAttributes[pcr].stateSave == SET) ? 1 : 0;
471
472            // Iterate each hash algorithm bank
473            for(j = 0; j < gp.pcrAllocated.count; j++)
474            {
475                BYTE    *pcrData;
```

```
476                 UINT32   pcrSize;
477
478                 pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[j].hash, pcr);
479
480                 if(pcrData != NULL)
481                 {
482                     pcrSize
483                         = CryptHashGetDigestSize(gp.pcrAllocated.pcrSelections[j].hash);
484
485                     if(stateSaved == 1)
486                     {
487                         // Restore saved PCR value
488                         BYTE    *pcrSavedData;
489                         pcrSavedData
490                             = GetSavedPcrPointer(gp.pcrAllocated.pcrSelections[j].hash,
491                                                  saveIndex);
492                         MemoryCopy(pcrSavedData, pcrData, pcrSize);
493                     }
494                 }
495             }
496             saveIndex += stateSaved;
497         }
498
499         return;
500     }
```

8.7.2.16 PCRIsStateSaved()

This function indicates if the selected PCR is a PCR that is state saved on TPM2_Shutdown(STATE). The return value is based on PCR attributes.

Return Value	Meaning
TRUE(1)	PCR is state saved
FALSE(0)	PCR is not state saved

```
501     BOOL
502     PCRIsStateSaved(
503         TPMI_DH_PCR      handle          // IN: PCR handle to be extended
504         )
505     {
506         UINT32              pcr = handle - PCR_FIRST;
507
508         if(s_initAttributes[pcr].stateSave == SET)
509             return TRUE;
510         else
511             return FALSE;
512     }
```

8.7.2.17 PCRIsResetAllowed()

This function indicates if a PCR may be reset by the current command locality. The return value is based on PCR attributes, and not the PCR allocation.

Return Value	Meaning
TRUE(1)	TPM2_PCR_Reset() is allowed
FALSE(0)	TPM2_PCR_Reset() is not allowed

```
513     BOOL
514     PCRIsResetAllowed(
```

```
515     TPMI_DH_PCR      handle         // IN: PCR handle to be extended
516     )
517 {
518     UINT8                commandLocality;
519     UINT8                localityBits = 1;
520     UINT32               pcr = handle - PCR_FIRST;
521
522     // Check for the locality
523     commandLocality = _plat__LocalityGet();
524
525 #ifdef DRTM_PCR
526     // For a TPM that does DRTM, Reset is not allowed at locality 4
527     if(commandLocality == 4)
528         return FALSE;
529 #endif
530
531     localityBits = localityBits << commandLocality;
532     if((localityBits & s_initAttributes[pcr].resetLocality) == 0)
533         return FALSE;
534     else
535         return TRUE;
536 }
```

8.7.2.18 PCRChanged()

This function checks a PCR handle to see if the attributes for the PCR are set so that any change to the PCR causes an increment of the *pcrCounter*. If it does, then the function increments the counter. Will also bump the counter if the handle is zero which means that PCR 0 can not be in the TCB group. Bump on zero is used by TPM2_Clear().

```
537 void
538 PCRChanged(
539     TPM_HANDLE       pcrHandle      // IN: the handle of the PCR that changed.
540     )
541 {
542     // For the reference implementation, the only change that does not cause
543     // increment is a change to a PCR in the TCB group.
544     if((pcrHandle == 0) || !PCRBelongsTCBGroup(pcrHandle))
545     {
546         gr.pcrCounter++;
547         if(gr.pcrCounter == 0)
548             FAIL(FATAL_ERROR_COUNTER_OVERFLOW);
549     }
550 }
```

8.7.2.19 PCRIsExtendAllowed()

This function indicates a PCR may be extended at the current command locality. The return value is based on PCR attributes, and not the PCR allocation.

Return Value	Meaning
TRUE(1)	extend is allowed
FALSE(0)	extend is not allowed

```
551 BOOL
552 PCRIsExtendAllowed(
553     TPMI_DH_PCR      handle         // IN: PCR handle to be extended
554     )
555 {
556     UINT8                commandLocality;
```

```
557     UINT8               localityBits = 1;
558     UINT32              pcr = handle - PCR_FIRST;
559
560     // Check for the locality
561     commandLocality = _plat__LocalityGet();
562     localityBits = localityBits << commandLocality;
563     if((localityBits & s_initAttributes[pcr].extendLocality) == 0)
564         return FALSE;
565     else
566         return TRUE;
567 }
```

8.7.2.20 PCRExtend()

This function is used to extend a PCR in a specific bank.

```
568 void
569 PCRExtend(
570     TPMI_DH_PCR      handle,        // IN: PCR handle to be extended
571     TPMI_ALG_HASH    hash,          // IN: hash algorithm of PCR
572     UINT32           size,          // IN: size of data to be extended
573     BYTE            *data           // IN: data to be extended
574     )
575 {
576     BYTE                *pcrData;
577     HASH_STATE           hashState;
578     UINT16               pcrSize;
579
580     pcrData = GetPcrPointer(hash, handle - PCR_FIRST);
581
582     // Extend PCR if it is allocated
583     if(pcrData != NULL)
584     {
585         pcrSize = CryptHashGetDigestSize(hash);
586         CryptHashStart(&hashState, hash);
587         CryptDigestUpdate(&hashState, pcrSize, pcrData);
588         CryptDigestUpdate(&hashState, size, data);
589         CryptHashEnd(&hashState, pcrSize, pcrData);
590
591         // PCR has changed so update the pcrCounter if necessary
592         PCRChanged(handle);
593     }
594
595     return;
596 }
```

8.7.2.21 PCRComputeCurrentDigest()

This function computes the digest of the selected PCR.

As a side-effect, *selection* is modified so that only the implemented PCR will have their bits still set.

```
597 void
598 PCRComputeCurrentDigest(
599     TPMI_ALG_HASH        hashAlg,       // IN: hash algorithm to compute digest
600     TPML_PCR_SELECTION  *selection,     // IN/OUT: PCR selection (filtered on
601                                         //     output)
602     TPM2B_DIGEST        *digest         // OUT: digest
603     )
604 {
605     HASH_STATE                hashState;
606     TPMS_PCR_SELECTION       *select;
607     BYTE                     *pcrData;   // will point to a digest
```

```
608     UINT32                  pcrSize;
609     UINT32                  pcr;
610     UINT32                  i;
611
612     // Initialize the hash
613     digest->t.size = CryptHashStart(&hashState, hashAlg);
614     pAssert(digest->t.size > 0 && digest->t.size < UINT16_MAX);
615
616     // Iterate through the list of PCR selection structures
617     for(i = 0; i < selection->count; i++)
618     {
619         // Point to the current selection
620         select = &selection->pcrSelections[i]; // Point to the current selection
621         FilterPcr(select);      // Clear out the bits for unimplemented PCR
622
623         // Need the size of each digest
624         pcrSize = CryptHashGetDigestSize(selection->pcrSelections[i].hash);
625
626         // Iterate through the selection
627         for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
628         {
629             if(IsPcrSelected(pcr, select))         // Is this PCR selected
630             {
631                 // Get pointer to the digest data for the bank
632                 pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
633                 pAssert(pcrData != NULL);
634                 CryptDigestUpdate(&hashState, pcrSize, pcrData);  // add to digest
635             }
636         }
637     }
638     // Complete hash stack
639     CryptHashEnd2B(&hashState, &digest->b);
640
641     return;
642 }
```

8.7.2.22 PCRRead()

This function is used to read a list of selected PCR. If the requested PCR number exceeds the maximum number that can be output, the *selection* is adjusted to reflect the actual output PCR.

```
643 void
644 PCRRead(
645     TPML_PCR_SELECTION  *selection,     // IN/OUT: PCR selection (filtered on
646                                         //     output)
647     TPML_DIGEST         *digest,        // OUT: digest
648     UINT32              *pcrCounter     // OUT: the current value of PCR generation
649                                         //     number
650     )
651 {
652     TPMS_PCR_SELECTION      *select;
653     BYTE                    *pcrData;       // will point to a digest
654     UINT32                   pcr;
655     UINT32                   i;
656
657     digest->count = 0;
658
659     // Iterate through the list of PCR selection structures
660     for(i = 0; i < selection->count; i++)
661     {
662         // Point to the current selection
663         select = &selection->pcrSelections[i]; // Point to the current selection
664         FilterPcr(select);      // Clear out the bits for unimplemented PCR
665
```

```
666             // Iterate through the selection
667             for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
668             {
669                 if(IsPcrSelected(pcr, select))          // Is this PCR selected
670                 {
671                     // Check if number of digest exceed upper bound
672                     if(digest->count > 7)
673                     {
674                         // Clear rest of the current select bitmap
675                         while(pcr < IMPLEMENTATION_PCR
676                                   // do not round up!
677                               && (pcr / 8) < select->sizeofSelect)
678                         {
679                             // do not round up!
680                             select->pcrSelect[pcr / 8] &= (BYTE)~(1 << (pcr % 8));
681                             pcr++;
682                         }
683                         // Exit inner loop
684                         break;
685                     }
686                     // Need the size of each digest
687                     digest->digests[digest->count].t.size =
688                         CryptHashGetDigestSize(selection->pcrSelections[i].hash);
689
690                     // Get pointer to the digest data for the bank
691                     pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
692                     pAssert(pcrData != NULL);
693                     // Add to the data to digest
694                     MemoryCopy(digest->digests[digest->count].t.buffer,
695                                pcrData,
696                                digest->digests[digest->count].t.size);
697                     digest->count++;
698                 }
699             }
700             // If we exit inner loop because we have exceed the output upper bound
701             if(digest->count > 7 && pcr < IMPLEMENTATION_PCR)
702             {
703                 // Clear rest of the selection
704                 while(i < selection->count)
705                 {
706                     MemorySet(selection->pcrSelections[i].pcrSelect, 0,
707                               selection->pcrSelections[i].sizeofSelect);
708                     i++;
709                 }
710                 // exit outer loop
711                 break;
712             }
713         }
714
715         *pcrCounter = gr.pcrCounter;
716
717         return;
718     }
```

8.7.2.23 PCRAllocate()

This function is used to change the PCR allocation.

Error Returns	Meaning
TPM_RC_NO_RESULT	allocate failed
TPM_RC_PCR	improper allocation

```
719   TPM_RC
720   PCRAllocate(
721       TPML_PCR_SELECTION  *allocate,      // IN: required allocation
722       UINT32              *maxPCR,        // OUT: Maximum number of PCR
723       UINT32              *sizeNeeded,    // OUT: required space
724       UINT32              *sizeAvailable  // OUT: available space
725       )
726   {
727       UINT32                  i, j, k;
728       TPML_PCR_SELECTION      newAllocate;
729       // Initialize the flags to indicate if HCRTM PCR and DRTM PCR are allocated.
730       BOOL                    pcrHcrtm = FALSE;
731       BOOL                    pcrDrtm = FALSE;
732
733       // Create the expected new PCR allocation based on the existing allocation
734       // and the new input:
735       //  1. if a PCR bank does not appear in the new allocation, the existing
736       //     allocation of this PCR bank will be preserved.
737       //  2. if a PCR bank appears multiple times in the new allocation, only the
738       //     last one will be in effect.
739       newAllocate = gp.pcrAllocated;
740       for(i = 0; i < allocate->count; i++)
741       {
742           for(j = 0; j < newAllocate.count; j++)
743           {
744               // If hash matches, the new allocation covers the old allocation
745               // for this particular bank.
746               // The assumption is the initial PCR allocation (from manufacture)
747               // has all the supported hash algorithms with an assigned bank
748               // (possibly empty).  So there must be a match for any new bank
749               // allocation from the input.
750               if(newAllocate.pcrSelections[j].hash ==
751                  allocate->pcrSelections[i].hash)
752               {
753                   newAllocate.pcrSelections[j] = allocate->pcrSelections[i];
754                   break;
755               }
756           }
757           // The j loop must exit with a match.
758           pAssert(j < newAllocate.count);
759       }
760
761       // Max PCR in a bank is MIN(implemented PCR, PCR with attributes defined)
762       *maxPCR = sizeof(s_initAttributes) / sizeof(PCR_Attributes);
763       if(*maxPCR > IMPLEMENTATION_PCR)
764           *maxPCR = IMPLEMENTATION_PCR;
765
766       // Compute required size for allocation
767       *sizeNeeded = 0;
768       for(i = 0; i < newAllocate.count; i++)
769       {
770           UINT32      digestSize
771               = CryptHashGetDigestSize(newAllocate.pcrSelections[i].hash);
772   #if defined(DRTM_PCR)
773           // Make sure that we end up with at least one DRTM PCR
774           pcrDrtm = pcrDrtm || TestBit(DRTM_PCR,
775                                        newAllocate.pcrSelections[i].pcrSelect,
776                                        newAllocate.pcrSelections[i].sizeofSelect);
777
```

```
778  #else   // if DRTM PCR is not required, indicate that the allocation is OK
779          pcrDrtm = TRUE;
780  #endif
781
782  #if defined(HCRTM_PCR)
783          // and one HCRTM PCR (since this is usually PCR 0...)
784          pcrHcrtm = pcrHcrtm || TestBit(HCRTM_PCR,
785                                         newAllocate.pcrSelections[i].pcrSelect,
786                                         newAllocate.pcrSelections[i].sizeofSelect);
787  #else
788          pcrHcrtm = TRUE;
789  #endif
790          for(j = 0; j < newAllocate.pcrSelections[i].sizeofSelect; j++)
791          {
792              BYTE         mask = 1;
793              for(k = 0; k < 8; k++)
794              {
795                  if((newAllocate.pcrSelections[i].pcrSelect[j] & mask) != 0)
796                      *sizeNeeded += digestSize;
797                  mask = mask << 1;
798              }
799          }
800      }
801
802      if(!pcrDrtm || !pcrHcrtm)
803          return TPM_RC_PCR;
804
805      // In this particular implementation, we always have enough space to
806      // allocate PCR.  Different implementation may return a sizeAvailable less
807      // than the sizeNeed.
808      *sizeAvailable = sizeof(s_pcrs);
809
810      // Save the required allocation to NV.  Note that after NV is written, the
811      // PCR allocation in NV is no longer consistent with the RAM data
812      // gp.pcrAllocated.  The NV version reflect the allocate after next
813      // TPM_RESET, while the RAM version reflects the current allocation
814      NV_WRITE_PERSISTENT(pcrAllocated, newAllocate);
815
816      return TPM_RC_SUCCESS;
817  }
```

8.7.2.24 PCRSetValue()

This function is used to set the designated PCR in all banks to an initial value. The initial value is signed and will be sign extended into the entire PCR.

```
818  void
819  PCRSetValue(
820      TPM_HANDLE       handle,        // IN: the handle of the PCR to set
821      INT8             initialValue   // IN: the value to set
822      )
823  {
824      int              i;
825      UINT32           pcr = handle - PCR_FIRST;
826      TPMI_ALG_HASH    hash;
827      UINT16           digestSize;
828      BYTE            *pcrData;
829
830      // Iterate supported PCR bank algorithms to reset
831      for(i = 0; i < HASH_COUNT; i++)
832      {
833          hash = CryptHashGetAlgByIndex(i);
834          // Prevent runaway
835          if(hash == TPM_ALG_NULL)
```

```
836                 break;
837
838             // Get a pointer to the data
839             pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
840
841             // If the PCR is allocated
842             if(pcrData != NULL)
843             {
844                 // And the size of the digest
845                 digestSize = CryptHashGetDigestSize(hash);
846
847                 // Set the LSO to the input value
848                 pcrData[digestSize - 1] = initialValue;
849
850                 // Sign extend
851                 if(initialValue >= 0)
852                     MemorySet(pcrData, 0, digestSize - 1);
853                 else
854                     MemorySet(pcrData, -1, digestSize - 1);
855             }
856         }
857 }
```

8.7.2.25 PCRResetDynamics

This function is used to reset a dynamic PCR to 0. This function is used in DRTM sequence.

```
858 void
859 PCRResetDynamics(
860     void
861     )
862 {
863     UINT32              pcr, i;
864
865     // Initialize PCR values
866     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
867     {
868         // Iterate each hash algorithm bank
869         for(i = 0; i < gp.pcrAllocated.count; i++)
870         {
871             BYTE    *pcrData;
872             UINT32  pcrSize;
873
874             pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
875
876             if(pcrData != NULL)
877             {
878                 pcrSize =
879                     CryptHashGetDigestSize(gp.pcrAllocated.pcrSelections[i].hash);
880
881                 // Reset PCR
882                 // Any PCR can be reset by locality 4 should be reset to 0
883                 if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
884                     MemorySet(pcrData, 0, pcrSize);
885             }
886         }
887     }
888     return;
889 }
```

8.7.2.26 PCRCapGetAllocation()

This function is used to get the current allocation of PCR banks.

Return Value	Meaning
YES	if the return count is 0
NO	if the return count is not 0

```
890  TPMI_YES_NO
891  PCRCapGetAllocation(
892      UINT32                 count,          // IN: count of return
893      TPML_PCR_SELECTION    *pcrSelection    // OUT: PCR allocation list
894      )
895  {
896      if(count == 0)
897      {
898          pcrSelection->count = 0;
899          return YES;
900      }
901      else
902      {
903          *pcrSelection = gp.pcrAllocated;
904          return NO;
905      }
906  }
```

8.7.2.27 PCRSetSelectBit()

This function sets a bit in a bitmap array.

```
907  static void
908  PCRSetSelectBit(
909      UINT32              pcr,            // IN: PCR number
910      BYTE               *bitmap          // OUT: bit map to be set
911      )
912  {
913      bitmap[pcr / 8] |= (1 << (pcr % 8));
914      return;
915  }
```

8.7.2.28 PCRGetProperty()

This function returns the selected PCR property.

Return Value	Meaning
TRUE(1)	the property type is implemented
FALSE(0)	the property type is not implemented

```
916  static BOOL
917  PCRGetProperty(
918      TPM_PT_PCR                 property,
919      TPMS_TAGGED_PCR_SELECT    *select
920      )
921  {
922      UINT32                  pcr;
923      UINT32                  groupIndex;
924
925      select->tag = property;
926      // Always set the bitmap to be the size of all PCR
927      select->sizeofSelect = (IMPLEMENTATION_PCR + 7) / 8;
928
929      // Initialize bitmap
```

 Copyright © TCG 2006-2020

```
930         MemorySet(select->pcrSelect, 0, select->sizeofSelect);
931
932         // Collecting properties
933         for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
934         {
935             switch(property)
936             {
937                 case TPM_PT_PCR_SAVE:
938                     if(s_initAttributes[pcr].stateSave == SET)
939                         PCRSetSelectBit(pcr, select->pcrSelect);
940                     break;
941                 case TPM_PT_PCR_EXTEND_L0:
942                     if((s_initAttributes[pcr].extendLocality & 0x01) != 0)
943                         PCRSetSelectBit(pcr, select->pcrSelect);
944                     break;
945                 case TPM_PT_PCR_RESET_L0:
946                     if((s_initAttributes[pcr].resetLocality & 0x01) != 0)
947                         PCRSetSelectBit(pcr, select->pcrSelect);
948                     break;
949                 case TPM_PT_PCR_EXTEND_L1:
950                     if((s_initAttributes[pcr].extendLocality & 0x02) != 0)
951                         PCRSetSelectBit(pcr, select->pcrSelect);
952                     break;
953                 case TPM_PT_PCR_RESET_L1:
954                     if((s_initAttributes[pcr].resetLocality & 0x02) != 0)
955                         PCRSetSelectBit(pcr, select->pcrSelect);
956                     break;
957                 case TPM_PT_PCR_EXTEND_L2:
958                     if((s_initAttributes[pcr].extendLocality & 0x04) != 0)
959                         PCRSetSelectBit(pcr, select->pcrSelect);
960                     break;
961                 case TPM_PT_PCR_RESET_L2:
962                     if((s_initAttributes[pcr].resetLocality & 0x04) != 0)
963                         PCRSetSelectBit(pcr, select->pcrSelect);
964                     break;
965                 case TPM_PT_PCR_EXTEND_L3:
966                     if((s_initAttributes[pcr].extendLocality & 0x08) != 0)
967                         PCRSetSelectBit(pcr, select->pcrSelect);
968                     break;
969                 case TPM_PT_PCR_RESET_L3:
970                     if((s_initAttributes[pcr].resetLocality & 0x08) != 0)
971                         PCRSetSelectBit(pcr, select->pcrSelect);
972                     break;
973                 case TPM_PT_PCR_EXTEND_L4:
974                     if((s_initAttributes[pcr].extendLocality & 0x10) != 0)
975                         PCRSetSelectBit(pcr, select->pcrSelect);
976                     break;
977                 case TPM_PT_PCR_RESET_L4:
978                     if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
979                         PCRSetSelectBit(pcr, select->pcrSelect);
980                     break;
981                 case TPM_PT_PCR_DRTM_RESET:
982                     // DRTM reset PCRs are the PCR reset by locality 4
983                     if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
984                         PCRSetSelectBit(pcr, select->pcrSelect);
985                     break;
986 #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
987                 case TPM_PT_PCR_POLICY:
988                     if(PCRBelongsPolicyGroup(pcr + PCR_FIRST, &groupIndex))
989                         PCRSetSelectBit(pcr, select->pcrSelect);
990                     break;
991 #endif
992 #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
993                 case TPM_PT_PCR_AUTH:
994                     if(PCRBelongsAuthGroup(pcr + PCR_FIRST, &groupIndex))
995                         PCRSetSelectBit(pcr, select->pcrSelect);
```

```
996                     break;
997     #endif
998     #if ENABLE_PCR_NO_INCREMENT == YES
999                 case TPM_PT_PCR_NO_INCREMENT:
1000                    if(PCRBelongsTCBGroup(pcr + PCR_FIRST))
1001                        PCRSetSelectBit(pcr, select->pcrSelect);
1002                    break;
1003    #endif
1004                default:
1005                    // If property is not supported, stop scanning PCR attributes
1006                    // and return.
1007                    return FALSE;
1008                    break;
1009            }
1010        }
1011        return TRUE;
1012    }
```

8.7.2.29 PCRCapGetProperties()

This function returns a list of PCR properties starting at *property*.

Return Value	Meaning
YES	if no more property is available
NO	if there are more properties not reported

```
1013    TPMI_YES_NO
1014    PCRCapGetProperties(
1015        TPM_PT_PCR                  property,       // IN: the starting PCR property
1016        UINT32                      count,          // IN: count of returned properties
1017        TPML_TAGGED_PCR_PROPERTY    *select         // OUT: PCR select
1018        )
1019    {
1020        TPMI_YES_NO     more = NO;
1021        UINT32          i;
1022
1023        // Initialize output property list
1024        select->count = 0;
1025
1026        // The maximum count of properties we may return is MAX_PCR_PROPERTIES
1027        if(count > MAX_PCR_PROPERTIES) count = MAX_PCR_PROPERTIES;
1028
1029        // TPM_PT_PCR_FIRST is defined as 0 in spec.  It ensures that property
1030        // value would never be less than TPM_PT_PCR_FIRST
1031        cAssert(TPM_PT_PCR_FIRST == 0);
1032
1033        // Iterate PCR properties. TPM_PT_PCR_LAST is the index of the last property
1034        // implemented on the TPM.
1035        for(i = property; i <= TPM_PT_PCR_LAST; i++)
1036        {
1037            if(select->count < count)
1038            {
1039                // If we have not filled up the return list, add more properties to it
1040                if(PCRGetProperty(i, &select->pcrProperty[select->count]))
1041                    // only increment if the property is implemented
1042                    select->count++;
1043            }
1044            else
1045            {
1046                // If the return list is full but we still have properties
1047                // available, report this and stop iterating.
1048                more = YES;
```

 Copyright © TCG 2006-2020

```
1049                break;
1050            }
1051        }
1052        return more;
1053    }
```

8.7.2.30 PCRCapGetHandles()

This function is used to get a list of handles of PCR, started from *handle*. If *handle* exceeds the maximum PCR handle range, an empty list will be returned and the return value will be NO.

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
1054    TPMI_YES_NO
1055    PCRCapGetHandles(
1056        TPMI_DH_PCR      handle,        // IN: start handle
1057        UINT32           count,         // IN: count of returned handles
1058        TPML_HANDLE     *handleList     // OUT: list of handle
1059        )
1060    {
1061        TPMI_YES_NO     more = NO;
1062        UINT32          i;
1063
1064        pAssert(HandleGetType(handle) == TPM_HT_PCR);
1065
1066        // Initialize output handle list
1067        handleList->count = 0;
1068
1069        // The maximum count of handles we may return is MAX_CAP_HANDLES
1070        if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1071
1072        // Iterate PCR handle range
1073        for(i = handle & HR_HANDLE_MASK; i <= PCR_LAST; i++)
1074        {
1075            if(handleList->count < count)
1076            {
1077                // If we have not filled up the return list, add this PCR
1078                // handle to it
1079                handleList->handle[handleList->count] = i + PCR_FIRST;
1080                handleList->count++;
1081            }
1082            else
1083            {
1084                // If the return list is full but we still have PCR handle
1085                // available, report this and stop iterating
1086                more = YES;
1087                break;
1088            }
1089        }
1090        return more;
1091    }
```

 Copyright © TCG 2006-2020

8.8 PP.c

8.8.1 Introduction

This file contains the functions that support the physical presence operations of the TPM.

8.8.2 Includes

```
1   #include "Tpm.h"
```

8.8.3 Functions

8.8.3.1 PhysicalPresencePreInstall_Init()

This function is used to initialize the array of commands that always require confirmation with physical presence. The array is an array of bits that has a correspondence with the command code.

This command should only ever be executable in a manufacturing setting or in a simulation.

When set, these cannot be cleared.

```
2   void
3   PhysicalPresencePreInstall_Init(
4       void
5       )
6   {
7       COMMAND_INDEX        commandIndex;
8       // Clear all the PP commands
9       MemorySet(&gp.ppList, 0, sizeof(gp.ppList));
10
11      // Any command that is PP_REQUIRED should be SET
12      for(commandIndex = 0; commandIndex < COMMAND_COUNT; commandIndex++)
13      {
14          if(s_commandAttributes[commandIndex] & IS_IMPLEMENTED
15             &&  s_commandAttributes[commandIndex] & PP_REQUIRED)
16              SET_BIT(commandIndex, gp.ppList);
17      }
18      // Write PP list to NV
19      NV_SYNC_PERSISTENT(ppList);
20      return;
21  }
```

8.8.3.2 PhysicalPresenceCommandSet()

This function is used to set the indicator that a command requires PP confirmation.

```
22  void
23  PhysicalPresenceCommandSet(
24      TPM_CC           commandCode     // IN: command code
25      )
26  {
27      COMMAND_INDEX       commandIndex = CommandCodeToCommandIndex(commandCode);
28
29      // if the command isn't implemented, the do nothing
30      if(commandIndex == UNIMPLEMENTED_COMMAND_INDEX)
31          return;
32
33      // only set the bit if this is a command for which PP is allowed
34      if(s_commandAttributes[commandIndex] & PP_COMMAND)
```

```
35              SET_BIT(commandIndex, gp.ppList);
36          return;
37      }
```

8.8.3.3 PhysicalPresenceCommandClear()

This function is used to clear the indicator that a command requires PP confirmation.

```
38      void
39      PhysicalPresenceCommandClear(
40          TPM_CC              commandCode     // IN: command code
41          )
42      {
43          COMMAND_INDEX         commandIndex = CommandCodeToCommandIndex(commandCode);
44
45          // If the command isn't implemented, then don't do anything
46          if(commandIndex == UNIMPLEMENTED_COMMAND_INDEX)
47              return;
48
49          // Only clear the bit if the command does not require PP
50          if((s_commandAttributes[commandIndex] & PP_REQUIRED) == 0)
51              CLEAR_BIT(commandIndex, gp.ppList);
52
53          return;
54      }
```

8.8.3.4 PhysicalPresenceIsRequired()

This function indicates if PP confirmation is required for a command.

Return Value	Meaning
TRUE(1)	physical presence is required
FALSE(0)	physical presence is not required

```
55      BOOL
56      PhysicalPresenceIsRequired(
57          COMMAND_INDEX       commandIndex    // IN: command index
58          )
59      {
60          // Check the bit map.  If the bit is SET, PP authorization is required
61          return (TEST_BIT(commandIndex, gp.ppList));
62      }
```

8.8.3.5 PhysicalPresenceCapGetCCList()

This function returns a list of commands that require PP confirmation. The list starts from the first implemented command that has a command code that the same or greater than *commandCode*.

Return Value	Meaning
YES	if there are more command codes available
NO	all the available command codes have been returned

```
63      TPMI_YES_NO
64      PhysicalPresenceCapGetCCList(
65          TPM_CC              commandCode,    // IN: start command code
66          UINT32              count,          // IN: count of returned TPM_CC
67          TPML_CC            *commandList     // OUT: list of TPM_CC
```

 Copyright © TCG 2006-2020

```
68      )
69  {
70      TPMI_YES_NO     more = NO;
71      COMMAND_INDEX   commandIndex;
72
73      // Initialize output handle list
74      commandList->count = 0;
75
76      // The maximum count of command we may return is MAX_CAP_CC
77      if(count > MAX_CAP_CC) count = MAX_CAP_CC;
78
79      // Collect PP commands
80      for(commandIndex = GetClosestCommandIndex(commandCode);
81      commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
82          commandIndex = GetNextCommandIndex(commandIndex))
83      {
84          if(PhysicalPresenceIsRequired(commandIndex))
85          {
86              if(commandList->count < count)
87              {
88                  // If we have not filled up the return list, add this command
89                  // code to it
90                  commandList->commandCodes[commandList->count]
91                      = GetCommandCode(commandIndex);
92                  commandList->count++;
93              }
94              else
95              {
96                  // If the return list is full but we still have PP command
97                  // available, report this and stop iterating
98                  more = YES;
99                  break;
100             }
101         }
102     }
103     return more;
104 }
```

 Copyright © TCG 2006-2020

8.9 Session.c

8.9.1 Introduction

The code in this file is used to manage the session context counter. The scheme implemented here is a "truncated counter". This scheme allows the TPM to not need TPM_SU_CLEAR for a very long period of time and still not have the context count for a session repeated.

The counter (*contextCounter*)in this implementation is a UINT64 but can be smaller. The "tracking array" (*contextArray*) only has 16-bits per context. The tracking array is the data that needs to be saved and restored across TPM_SU_STATE so that sessions are not lost when the system enters the sleep state. Also, when the TPM is active, the tracking array is kept in RAM making it important that the number of bytes for each entry be kept as small as possible.

The TPM prevents **collisions** of these truncated values by not allowing a *contextID* to be assigned if it would be the same as an existing value. Since the array holds 16 bits, after a context has been saved, an additional 2^16-1 contexts may be saved before the count would again match. The normal expectation is that the context will be flushed before its count value is needed again but it is always possible to have long-lived sessions.

The *contextID* is assigned when the context is saved (TPM2_ContextSave()). At that time, the TPM will compare the low-order 16 bits of *contextCounter* to the existing values in *contextArray* and if one matches, the TPM will return TPM_RC_CONTEXT_GAP (by construction, the entry that contains the matching value is the oldest context).

The expected remediation by the TRM is to load the oldest saved session context (the one found by the TPM), and save it. Since loading the oldest session also eliminates its *contextID* value from *contextArray*, there TPM will always be able to load and save the oldest existing context.

In the worst case, software may have to load and save several contexts in order to save an additional one. This should happen very infrequently.

When the TPM searches *contextArray* and finds that none of the *contextIDs* match the low-order 16-bits of *contextCount*, the TPM can copy the low bits to the *contextArray* associated with the session, and increment *contextCount*.

There is one entry in *contextArray* for each of the active sessions allowed by the TPM implementation. This array contains either a context count, an index, or a value indicating the slot is available (0).

The index into the *contextArray* is the handle for the session with the region selector byte of the session set to zero. If an entry in *contextArray* contains 0, then the corresponding handle may be assigned to a session. If the entry contains a value that is less than or equal to the number of loaded sessions for the TPM, then the array entry is the slot in which the context is loaded.

EXAMPLE: If the TPM allows 8 loaded sessions, then the slot numbers would be 1-8 and a *contextArrary* value in that range would represent the loaded session.

NOTE: When the TPM firmware determines that the array entry is for a loaded session, it will subtract 1 to create the zero-based slot number.

There is one significant corner case in this scheme. When the *contextCount* is equal to a value in the *contextArray*, the oldest session needs to be recycled or flushed. In order to recycle the session, it must be loaded. To be loaded, there must be an available slot. Rather than require that a spare slot be available all the time, the TPM will check to see if the *contextCount* is equal to some value in the *contextArray* when a session is created. This prevents the last session slot from being used when it is likely that a session will need to be recycled.

If a TPM with both 1.2 and 2.0 functionality uses this scheme for both 1.2 and 2.0 sessions, and the list of active contexts is read with TPM_GetCapabiltiy(), the TPM will create 32-bit representations of the list that contains 16-bit values (the TPM2_GetCapability() returns a list of handles for active sessions rather than

 Copyright © TCG 2006-2020

a list of *contextID*). The full *contextID* has high-order bits that are either the same as the current *contextCount* or one less. It is one less if the 16-bits of the *contextArray* has a value that is larger than the low-order 16 bits of *contextCount*.

8.9.2 Includes, Defines, and Local Variables

```
1   #define SESSION_C
2   #include "Tpm.h"
```

8.9.3 File Scope Function -- ContextIdSetOldest()

This function is called when the oldest *contextID* is being loaded or deleted. Once a saved context becomes the oldest, it stays the oldest until it is deleted.

Finding the oldest is a bit tricky. It is not just the numeric comparison of values but is dependent on the value of *contextCounter*.

Assume we have a small *contextArray* with 8, 4-bit values with values 1 and 2 used to indicate the loaded context slot number. Also assume that the array contains hex values of (0 0 1 0 3 0 9 F) and that the *contextCounter* is an 8-bit counter with a value of 0x37. Since the low nibble is 7, that means that values closest to but above 7 are older than values below it and, in this example, 9 is the oldest value.

Note if we subtract the counter value, from each slot that contains a saved *contextID* we get (- - - - B - 2 - 8) and the oldest entry is now easy to find because it has the lowest value.

```
3   static void
4   ContextIdSetOldest(
5       void
6       )
7   {
8       CONTEXT_SLOT    lowBits;
9       CONTEXT_SLOT    entry;
10      CONTEXT_SLOT    smallest = ((CONTEXT_SLOT)~0);  // Set to the maximum possible
11      UINT32  i;
12
13      // Set oldestSaveContext to a value indicating none assigned
14      s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;
15
16      lowBits = (CONTEXT_SLOT)gr.contextCounter;
17      for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
18      {
19          entry = gr.contextArray[i];
20
21          // only look at entries that are saved contexts
22          if(entry > MAX_LOADED_SESSIONS)
23          {
24              // Use a less than or equal in case the oldest
25              // is brand new (= lowBits-1) and equal to our initial
26              // value for smallest.
27              if(((CONTEXT_SLOT)(entry - lowBits)) <= smallest)
28              {
29                  smallest = (entry - lowBits);
30                  s_oldestSavedSession = i;
31              }
32          }
33      }
34      // When we finish, either the s_oldestSavedSession still has its initial
35      // value, or it has the index of the oldest saved context.
36  }
```

8.9.4 Startup Function -- SessionStartup()

This function initializes the session subsystem on TPM2_Startup().

```
37  BOOL
38  SessionStartup(
39      STARTUP_TYPE     type
40      )
41  {
42      UINT32              i;
43
44      // Initialize session slots.  At startup, all the in-memory session slots
45      // are cleared and marked as not occupied
46      for(i = 0; i < MAX_LOADED_SESSIONS; i++)
47          s_sessions[i].occupied = FALSE;   // session slot is not occupied
48
49      // The free session slots the number of maximum allowed loaded sessions
50      s_freeSessionSlots = MAX_LOADED_SESSIONS;
51
52      // Initialize context ID data.  On a ST_SAVE or hibernate sequence, it will
53      // scan the saved array of session context counts, and clear any entry that
54      // references a session that was in memory during the state save since that
55      // memory was not preserved over the ST_SAVE.
56      if(type == SU_RESUME || type == SU_RESTART)
57      {
58          // On ST_SAVE we preserve the contexts that were saved but not the ones
59          // in memory
60          for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
61          {
62              // If the array value is unused or references a loaded session then
63              // that loaded session context is lost and the array entry is
64              // reclaimed.
65              if(gr.contextArray[i] <= MAX_LOADED_SESSIONS)
66                  gr.contextArray[i] = 0;
67          }
68          // Find the oldest session in context ID data and set it in
69          // s_oldestSavedSession
70          ContextIdSetOldest();
71      }
72      else
73      {
74          // For STARTUP_CLEAR, clear out the contextArray
75          for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
76              gr.contextArray[i] = 0;
77
78          // reset the context counter
79          gr.contextCounter = MAX_LOADED_SESSIONS + 1;
80
81          // Initialize oldest saved session
82          s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;
83      }
84      return TRUE;
85  }
```

8.9.5 Access Functions

8.9.5.1 SessionIsLoaded()

This function test a session handle references a loaded session. The handle must have previously been checked to make sure that it is a valid handle for an authorization session.

 Copyright © TCG 2006-2020

NOTE: A PWAP authorization does not have a session.

Return Value	Meaning
TRUE(1)	session is loaded
FALSE(0)	session is not loaded

```
 86 BOOL
 87 SessionIsLoaded(
 88     TPM_HANDLE       handle          // IN: session handle
 89     )
 90 {
 91     pAssert(HandleGetType(handle) == TPM_HT_POLICY_SESSION
 92             || HandleGetType(handle) == TPM_HT_HMAC_SESSION);
 93
 94     handle = handle & HR_HANDLE_MASK;
 95
 96     // if out of range of possible active session, or not assigned to a loaded
 97     // session return false
 98     if(handle >= MAX_ACTIVE_SESSIONS
 99        || gr.contextArray[handle] == 0
100        || gr.contextArray[handle] > MAX_LOADED_SESSIONS)
101         return FALSE;
102
103     return TRUE;
104 }
```

8.9.5.2 SessionIsSaved()

This function test a session handle references a saved session. The handle must have previously been checked to make sure that it is a valid handle for an authorization session.

NOTE: A password authorization does not have a session.

This function requires that the handle be a valid session handle.

Return Value	Meaning
TRUE(1)	session is saved
FALSE(0)	session is not saved

```
105 BOOL
106 SessionIsSaved(
107     TPM_HANDLE       handle          // IN: session handle
108     )
109 {
110     pAssert(HandleGetType(handle) == TPM_HT_POLICY_SESSION
111             || HandleGetType(handle) == TPM_HT_HMAC_SESSION);
112
113     handle = handle & HR_HANDLE_MASK;
114     // if out of range of possible active session, or not assigned, or
115     // assigned to a loaded session, return false
116     if(handle >= MAX_ACTIVE_SESSIONS
117        || gr.contextArray[handle] == 0
118        || gr.contextArray[handle] <= MAX_LOADED_SESSIONS
119        )
120         return FALSE;
121
122     return TRUE;
123 }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

8.9.5.3 SequenceNumberForSavedContextIsValid()

This function validates that the sequence number and handle value within a saved context are valid.

```
124 BOOL
125 SequenceNumberForSavedContextIsValid(
126     TPMS_CONTEXT    *context        // IN: pointer to a context structure to be
127                                     //     validated
128     )
129 {
130 #define MAX_CONTEXT_GAP ((UINT64)((CONTEXT_SLOT) ~0) + 1)
131
132     TPM_HANDLE           handle = context->savedHandle & HR_HANDLE_MASK;
133
134     if(// Handle must be with the range of active sessions
135        handle >= MAX_ACTIVE_SESSIONS
136        // the array entry must be for a saved context
137        || gr.contextArray[handle] <= MAX_LOADED_SESSIONS
138           // the array entry must agree with the sequence number
139        || gr.contextArray[handle] != (CONTEXT_SLOT)context->sequence
140           // the provided sequence number has to be less than the current counter
141        || context->sequence > gr.contextCounter
142           // but not so much that it could not be a valid sequence number
143        || gr.contextCounter - context->sequence > MAX_CONTEXT_GAP)
144         return FALSE;
145
146     return TRUE;
147 }
```

8.9.5.4 SessionPCRValueIsCurrent()

This function is used to check if PCR values have been updated since the last time they were checked in a policy session.

This function requires the session is loaded.

Return Value	Meaning
TRUE(1)	PCR value is current
FALSE(0)	PCR value is not current

```
148 BOOL
149 SessionPCRValueIsCurrent(
150     SESSION         *session        // IN: session structure
151     )
152 {
153     if(session->pcrCounter != 0
154        && session->pcrCounter != gr.pcrCounter
155        )
156         return FALSE;
157     else
158         return TRUE;
159 }
```

8.9.5.5 SessionGet()

This function returns a pointer to the session object associated with a session handle.

The function requires that the session is loaded.

```
160 SESSION *
```

 Copyright © TCG 2006-2020

```
161     SessionGet(
162         TPM_HANDLE          handle          // IN: session handle
163         )
164     {
165         size_t          slotIndex;
166         CONTEXT_SLOT    sessionIndex;
167
168         pAssert(HandleGetType(handle) == TPM_HT_POLICY_SESSION
169                 || HandleGetType(handle) == TPM_HT_HMAC_SESSION
170                 );
171
172         slotIndex = handle & HR_HANDLE_MASK;
173
174         pAssert(slotIndex < MAX_ACTIVE_SESSIONS);
175
176         // get the contents of the session array.  Because session is loaded, we
177         // should always get a valid sessionIndex
178         sessionIndex = gr.contextArray[slotIndex] - 1;
179
180         pAssert(sessionIndex < MAX_LOADED_SESSIONS);
181
182         return &s_sessions[sessionIndex].session;
183     }
```

8.9.6 Utility Functions

8.9.6.1 ContextIdSessionCreate()

This function is called when a session is created. It will check to see if the current gap would prevent a context from being saved. If so it will return TPM_RC_CONTEXT_GAP. Otherwise, it will try to find an open slot in *contextArray*, set *contextArray* to the slot.

This routine requires that the caller has determined the session array index for the session.

Error Returns	Meaning
TPM_RC_CONTEXT_GAP	can't assign a new *contextID* until the oldest saved session context is recycled
TPM_RC_SESSION_HANDLE	there is no slot available in the context array for tracking of this session context

```
184     static TPM_RC
185     ContextIdSessionCreate(
186         TPM_HANDLE      *handle,         // OUT: receives the assigned handle. This will
187                                          //     be an index that must be adjusted by the
188                                          //     caller according to the type of the
189                                          //     session created
190         UINT32           sessionIndex    // IN: The session context array entry that will
191                                          //     be occupied by the created session
192         )
193     {
194         pAssert(sessionIndex < MAX_LOADED_SESSIONS);
195
196         // check to see if creating the context is safe
197         // Is this going to be an assignment for the last session context
198         // array entry?  If so, then there will be no room to recycle the
199         // oldest context if needed.  If the gap is not at maximum, then
200         // it will be possible to save a context if it becomes necessary.
201         if(s_oldestSavedSession < MAX_ACTIVE_SESSIONS
202            && s_freeSessionSlots == 1)
203         {
204             // See if the gap is at maximum
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
205          // The current value of the contextCounter will be assigned to the next
206          // saved context. If the value to be assigned would make the same as an
207          // existing context, then we can't use it because of the ambiguity it would
208          // create.
209          if((CONTEXT_SLOT)gr.contextCounter
210             == gr.contextArray[s_oldestSavedSession])
211              return TPM_RC_CONTEXT_GAP;
212      }
213
214      // Find an unoccupied entry in the contextArray
215      for(*handle = 0; *handle < MAX_ACTIVE_SESSIONS; (*handle)++)
216      {
217          if(gr.contextArray[*handle] == 0)
218          {
219              // indicate that the session associated with this handle
220              // references a loaded session
221              gr.contextArray[*handle] = (CONTEXT_SLOT)(sessionIndex + 1);
222              return TPM_RC_SUCCESS;
223          }
224      }
225      return TPM_RC_SESSION_HANDLES;
226  }
```

8.9.6.2 SessionCreate()

This function does the detailed work for starting an authorization session. This is done in a support routine rather than in the action code because the session management may differ in implementations. This implementation uses a fixed memory allocation to hold sessions and a fixed allocation to hold the *contextID* for the saved contexts.

Error Returns	Meaning
TPM_RC_CONTEXT_GAP	need to recycle sessions
TPM_RC_SESSION_HANDLE	active session space is full
TPM_RC_SESSION_MEMORY	loaded session space is full

```
227  TPM_RC
228  SessionCreate(
229      TPM_SE            sessionType,    // IN: the session type
230      TPMI_ALG_HASH     authHash,       // IN: the hash algorithm
231      TPM2B_NONCE      *nonceCaller,    // IN: initial nonceCaller
232      TPMT_SYM_DEF     *symmetric,      // IN: the symmetric algorithm
233      TPMI_DH_ENTITY    bind,           // IN: the bind object
234      TPM2B_DATA       *seed,           // IN: seed data
235      TPM_HANDLE       *sessionHandle,  // OUT: the session handle
236      TPM2B_NONCE      *nonceTpm        // OUT: the session nonce
237      )
238  {
239      TPM_RC                result = TPM_RC_SUCCESS;
240      CONTEXT_SLOT          slotIndex;
241      SESSION              *session = NULL;
242
243      pAssert(sessionType == TPM_SE_HMAC
244              || sessionType == TPM_SE_POLICY
245              || sessionType == TPM_SE_TRIAL);
246
247      // If there are no open spots in the session array, then no point in searching
248      if(s_freeSessionSlots == 0)
249          return TPM_RC_SESSION_MEMORY;
250
251      // Find a space for loading a session
252      for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
```

```
253         {
254             // Is this available?
255             if(s_sessions[slotIndex].occupied == FALSE)
256             {
257                 session = &s_sessions[slotIndex].session;
258                 break;
259             }
260         }
261         // if no spot found, then this is an internal error
262         if(slotIndex >= MAX_LOADED_SESSIONS)
263             FAIL(FATAL_ERROR_INTERNAL);
264
265         // Call context ID function to get a handle.  TPM_RC_SESSION_HANDLE may be
266         // returned from ContextIdHandelAssign()
267         result = ContextIdSessionCreate(sessionHandle, slotIndex);
268         if(result != TPM_RC_SUCCESS)
269             return result;
270
271         //*** Only return from this point on is TPM_RC_SUCCESS
272
273         // Can now indicate that the session array entry is occupied.
274         s_freeSessionSlots--;
275         s_sessions[slotIndex].occupied = TRUE;
276
277         // Initialize the session data
278         MemorySet(session, 0, sizeof(SESSION));
279
280         // Initialize internal session data
281         session->authHashAlg = authHash;
282         // Initialize session type
283         if(sessionType == TPM_SE_HMAC)
284         {
285             *sessionHandle += HMAC_SESSION_FIRST;
286         }
287         else
288         {
289             *sessionHandle += POLICY_SESSION_FIRST;
290
291             // For TPM_SE_POLICY or TPM_SE_TRIAL
292             session->attributes.isPolicy = SET;
293             if(sessionType == TPM_SE_TRIAL)
294                 session->attributes.isTrialPolicy = SET;
295
296             SessionSetStartTime(session);
297
298             // Initialize policyDigest.  policyDigest is initialized with a string of 0
299             // of session algorithm digest size. Since the session is already clear.
300             // Just need to set the size
301             session->u2.policyDigest.t.size =
302                 CryptHashGetDigestSize(session->authHashAlg);
303         }
304         // Create initial session nonce
305         session->nonceTPM.t.size = nonceCaller->t.size;
306         CryptRandomGenerate(session->nonceTPM.t.size, session->nonceTPM.t.buffer);
307         MemoryCopy2B(&nonceTpm->b, &session->nonceTPM.b,
308                      sizeof(nonceTpm->t.buffer));
309
310         // Set up session parameter encryption algorithm
311         session->symmetric = *symmetric;
312
313         // If there is a bind object or a session secret, then need to compute
314         // a sessionKey.
315         if(bind != TPM_RH_NULL || seed->t.size != 0)
316         {
317             // sessionKey = KDFa(hash, (authValue || seed), "ATH", nonceTPM,
318             //                      nonceCaller, bits)
```

```
319             // The HMAC key for generating the sessionSecret can be the concatenation
320             // of an authorization value and a seed value
321             TPM2B_TYPE(KEY, (sizeof(TPMT_HA) + sizeof(seed->t.buffer)));
322             TPM2B_KEY            key;
323
324             // Get hash size, which is also the length of sessionKey
325             session->sessionKey.t.size = CryptHashGetDigestSize(session->authHashAlg);
326
327             // Get authValue of associated entity
328             EntityGetAuthValue(bind, (TPM2B_AUTH *)&key);
329             pAssert(key.t.size + seed->t.size <= sizeof(key.t.buffer));
330
331             // Concatenate authValue and seed
332             MemoryConcat2B(&key.b, &seed->b, sizeof(key.t.buffer));
333
334             // Compute the session key
335             CryptKDFa(session->authHashAlg, &key.b, SESSION_KEY, &session->nonceTPM.b,
336                       &nonceCaller->b,
337                       session->sessionKey.t.size * 8, session->sessionKey.t.buffer,
338                       NULL, FALSE);
339         }
340
341         // Copy the name of the entity that the HMAC session is bound to
342         // Policy session is not bound to an entity
343         if(bind != TPM_RH_NULL && sessionType == TPM_SE_HMAC)
344         {
345             session->attributes.isBound = SET;
346             SessionComputeBoundEntity(bind, &session->u1.boundEntity);
347         }
348         // If there is a bind object and it is subject to DA, then use of this session
349         // is subject to DA regardless of how it is used.
350         session->attributes.isDaBound = (bind != TPM_RH_NULL)
351             && (IsDAExempted(bind) == FALSE);
352
353     // If the session is bound, then check to see if it is bound to lockoutAuth
354         session->attributes.isLockoutBound = (session->attributes.isDaBound == SET)
355             && (bind == TPM_RH_LOCKOUT);
356         return TPM_RC_SUCCESS;
357     }
```

8.9.6.3 SessionContextSave()

This function is called when a session context is to be saved. The *contextID* of the saved session is returned. If no *contextID* can be assigned, then the routine returns TPM_RC_CONTEXT_GAP. If the function completes normally, the session slot will be freed.

This function requires that *handle* references a loaded session. Otherwise, it should not be called at the first place.

Error Returns	Meaning
TPM_RC_CONTEXT_GAP	a *contextID* could not be assigned
TPM_RC_TOO_MANY_CONTEXTS	the counter maxed out

```
358     TPM_RC
359     SessionContextSave(
360         TPM_HANDLE           handle,         // IN: session handle
361         CONTEXT_COUNTER     *contextID       // OUT: assigned contextID
362         )
363     {
364         UINT32                       contextIndex;
365         CONTEXT_SLOT                 slotIndex;
366
```

 Copyright © TCG 2006-2020

```
367       pAssert(SessionIsLoaded(handle));
368
369       // check to see if the gap is already maxed out
370       // Need to have a saved session
371       if(s_oldestSavedSession < MAX_ACTIVE_SESSIONS
372             // if the oldest saved session has the same value as the low bits
373             // of the contextCounter, then the GAP is maxed out.
374          && gr.contextArray[s_oldestSavedSession] == (CONTEXT_SLOT)gr.contextCounter)
375           return TPM_RC_CONTEXT_GAP;
376
377       // if the caller wants the context counter, set it
378       if(contextID != NULL)
379           *contextID = gr.contextCounter;
380
381       contextIndex = handle & HR_HANDLE_MASK;
382       pAssert(contextIndex < MAX_ACTIVE_SESSIONS);
383
384       // Extract the session slot number referenced by the contextArray
385       // because we are going to overwrite this with the low order
386       // contextID value.
387       slotIndex = gr.contextArray[contextIndex] - 1;
388
389       // Set the contextID for the contextArray
390       gr.contextArray[contextIndex] = (CONTEXT_SLOT)gr.contextCounter;
391
392       // Increment the counter
393       gr.contextCounter++;
394
395       // In the unlikely event that the 64-bit context counter rolls over...
396       if(gr.contextCounter == 0)
397       {
398           // back it up
399           gr.contextCounter--;
400           // return an error
401           return TPM_RC_TOO_MANY_CONTEXTS;
402       }
403       // if the low-order bits wrapped, need to advance the value to skip over
404       // the values used to indicate that a session is loaded
405       if(((CONTEXT_SLOT)gr.contextCounter) == 0)
406           gr.contextCounter += MAX_LOADED_SESSIONS + 1;
407
408       // If no other sessions are saved, this is now the oldest.
409       if(s_oldestSavedSession >= MAX_ACTIVE_SESSIONS)
410           s_oldestSavedSession = contextIndex;
411
412       // Mark the session slot as unoccupied
413       s_sessions[slotIndex].occupied = FALSE;
414
415       // and indicate that there is an additional open slot
416       s_freeSessionSlots++;
417
418       return TPM_RC_SUCCESS;
419   }
```

8.9.6.4 SessionContextLoad()

This function is used to load a session from saved context. The session handle must be for a saved context.

If the gap is at a maximum, then the only session that can be loaded is the oldest session, otherwise TPM_RC_CONTEXT_GAP is returned.

This function requires that *handle* references a valid saved session.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

Error Returns	Meaning
TPM_RC_SESSION_MEMORY	no free session slots
TPM_RC_CONTEXT_GAP	the gap count is maximum and this is not the oldest saved context

```
420  TPM_RC
421  SessionContextLoad(
422      SESSION_BUF     *session,       // IN: session structure from saved context
423      TPM_HANDLE      *handle         // IN/OUT: session handle
424      )
425  {
426      UINT32              contextIndex;
427      CONTEXT_SLOT        slotIndex;
428
429      pAssert(HandleGetType(*handle) == TPM_HT_POLICY_SESSION
430              || HandleGetType(*handle) == TPM_HT_HMAC_SESSION);
431
432      // Don't bother looking if no openings
433      if(s_freeSessionSlots == 0)
434          return TPM_RC_SESSION_MEMORY;
435
436      // Find a free session slot to load the session
437      for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
438          if(s_sessions[slotIndex].occupied == FALSE) break;
439
440      // if no spot found, then this is an internal error
441      pAssert(slotIndex < MAX_LOADED_SESSIONS);
442
443      contextIndex = *handle & HR_HANDLE_MASK;   // extract the index
444
445      // If there is only one slot left, and the gap is at maximum, the only session
446      // context that we can safely load is the oldest one.
447      if(s_oldestSavedSession < MAX_ACTIVE_SESSIONS
448         && s_freeSessionSlots == 1
449         && (CONTEXT_SLOT)gr.contextCounter == gr.contextArray[s_oldestSavedSession]
450         && contextIndex != s_oldestSavedSession)
451          return TPM_RC_CONTEXT_GAP;
452
453      pAssert(contextIndex < MAX_ACTIVE_SESSIONS);
454
455      // set the contextArray value to point to the session slot where
456      // the context is loaded
457      gr.contextArray[contextIndex] = slotIndex + 1;
458
459      // if this was the oldest context, find the new oldest
460      if(contextIndex == s_oldestSavedSession)
461          ContextIdSetOldest();
462
463      // Copy session data to session slot
464      MemoryCopy(&s_sessions[slotIndex].session, session, sizeof(SESSION));
465
466             // Set session slot as occupied
467      s_sessions[slotIndex].occupied = TRUE;
468
469      // Reduce the number of open spots
470      s_freeSessionSlots--;
471
472      return TPM_RC_SUCCESS;
473  }
```

8.9.6.5 SessionFlush()

This function is used to flush a session referenced by its handle. If the session associated with *handle* is loaded, the session array entry is marked as available.

This function requires that *handle* be a valid active session.

```
474 void
475 SessionFlush(
476     TPM_HANDLE       handle          // IN: loaded or saved session handle
477     )
478 {
479     CONTEXT_SLOT         slotIndex;
480     UINT32               contextIndex;    // Index into contextArray
481
482     pAssert((HandleGetType(handle) == TPM_HT_POLICY_SESSION
483              || HandleGetType(handle) == TPM_HT_HMAC_SESSION
484              )
485             && (SessionIsLoaded(handle) || SessionIsSaved(handle))
486             );
487
488          // Flush context ID of this session
489          // Convert handle to an index into the contextArray
490     contextIndex = handle & HR_HANDLE_MASK;
491
492     pAssert(contextIndex < sizeof(gr.contextArray) / sizeof(gr.contextArray[0]));
493
494     // Get the current contents of the array
495     slotIndex = gr.contextArray[contextIndex];
496
497     // Mark context array entry as available
498     gr.contextArray[contextIndex] = 0;
499
500     // Is this a saved session being flushed
501     if(slotIndex > MAX_LOADED_SESSIONS)
502     {
503         // Flushing the oldest session?
504         if(contextIndex == s_oldestSavedSession)
505             // If so, find a new value for oldest.
506             ContextIdSetOldest();
507     }
508     else
509     {
510         // Adjust slot index to point to session array index
511         slotIndex -= 1;
512
513         // Free session array index
514         s_sessions[slotIndex].occupied = FALSE;
515         s_freeSessionSlots++;
516     }
517
518     return;
519 }
```

8.9.6.6 SessionComputeBoundEntity()

This function computes the binding value for a session. The binding value for a reserved handle is the handle itself. For all the other entities, the *authValue* at the time of binding is included to prevent squatting. For those values, the Name and the *authValue* are concatenated into the bind buffer. If they will not both fit, the will be overlapped by XORing bytes. If XOR is required, the bind value will be full.

```
520 void
521 SessionComputeBoundEntity(
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
522      TPMI_DH_ENTITY       entityHandle,  // IN: handle of entity
523      TPM2B_NAME          *bind           // OUT: binding value
524      )
525  {
526      TPM2B_AUTH           auth;
527      BYTE                *pAuth = auth.t.buffer;
528      UINT16               i;
529
530      // Get name
531      EntityGetName(entityHandle, bind);
532
533  //    // The bound value of a reserved handle is the handle itself
534  //    if(bind->t.size == sizeof(TPM_HANDLE)) return;
535
536      // For all the other entities, concatenate the authorization value to the name.
537      // Get a local copy of the authorization value because some overlapping
538      // may be necessary.
539      EntityGetAuthValue(entityHandle, &auth);
540
541      // Make sure that the extra space is zeroed
542      MemorySet(&bind->t.name[bind->t.size], 0, sizeof(bind->t.name) - bind->t.size);
543      // XOR the authValue at the end of the name
544      for(i = sizeof(bind->t.name) - auth.t.size; i < sizeof(bind->t.name); i++)
545          bind->t.name[i] ^= *pAuth++;
546
547      // Set the bind value to the maximum size
548      bind->t.size = sizeof(bind->t.name);
549
550      return;
551  }
```

8.9.6.7 SessionSetStartTime()

This function is used to initialize the session timing

```
552  void
553  SessionSetStartTime(
554      SESSION         *session        // IN: the session to update
555      )
556  {
557      session->startTime = g_time;
558      session->epoch = g_timeEpoch;
559      session->timeout = 0;
560  }
```

8.9.6.8 SessionResetPolicyData()

This function is used to reset the policy data without changing the nonce or the start time of the session.

```
561  void
562  SessionResetPolicyData(
563      SESSION         *session        // IN: the session to reset
564      )
565  {
566      SESSION_ATTRIBUTES      oldAttributes;
567      pAssert(session != NULL);
568
569      // Will need later
570      oldAttributes = session->attributes;
571
572      // No command
573      session->commandCode = 0;
574
```

 Copyright © TCG 2006-2020

```
575         // No locality selected
576         MemorySet(&session->commandLocality, 0, sizeof(session->commandLocality));
577
578         // The cpHash size to zero
579         session->u1.cpHash.b.size = 0;
580
581         // No timeout
582         session->timeout = 0;
583
584         // Reset the pcrCounter
585         session->pcrCounter = 0;
586
587         // Reset the policy hash
588         MemorySet(&session->u2.policyDigest.t.buffer, 0,
589                   session->u2.policyDigest.t.size);
590
591         // Reset the session attributes
592         MemorySet(&session->attributes, 0, sizeof(SESSION_ATTRIBUTES));
593
594         // Restore the policy attributes
595         session->attributes.isPolicy = SET;
596         session->attributes.isTrialPolicy = oldAttributes.isTrialPolicy;
597
598         // Restore the bind attributes
599         session->attributes.isDaBound = oldAttributes.isDaBound;
600         session->attributes.isLockoutBound = oldAttributes.isLockoutBound;
601     }
```

8.9.6.9 SessionCapGetLoaded()

This function returns a list of handles of loaded session, started from input *handle*

Handle must be in valid loaded session handle range, but does not have to point to a loaded session.

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
602     TPMI_YES_NO
603     SessionCapGetLoaded(
604         TPMI_SH_POLICY    handle,         // IN: start handle
605         UINT32            count,          // IN: count of returned handles
606         TPML_HANDLE      *handleList      // OUT: list of handle
607         )
608     {
609         TPMI_YES_NO     more = NO;
610         UINT32          i;
611
612         pAssert(HandleGetType(handle) == TPM_HT_LOADED_SESSION);
613
614         // Initialize output handle list
615         handleList->count = 0;
616
617         // The maximum count of handles we may return is MAX_CAP_HANDLES
618         if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
619
620         // Iterate session context ID slots to get loaded session handles
621         for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
622         {
623             // If session is active
624             if(gr.contextArray[i] != 0)
625             {
626                 // If session is loaded
```

```
627             if(gr.contextArray[i] <= MAX_LOADED_SESSIONS)
628             {
629                 if(handleList->count < count)
630                 {
631                     SESSION         *session;
632
633                     // If we have not filled up the return list, add this
634                     // session handle to it
635                     // assume that this is going to be an HMAC session
636                     handle = i + HMAC_SESSION_FIRST;
637                     session = SessionGet(handle);
638                     if(session->attributes.isPolicy)
639                         handle = i + POLICY_SESSION_FIRST;
640                     handleList->handle[handleList->count] = handle;
641                     handleList->count++;
642                 }
643                 else
644                 {
645                     // If the return list is full but we still have loaded object
646                     // available, report this and stop iterating
647                     more = YES;
648                     break;
649                 }
650             }
651         }
652     }
653
654     return more;
655 }
```

8.9.6.10 SessionCapGetSaved()

This function returns a list of handles for saved session, starting at *handle*.

Handle must be in a valid handle range, but does not have to point to a saved session

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
656 TPMI_YES_NO
657 SessionCapGetSaved(
658     TPMI_SH_HMAC     handle,         // IN: start handle
659     UINT32           count,          // IN: count of returned handles
660     TPML_HANDLE     *handleList      // OUT: list of handle
661     )
662 {
663     TPMI_YES_NO     more = NO;
664     UINT32          i;
665
666 #ifdef  TPM_HT_SAVED_SESSION
667     pAssert(HandleGetType(handle) == TPM_HT_SAVED_SESSION);
668 #else
669     pAssert(HandleGetType(handle) == TPM_HT_ACTIVE_SESSION);
670 #endif
671
672     // Initialize output handle list
673     handleList->count = 0;
674
675     // The maximum count of handles we may return is MAX_CAP_HANDLES
676     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
677
678     // Iterate session context ID slots to get loaded session handles
```

```
679         for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
680         {
681             // If session is active
682             if(gr.contextArray[i] != 0)
683             {
684                 // If session is saved
685                 if(gr.contextArray[i] > MAX_LOADED_SESSIONS)
686                 {
687                     if(handleList->count < count)
688                     {
689                         // If we have not filled up the return list, add this
690                         // session handle to it
691                         handleList->handle[handleList->count] = i + HMAC_SESSION_FIRST;
692                         handleList->count++;
693                     }
694                     else
695                     {
696                         // If the return list is full but we still have loaded object
697                         // available, report this and stop iterating
698                         more = YES;
699                         break;
700                     }
701                 }
702             }
703         }
704
705         return more;
706     }
```

8.9.6.11 SessionCapGetLoadedNumber()

This function return the number of authorization sessions currently loaded into TPM RAM.

```
707     UINT32
708     SessionCapGetLoadedNumber(
709         void
710         )
711     {
712         return MAX_LOADED_SESSIONS - s_freeSessionSlots;
713     }
```

8.9.6.12 SessionCapGetLoadedAvail()

This function returns the number of additional authorization sessions, of any type, that could be loaded into TPM RAM.

NOTE: In other implementations, this number may just be an estimate. The only requirement for the estimate is, if it is one or more, then at least one session must be loadable.

```
714     UINT32
715     SessionCapGetLoadedAvail(
716         void
717         )
718     {
719         return s_freeSessionSlots;
720     }
```

8.9.6.13 SessionCapGetActiveNumber()

This function returns the number of active authorization sessions currently being tracked by the TPM.

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
721  UINT32
722  SessionCapGetActiveNumber(
723      void
724      )
725  {
726      UINT32              i;
727      UINT32              num = 0;
728
729      // Iterate the context array to find the number of non-zero slots
730      for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
731      {
732          if(gr.contextArray[i] != 0) num++;
733      }
734
735      return num;
736  }
```

8.9.6.14 SessionCapGetActiveAvail()

This function returns the number of additional authorization sessions, of any type, that could be created. This not the number of slots for sessions, but the number of additional sessions that the TPM is capable of tracking.

```
737  UINT32
738  SessionCapGetActiveAvail(
739      void
740      )
741  {
742      UINT32              i;
743      UINT32              num = 0;
744
745      // Iterate the context array to find the number of zero slots
746      for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
747      {
748          if(gr.contextArray[i] == 0) num++;
749      }
750
751      return num;
752  }
```

8.10 Time.c

8.10.1 Introduction

This file contains the functions relating to the TPM's time functions including the interface to the implementation-specific time functions.

8.10.2 Includes

```
1   #include "Tpm.h"
2   #include "PlatformClock.h"
```

8.10.3 Functions

8.10.3.1 TimePowerOn()

This function initialize time info at _TPM_Init().

This function is called at _TPM_Init() so that the TPM time can start counting as soon as the TPM comes out of reset and doesn't have to wait until TPM2_Startup() in order to begin the new time epoch. This could be significant for systems that could get powered up but not run any TPM commands for some period of time.

```
3   void
4   TimePowerOn(
5       void
6       )
7   {
8       g_time = _plat__TimerRead();
9   }
```

8.10.3.2 TimeNewEpoch()

This function does the processing to generate a new time epoch nonce and set NV for update. This function is only called when NV is known to be available and the clock is running. The epoch is updated to persistent data.

```
10  static void
11  TimeNewEpoch(
12      void
13      )
14  {
15  #if CLOCK_STOPS
16      CryptRandomGenerate(sizeof(CLOCK_NONCE), (BYTE *)&g_timeEpoch);
17  #else
18      // if the epoch is kept in NV, update it.
19      gp.timeEpoch++;
20      NV_SYNC_PERSISTENT(timeEpoch);
21  #endif
22      // Clean out any lingering state
23      _plat__TimerWasStopped();
24  }
```

8.10.3.3 TimeStartup()

This function updates the *resetCount* and *restartCount* components of TPMS_CLOCK_INFO structure at TPM2_Startup().

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

This function will deal with the deferred creation of a new epoch. TimeUpdateToCurrent() will not start a new epoch even if one is due when TPM_Startup() has not been run. This is because the state of NV is not known until startup completes. When Startup is done, then it will create the epoch nonce to complete the initializations by calling this function.

```
25  BOOL
26  TimeStartup(
27      STARTUP_TYPE      type              // IN: start up type
28      )
29  {
30      NOT_REFERENCED(type);
31      // If the previous cycle is orderly shut down, the value of the safe bit
32      // the same as previously saved.  Otherwise, it is not safe.
33      if(!NV_IS_ORDERLY)
34          go.clockSafe = NO;
35      return TRUE;
36  }
```

8.10.3.4 TimeClockUpdate()

This function updates go.clock. If *newTime* requires an update of NV, then NV is checked for availability. If it is not available or is rate limiting, then go.clock is not updated and the function returns an error. If *newTime* would not cause an NV write, then go.clock is updated. If an NV write occurs, then go.safe is SET.

```
37  void
38  TimeClockUpdate(
39      UINT64            newTime     // IN: New time value in mS.
40      )
41  {
42  #define CLOCK_UPDATE_MASK  ((1ULL << NV_CLOCK_UPDATE_INTERVAL)- 1)
43
44      // Check to see if the update will cause a need for an nvClock update
45      if((newTime | CLOCK_UPDATE_MASK) > (go.clock | CLOCK_UPDATE_MASK))
46      {
47          pAssert(g_NvStatus == TPM_RC_SUCCESS);
48
49          // Going to update the NV time state so SET the safe flag
50          go.clockSafe = YES;
51
52          // update the time
53          go.clock = newTime;
54
55          NvWrite(NV_ORDERLY_DATA, sizeof(go), &go);
56      }
57      else
58          // No NV update needed so just update
59          go.clock = newTime;
60
61  }
```

8.10.3.5 TimeUpdate()

This function is used to update the time and clock values. If the TPM has run TPM2_Startup(), this function is called at the start of each command. If the TPM has not run TPM2_Startup(), this is called from TPM2_Startup() to get the clock values initialized. It is not called on command entry because, in this implementation, the go structure is not read from NV until TPM2_Startup(). The reason for this is that the initialization code (_TPM_Init()) may run before NV is accessible.

```
62  void
63  TimeUpdate(
```

```
64       void
65       )
66   {
67       UINT64          elapsed;
68   //
69       // Make sure that we consume the current _plat__TimerWasStopped() state.
70      if(_plat__TimerWasStopped())
71       {
72           TimeNewEpoch();
73       }
74       // Get the difference between this call and the last time we updated the tick
75       // timer.
76       elapsed = _plat__TimerRead() - g_time;
77       // Don't read +
78       g_time += elapsed;
79
80       // Don't need to check the result because it has to be success because have
81       // already checked that NV is available.
82       TimeClockUpdate(go.clock + elapsed);
83
84       // Call self healing logic for dictionary attack parameters
85       DASelfHeal();
86   }
```

8.10.3.6 TimeUpdateToCurrent()

This function updates the *Time* and *Clock* in the global TPMS_TIME_INFO structure.

In this implementation, *Time* and *Clock* are updated at the beginning of each command and the values are unchanged for the duration of the command.

Because *Clock* updates may require a write to NV memory, *Time* and *Clock* are not allowed to advance if NV is not available. When clock is not advancing, any function that uses *Clock* will fail and return TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE.

This implementation does not do rate limiting. If the implementation does do rate limiting, then the *Clock* update should not be inhibited even when doing rate limiting.

```
87   void
88   TimeUpdateToCurrent(
89       void
90   )
91   {
92       // Can't update time during the dark interval or when rate limiting so don't
93       // make any modifications to the internal clock value. Also, defer any clock
94       // processing until TPM has run TPM2_Startup()
95       if(!NV_IS_AVAILABLE || !TPMIsStarted())
96           return;
97
98       TimeUpdate();
99   }
```

8.10.3.7 TimeSetAdjustRate()

This function is used to perform rate adjustment on *Time* and *Clock*.

```
100  void
101  TimeSetAdjustRate(
102      TPM_CLOCK_ADJUST     adjust          // IN: adjust constant
103      )
104  {
105      switch(adjust)
106      {
```

```
107             case TPM_CLOCK_COARSE_SLOWER:
108                 _plat__ClockAdjustRate(CLOCK_ADJUST_COARSE);
109                 break;
110             case TPM_CLOCK_COARSE_FASTER:
111                 _plat__ClockAdjustRate(-CLOCK_ADJUST_COARSE);
112                 break;
113             case TPM_CLOCK_MEDIUM_SLOWER:
114                 _plat__ClockAdjustRate(CLOCK_ADJUST_MEDIUM);
115                 break;
116             case TPM_CLOCK_MEDIUM_FASTER:
117                 _plat__ClockAdjustRate(-CLOCK_ADJUST_MEDIUM);
118                 break;
119             case TPM_CLOCK_FINE_SLOWER:
120                 _plat__ClockAdjustRate(CLOCK_ADJUST_FINE);
121                 break;
122             case TPM_CLOCK_FINE_FASTER:
123                 _plat__ClockAdjustRate(-CLOCK_ADJUST_FINE);
124                 break;
125             case TPM_CLOCK_NO_CHANGE:
126                 break;
127             default:
128                 FAIL(FATAL_ERROR_INTERNAL);
129                 break;
130         }
131
132         return;
133     }
```

8.10.3.8 TimeGetMarshaled()

This function is used to access TPMS_TIME_INFO in canonical form. The function collects the time information and marshals it into *dataBuffer* and returns the marshaled size

```
134     UINT16
135     TimeGetMarshaled(
136         TIME_INFO       *dataBuffer     // OUT: result buffer
137         )
138     {
139         TPMS_TIME_INFO      timeInfo;
140
141         // Fill TPMS_TIME_INFO structure
142         timeInfo.time = g_time;
143         TimeFillInfo(&timeInfo.clockInfo);
144
145         // Marshal TPMS_TIME_INFO to canonical form
146         return TPMS_TIME_INFO_Marshal(&timeInfo, (BYTE **)&dataBuffer, NULL);
147     }
```

8.10.3.9 TimeFillInfo

This function gathers information to fill in a TPMS_CLOCK_INFO structure.

```
148     void
149     TimeFillInfo(
150         TPMS_CLOCK_INFO     *clockInfo
151         )
152     {
153         clockInfo->clock = go.clock;
154         clockInfo->resetCount = gp.resetCount;
155         clockInfo->restartCount = gr.restartCount;
156
157         // If NV is not available, clock stopped advancing and the value reported is
158         // not "safe".
```

```
159     if(NV_IS_AVAILABLE)
160         clockInfo->safe = go.clockSafe;
161     else
162         clockInfo->safe = NO;
163
164     return;
165 }
```

9 Support

9.1 AlgorithmCap.c

9.1.1 Description

This file contains the algorithm property definitions for the algorithms and the code for the TPM2_GetCapability() to return the algorithm properties.

9.1.2 Includes and Defines

```
1   #include "Tpm.h"
2   typedef struct
3   {
4       TPM_ALG_ID          algID;
5       TPMA_ALGORITHM      attributes;
6   } ALGORITHM;
7   static const ALGORITHM    s_algorithms[] =
8   {
9   // The entries in this table need to be in ascending order but the table doesn't
10  // need to be full (gaps are allowed). One day, a tool might exist to fill in the
11  // table from the TPM_ALG description
12  #if ALG_RSA
13      {TPM_ALG_RSA,           TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 1, 0, 0, 0, 0, 0)},
14  #endif
15  #if ALG_TDES
16      {TPM_ALG_TDES,          TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
17  #endif
18  #if ALG_SHA1
19      {TPM_ALG_SHA1,          TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
20  #endif
21
22      {TPM_ALG_HMAC,          TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 1, 0, 0, 0)},
23
24  #if ALG_AES
25      {TPM_ALG_AES,           TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
26  #endif
27  #if ALG_MGF1
28      {TPM_ALG_MGF1,          TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)},
29  #endif
30
31      {TPM_ALG_KEYEDHASH,     TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 1, 0, 1, 1, 0, 0)},
32
33  #if ALG_XOR
34      {TPM_ALG_XOR,           TPMA_ALGORITHM_INITIALIZER(0, 1, 1, 0, 0, 0, 0, 0, 0)},
35  #endif
36
37  #if ALG_SHA256
38      {TPM_ALG_SHA256,        TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
39  #endif
40  #if ALG_SHA384
41      {TPM_ALG_SHA384,        TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
42  #endif
43  #if ALG_SHA512
44      {TPM_ALG_SHA512,        TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
45  #endif
46  #if ALG_SM3_256
47      {TPM_ALG_SM3_256,       TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
48  #endif
49  #if ALG_SM4
50      {TPM_ALG_SM4,           TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
```

 Copyright © TCG 2006-2020

```
51  #endif
52  #if ALG_RSASSA
53      {TPM_ALG_RSASSA,         TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
54  #endif
55  #if ALG_RSAES
56      {TPM_ALG_RSAES,          TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 1, 0, 0)},
57  #endif
58  #if ALG_RSAPSS
59      {TPM_ALG_RSAPSS,         TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
60  #endif
61  #if ALG_OAEP
62      {TPM_ALG_OAEP,           TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 1, 0, 0)},
63  #endif
64  #if ALG_ECDSA
65      {TPM_ALG_ECDSA,          TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 1, 0)},
66  #endif
67  #if ALG_ECDH
68      {TPM_ALG_ECDH,           TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 0, 1, 0)},
69  #endif
70  #if ALG_ECDAA
71      {TPM_ALG_ECDAA,          TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
72  #endif
73  #if ALG_SM2
74      {TPM_ALG_SM2,            TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 1, 0)},
75  #endif
76  #if ALG_ECSCHNORR
77      {TPM_ALG_ECSCHNORR,       TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
78  #endif
79  #if ALG_ECMQV
80      {TPM_ALG_ECMQV,           TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 0, 1, 0)},
81  #endif
82  #if ALG_KDF1_SP800_56A
83      {TPM_ALG_KDF1_SP800_56A, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)},
84  #endif
85  #if ALG_KDF2
86      {TPM_ALG_KDF2,            TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)},
87  #endif
88  #if ALG_KDF1_SP800_108
89      {TPM_ALG_KDF1_SP800_108, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)},
90  #endif
91  #if ALG_ECC
92      {TPM_ALG_ECC,             TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 1, 0, 0, 0, 0, 0)},
93  #endif
94
95      {TPM_ALG_SYMCIPHER,       TPMA_ALGORITHM_INITIALIZER(0, 0, 0, 1, 0, 0, 0, 0, 0)},
96
97  #if ALG_CAMELLIA
98      {TPM_ALG_CAMELLIA,        TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
99  #endif
100 #if ALG_CMAC
101     {TPM_ALG_CMAC,            TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 1, 0, 0, 0)},
102 #endif
103 #if ALG_CTR
104     {TPM_ALG_CTR,             TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
105 #endif
106 #if ALG_OFB
107     {TPM_ALG_OFB,             TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
108 #endif
109 #if ALG_CBC
110     {TPM_ALG_CBC,             TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
111 #endif
112 #if ALG_CFB
113     {TPM_ALG_CFB,             TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
114 #endif
115 #if ALG_ECB
116     {TPM_ALG_ECB,             TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
```

 Copyright © TCG 2006-2020

```
117 #endif
118 };
```

9.1.3 AlgorithmCapGetImplemented()

This function is used by TPM2_GetCapability() to return a list of the implemented algorithms.

Return Value	Meaning
YES	more algorithms to report
NO	no more algorithms to report

```
119 TPMI_YES_NO
120 AlgorithmCapGetImplemented(
121     TPM_ALG_ID                      algID,      // IN: the starting algorithm ID
122     UINT32                          count,      // IN: count of returned algorithms
123     TPML_ALG_PROPERTY              *algList     // OUT: algorithm list
124     )
125 {
126     TPMI_YES_NO     more = NO;
127     UINT32          i;
128     UINT32          algNum;
129
130     // initialize output algorithm list
131     algList->count = 0;
132
133     // The maximum count of algorithms we may return is MAX_CAP_ALGS.
134     if(count > MAX_CAP_ALGS)
135         count = MAX_CAP_ALGS;
136
137     // Compute how many algorithms are defined in s_algorithms array.
138     algNum = sizeof(s_algorithms) / sizeof(s_algorithms[0]);
139
140     // Scan the implemented algorithm list to see if there is a match to 'algID'.
141     for(i = 0; i < algNum; i++)
142     {
143         // If algID is less than the starting algorithm ID, skip it
144         if(s_algorithms[i].algID < algID)
145             continue;
146         if(algList->count < count)
147         {
148             // If we have not filled up the return list, add more algorithms
149             // to it
150             algList->algProperties[algList->count].alg = s_algorithms[i].algID;
151             algList->algProperties[algList->count].algProperties =
152                 s_algorithms[i].attributes;
153             algList->count++;
154         }
155         else
156         {
157             // If the return list is full but we still have algorithms
158             // available, report this and stop scanning.
159             more = YES;
160             break;
161         }
162     }
163
164     return more;
165 }
```

9.1.4 AlgorithmGetImplementedVector()

This function returns the bit vector of the implemented algorithms.

```
166 LIB_EXPORT
167 void
168 AlgorithmGetImplementedVector(
169     ALGORITHM_VECTOR    *implemented    // OUT: the implemented bits are SET
170     )
171 {
172     int                         index;
173
174     // Nothing implemented until we say it is
175     MemorySet(implemented, 0, sizeof(ALGORITHM_VECTOR));
176
177     for(index = (sizeof(s_algorithms) / sizeof(s_algorithms[0])) - 1;
178     index >= 0;
179         index--)
180         SET_BIT(s_algorithms[index].algID, *implemented);
181     return;
182 }
```

9.2 Bits.c

9.2.1 Introduction

This file contains bit manipulation routines. They operate on bit arrays.

The 0th bit in the array is the right-most bit in the 0th octet in the array.

NOTE: If pAssert() is defined, the functions will assert if the indicated bit number is outside of the range of *bArray*. How the assert is handled is implementation dependent.

9.2.2 Includes

```
1   #include "Tpm.h"
```

9.2.3 Functions

9.2.3.1 TestBit()

This function is used to check the setting of a bit in an array of bits.

Return Value	Meaning
TRUE(1)	bit is set
FALSE(0)	bit is not set

```
2   BOOL
3   TestBit(
4       unsigned int     bitNum,         // IN: number of the bit in 'bArray'
5       BYTE            *bArray,         // IN: array containing the bits
6       unsigned int     bytesInArray    // IN: size in bytes of 'bArray'
7       )
8   {
9       pAssert(bytesInArray > (bitNum >> 3));
10      return((bArray[bitNum >> 3] & (1 << (bitNum & 7))) != 0);
11  }
```

9.2.3.2 SetBit()

This function will set the indicated bit in *bArray*.

```
12  void
13  SetBit(
14      unsigned int     bitNum,         // IN: number of the bit in 'bArray'
15      BYTE            *bArray,         // IN: array containing the bits
16      unsigned int     bytesInArray    // IN: size in bytes of 'bArray'
17      )
18  {
19      pAssert(bytesInArray > (bitNum >> 3));
20      bArray[bitNum >> 3] |= (1 << (bitNum & 7));
21  }
```

9.2.3.3 ClearBit()

This function will clear the indicated bit in *bArray*.

```
22  void
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
23  ClearBit(
24      unsigned int      bitNum,        // IN: number of the bit in 'bArray'.
25      BYTE             *bArray,        // IN: array containing the bits
26      unsigned int      bytesInArray   // IN: size in bytes of 'bArray'
27      )
28  {
29      pAssert(bytesInArray > (bitNum >> 3));
30      bArray[bitNum >> 3] &= ~(1 << (bitNum & 7));
31  }
```

9.3 CommandCodeAttributes.c

9.3.1 Introduction

This file contains the functions for testing various command properties.

9.3.2 Includes and Defines

```
1   #include "Tpm.h"
2   #include "CommandCodeAttributes_fp.h"
```

Set the default value for CC_VEND if not already set

```
3   #ifndef CC_VEND
4   #define     CC_VEND     (TPM_CC)(0x20000000)
5   #endif
6   typedef UINT16          ATTRIBUTE_TYPE;
```

The following file is produced from the command tables in part 3 of the specification. It defines the attributes for each of the commands.

NOTE: This file is currently produced by an automated process. Files produced from Part 2 or Part 3 tables through automated processes are not included in the specification so that there is no ambiguity about the table containing the information being the normative definition.

```
7   #define _COMMAND_CODE_ATTRIBUTES_
8   #include    "CommandAttributeData.h"
```

9.3.3 Command Attribute Functions

9.3.3.1 NextImplementedIndex()

This function is used when the lists are not compressed. In a compressed list, only the implemented commands are present. So, a search might find a value but that value may not be implemented. This function checks to see if the input *commandIndex* points to an implemented command and, if not, it searches upwards until it finds one. When the list is compressed, this function gets defined as a no-op.

Return Value	Meaning
UNIMPLEMENTED_COMMAND_INDEX	command is not implemented
other	index of the command

```
9   #if !COMPRESSED_LISTS
10  static COMMAND_INDEX
11  NextImplementedIndex(
12      COMMAND_INDEX       commandIndex
13      )
14  {
15      for(;commandIndex < COMMAND_COUNT; commandIndex++)
16      {
17          if(s_commandAttributes[commandIndex] & IS_IMPLEMENTED)
18              return commandIndex;
19      }
20      return UNIMPLEMENTED_COMMAND_INDEX;
21  }
22  #else
23  #define NextImplementedIndex(x)  (x)
24  #endif
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

9.3.3.2 GetClosestCommandIndex()

This function returns the command index for the command with a value that is equal to or greater than the input value

Return Value	Meaning
UNIMPLEMENTED_COMMAND_INDEX	command is not implemented
other	index of a command

```
25  COMMAND_INDEX
26  GetClosestCommandIndex(
27      TPM_CC            commandCode    // IN: the command code to start at
28      )
29  {
30      BOOL                vendor = (commandCode & CC_VEND) != 0;
31      COMMAND_INDEX       searchIndex = (COMMAND_INDEX)commandCode;
32
33      // The commandCode is a UINT32 and the search index is UINT16. We are going to
34      // search for a match but need to make sure that the commandCode value is not
35      // out of range. To do this, need to clear the vendor bit of the commandCode
36      // (if set) and compare the result to the 16-bit searchIndex value. If it is
37      // out of range, indicate that the command is not implemented
38      if((commandCode & ~CC_VEND) != searchIndex)
39          return UNIMPLEMENTED_COMMAND_INDEX;
40
41      // if there is at least one vendor command, the last entry in the array will
42      // have the v bit set. If the input commandCode is larger than the last
43      // vendor-command, then it is out of range.
44      if(vendor)
45      {
46  #if VENDOR_COMMAND_ARRAY_SIZE > 0
47          COMMAND_INDEX       commandIndex;
48          COMMAND_INDEX       min;
49          COMMAND_INDEX       max;
50          int                 diff;
51  #if LIBRARY_COMMAND_ARRAY_SIZE == COMMAND_COUNT
52  #error "Constants are not consistent."
53  #endif
54          // Check to see if the value is equal to or below the minimum
55          // entry.
56          // Note: Put this check first so that the typical case of only one vendor-
57          // specific command doesn't waste any more time.
58          if(GET_ATTRIBUTE(s_ccAttr[LIBRARY_COMMAND_ARRAY_SIZE], TPMA_CC,
59                           commandIndex) >= searchIndex)
60          {
61              // the vendor array is always assumed to be packed so there is
62              // no need to check to see if the command is implemented
63              return LIBRARY_COMMAND_ARRAY_SIZE;
64          }
65          // See if this is out of range on the top
66          if(GET_ATTRIBUTE(s_ccAttr[COMMAND_COUNT - 1], TPMA_CC, commandIndex)
67             < searchIndex)
68          {
69              return UNIMPLEMENTED_COMMAND_INDEX;
70          }
71          commandIndex = UNIMPLEMENTED_COMMAND_INDEX; // Needs initialization to keep
72                                                      // compiler happy
73          min = LIBRARY_COMMAND_ARRAY_SIZE;       // first vendor command
74          max = COMMAND_COUNT - 1;                // last vendor command
75          diff = 1;                               // needs initialization to keep
76                                                  // compiler happy
77          while(min <= max)
78          {
```

 Copyright © TCG 2006-2020

```
79              commandIndex = (min + max + 1) / 2;
80              diff = GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex)
81                  - searchIndex;
82              if(diff == 0)
83                  return commandIndex;
84              if(diff > 0)
85                  max = commandIndex - 1;
86              else
87                  min = commandIndex + 1;
88          }
89          // didn't find and exact match. commandIndex will be pointing at the last
90          // item tested. If 'diff' is positive, then the last item tested was
91          // larger index of the command code so it is the smallest value
92          // larger than the requested value.
93          if(diff > 0)
94              return commandIndex;
95          // if 'diff' is negative, then the value tested was smaller than
96          // the commandCode index and the next higher value is the correct one.
97          // Note: this will necessarily be in range because of the earlier check
98          // that the index was within range.
99          return commandIndex + 1;
100 #else
101     // If there are no vendor commands so anything with the vendor bit set is out
102     // of range
103         return UNIMPLEMENTED_COMMAND_INDEX;
104 #endif
105     }
106     // Get here if the V-Bit was not set in 'commandCode'
107
108     if(GET_ATTRIBUTE(s_ccAttr[LIBRARY_COMMAND_ARRAY_SIZE - 1], TPMA_CC,
109                      commandIndex) < searchIndex)
110     {
111         // requested index is out of the range to the top
112 #if VENDOR_COMMAND_ARRAY_SIZE > 0
113         // If there are vendor commands, then the first vendor command
114         // is the next value greater than the commandCode.
115         // NOTE: we got here if the starting index did not have the V bit but we
116         // reached the end of the array of library commands (non-vendor). Since
117         // there is at least one vendor command, and vendor commands are always
118         // in a compressed list that starts after the library list, the next
119         // index value contains a valid vendor command.
120         return LIBRARY_COMMAND_ARRAY_SIZE;
121 #else
122         // if there are no vendor commands, then this is out of range
123         return UNIMPLEMENTED_COMMAND_INDEX;
124 #endif
125     }
126     // If the request is lower than any value in the array, then return
127     // the lowest value (needs to be an index for an implemented command
128     if(GET_ATTRIBUTE(s_ccAttr[0], TPMA_CC, commandIndex) >= searchIndex)
129     {
130         return NextImplementedIndex(0);
131     }
132     else
133     {
134 #if COMPRESSED_LISTS
135         COMMAND_INDEX       commandIndex = UNIMPLEMENTED_COMMAND_INDEX;
136         COMMAND_INDEX       min = 0;
137         COMMAND_INDEX       max = LIBRARY_COMMAND_ARRAY_SIZE - 1;
138         int                 diff = 1;
139 #if LIBRARY_COMMAND_ARRAY_SIZE == 0
140 #error  "Something is terribly wrong"
141 #endif
142         // The s_ccAttr array contains an extra entry at the end (a zero value).
143         // Don't count this as an array entry. This means that max should start
144         // out pointing to the last valid entry in the array which is - 2
```

```
145             pAssert(max == (sizeof(s_ccAttr) / sizeof(TPMA_CC)
146                             - VENDOR_COMMAND_ARRAY_SIZE - 2));
147             while(min <= max)
148             {
149                 commandIndex = (min + max + 1) / 2;
150                 diff = GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC,
151                                      commandIndex) - searchIndex;
152                 if(diff == 0)
153                     return commandIndex;
154                 if(diff > 0)
155                     max = commandIndex - 1;
156                 else
157                     min = commandIndex + 1;
158             }
159             // didn't find and exact match. commandIndex will be pointing at the
160             // last item tested. If diff is positive, then the last item tested was
161             // larger index of the command code so it is the smallest value
162             // larger than the requested value.
163             if(diff > 0)
164                 return commandIndex;
165             // if diff is negative, then the value tested was smaller than
166             // the commandCode index and the next higher value is the correct one.
167             // Note: this will necessarily be in range because of the earlier check
168             // that the index was within range.
169             return commandIndex + 1;
170     #else
171             // The list is not compressed so offset into the array by the command
172             // code value of the first entry in the list. Then go find the first
173             // implemented command.
174             return NextImplementedIndex(searchIndex
175                                         - (COMMAND_INDEX)s_ccAttr[0].commandIndex);
176     #endif
177         }
178     }
```

9.3.3.3 CommandCodeToComandIndex()

This function returns the index in the various attributes arrays of the command.

Return Value	Meaning
UNIMPLEMENTED_COMMAND_INDEX	command is not implemented
other	index of the command

```
179     COMMAND_INDEX
180     CommandCodeToCommandIndex(
181         TPM_CC          commandCode     // IN: the command code to look up
182         )
183     {
184         // Extract the low 16-bits of the command code to get the starting search index
185         COMMAND_INDEX       searchIndex = (COMMAND_INDEX)commandCode;
186         BOOL                vendor = (commandCode & CC_VEND) != 0;
187         COMMAND_INDEX       commandIndex;
188     #if !COMPRESSED_LISTS
189         if(!vendor)
190         {
191             commandIndex = searchIndex - (COMMAND_INDEX)s_ccAttr[0].commandIndex;
192             // Check for out of range or unimplemented.
193             // Note, since a COMMAND_INDEX is unsigned, if searchIndex is smaller than
194             // the lowest value of command, it will become a 'negative' number making
195             // it look like a large unsigned number, this will cause it to fail
196             // the unsigned check below.
197             if(commandIndex >= LIBRARY_COMMAND_ARRAY_SIZE
```

```
198                 || (s_commandAttributes[commandIndex] & IS_IMPLEMENTED) == 0)
199                 return UNIMPLEMENTED_COMMAND_INDEX;
200             return commandIndex;
201         }
202     #endif
203         // Need this code for any vendor code lookup or for compressed lists
204         commandIndex = GetClosestCommandIndex(commandCode);
205
206         // Look at the returned value from get closest. If it isn't the one that was
207         // requested, then the command is not implemented.
208         if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
209         {
210             if((GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex)
211                 != searchIndex)
212                || (IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V)) != vendor)
213                 commandIndex = UNIMPLEMENTED_COMMAND_INDEX;
214         }
215         return commandIndex;
216     }
```

9.3.3.4 GetNextCommandIndex()

This function returns the index of the next implemented command.

Return Value	Meaning
UNIMPLEMENTED_COMMAND_INDEX	no more implemented commands
other	the index of the next implemented command

```
217     COMMAND_INDEX
218     GetNextCommandIndex(
219         COMMAND_INDEX     commandIndex    // IN: the starting index
220         )
221     {
222         while(++commandIndex < COMMAND_COUNT)
223         {
224     #if !COMPRESSED_LISTS
225             if(s_commandAttributes[commandIndex] & IS_IMPLEMENTED)
226     #endif
227                 return commandIndex;
228         }
229         return UNIMPLEMENTED_COMMAND_INDEX;
230     }
```

9.3.3.5 GetCommandCode()

This function returns the *commandCode* associated with the command index

```
231     TPM_CC
232     GetCommandCode(
233         COMMAND_INDEX     commandIndex    // IN: the command index
234         )
235     {
236         TPM_CC            commandCode = GET_ATTRIBUTE(s_ccAttr[commandIndex],
237                                                       TPMA_CC, commandIndex);
238         if(IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
239             commandCode += CC_VEND;
240         return commandCode;
241     }
```

9.3.3.6 CommandAuthRole()

This function returns the authorization role required of a handle.

Return Value	Meaning
AUTH_NONE	no authorization is required
AUTH_USER	user role authorization is required
AUTH_ADMIN	admin role authorization is required
AUTH_DUP	duplication role authorization is required

```
242 AUTH_ROLE
243 CommandAuthRole(
244     COMMAND_INDEX      commandIndex,  // IN: command index
245     UINT32             handleIndex    // IN: handle index (zero based)
246     )
247 {
248     if(0 == handleIndex)
249     {
250         // Any authorization role set?
251         COMMAND_ATTRIBUTES  properties = s_commandAttributes[commandIndex];
252
253         if(properties & HANDLE_1_USER)
254             return AUTH_USER;
255         if(properties & HANDLE_1_ADMIN)
256             return AUTH_ADMIN;
257         if(properties & HANDLE_1_DUP)
258             return AUTH_DUP;
259     }
260     else if(1 == handleIndex)
261     {
262         if(s_commandAttributes[commandIndex] & HANDLE_2_USER)
263             return AUTH_USER;
264     }
265     return AUTH_NONE;
266 }
```

9.3.3.7 EncryptSize()

This function returns the size of the decrypt size field. This function returns 0 if encryption is not allowed

Return Value	Meaning
0	encryption not allowed
2	size field is two bytes
4	size field is four bytes

```
267 int
268 EncryptSize(
269     COMMAND_INDEX      commandIndex   // IN: command index
270     )
271 {
272     return ((s_commandAttributes[commandIndex] & ENCRYPT_2) ? 2 :
273             (s_commandAttributes[commandIndex] & ENCRYPT_4) ? 4 : 0);
274 }
```

 Copyright © TCG 2006-2020

9.3.3.8 DecryptSize()

This function returns the size of the decrypt size field. This function returns 0 if decryption is not allowed

Return Value	Meaning
0	encryption not allowed
2	size field is two bytes
4	size field is four bytes

```
275 int
276 DecryptSize(
277     COMMAND_INDEX    commandIndex   // IN: command index
278     )
279 {
280     return ((s_commandAttributes[commandIndex] & DECRYPT_2) ? 2 :
281             (s_commandAttributes[commandIndex] & DECRYPT_4) ? 4 : 0);
282 }
```

9.3.3.9 IsSessionAllowed()

This function indicates if the command is allowed to have sessions.

This function must not be called if the command is not known to be implemented.

Return Value	Meaning
TRUE(1)	session is allowed with this command
FALSE(0)	session is not allowed with this command

```
283 BOOL
284 IsSessionAllowed(
285     COMMAND_INDEX    commandIndex   // IN: the command to be checked
286     )
287 {
288     return ((s_commandAttributes[commandIndex] & NO_SESSIONS) == 0);
289 }
```

9.3.3.10 IsHandleInResponse()

This function determines if a command has a handle in the response

```
290 BOOL
291 IsHandleInResponse(
292     COMMAND_INDEX    commandIndex
293     )
294 {
295     return ((s_commandAttributes[commandIndex] & R_HANDLE) != 0);
296 }
```

9.3.3.11 IsWriteOperation()

Checks to see if an operation will write to an NV Index and is subject to being blocked by read-lock

```
297 BOOL
298 IsWriteOperation(
299     COMMAND_INDEX    commandIndex   // IN: Command to check
300     )
```

 Copyright © TCG 2006-2020

```
301 {
302 #ifdef  WRITE_LOCK
303     return ((s_commandAttributes[commandIndex] & WRITE_LOCK) != 0);
304 #else
305     if(!IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
306     {
307         switch(GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex))
308         {
309             case TPM_CC_NV_Write:
310 #if CC_NV_Increment
311             case TPM_CC_NV_Increment:
312 #endif
313 #if CC_NV_SetBits
314             case TPM_CC_NV_SetBits:
315 #endif
316 #if CC_NV_Extend
317             case TPM_CC_NV_Extend:
318 #endif
319 #if CC_AC_Send
320             case TPM_CC_AC_Send:
321 #endif
322             // NV write lock counts as a write operation for authorization purposes.
323             // We check to see if the NV is write locked before we do the
324             // authorization. If it is locked, we fail the command early.
325             case TPM_CC_NV_WriteLock:
326                 return TRUE;
327             default:
328                 break;
329         }
330     }
331     return FALSE;
332 #endif
333 }
```

9.3.3.12 IsReadOperation()

Checks to see if an operation will write to an NV Index and is subject to being blocked by write-lock.

```
334 BOOL
335 IsReadOperation(
336     COMMAND_INDEX     commandIndex    // IN: Command to check
337     )
338 {
339 #ifdef  READ_LOCK
340     return ((s_commandAttributes[commandIndex] & READ_LOCK) != 0);
341 #else
342 
343     if(!IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
344     {
345         switch(GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex))
346         {
347             case TPM_CC_NV_Read:
348             case TPM_CC_PolicyNV:
349             case TPM_CC_NV_Certify:
350             // NV read lock counts as a read operation for authorization purposes.
351             // We check to see if the NV is read locked before we do the
352             // authorization. If it is locked, we fail the command early.
353             case TPM_CC_NV_ReadLock:
354                 return TRUE;
355             default:
356                 break;
357         }
358     }
359     return FALSE;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
360 #endif
361 }
```

9.3.3.13 CommandCapGetCCList()

This function returns a list of implemented commands and command attributes starting from the command in *commandCode*.

Return Value	Meaning
YES	more command attributes are available
NO	no more command attributes are available

```
362 TPMI_YES_NO
363 CommandCapGetCCList(
364     TPM_CC          commandCode,    // IN: start command code
365     UINT32          count,          // IN: maximum count for number of entries in
366                                     //     'commandList'
367     TPML_CCA       *commandList     // OUT: list of TPMA_CC
368     )
369 {
370     TPMI_YES_NO      more = NO;
371     COMMAND_INDEX    commandIndex;
372
373     // initialize output handle list count
374     commandList->count = 0;
375
376     for(commandIndex = GetClosestCommandIndex(commandCode);
377     commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
378         commandIndex = GetNextCommandIndex(commandIndex))
379     {
380 #if !COMPRESSED_LISTS
381         // this check isn't needed for compressed lists.
382         if(!(s_commandAttributes[commandIndex] & IS_IMPLEMENTED))
383             continue;
384 #endif
385         if(commandList->count < count)
386         {
387             // If the list is not full, add the attributes for this command.
388             commandList->commandAttributes[commandList->count]
389                 = s_ccAttr[commandIndex];
390             commandList->count++;
391         }
392         else
393         {
394             // If the list is full but there are more commands to report,
395             // indicate this and return.
396             more = YES;
397             break;
398         }
399     }
400     return more;
401 }
```

9.3.3.14 IsVendorCommand()

Function indicates if a command index references a vendor command.

Return Value	Meaning
TRUE(1)	command is a vendor command
FALSE(0)	command is not a vendor command

```
402 BOOL
403 IsVendorCommand(
404     COMMAND_INDEX     commandIndex    // IN: command index to check
405     )
406 {
407     return (IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V));
408 }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

9.4 Entity.c

9.4.1 Description

The functions in this file are used for accessing properties for handles of various types. Functions in other files require handles of a specific type but the functions in this file allow use of any handle type.

9.4.2 Includes

```
1   #include "Tpm.h"
```

9.4.3 Functions

9.4.3.1 EntityGetLoadStatus()

This function will check that all the handles access loaded entities.

Error Returns	Meaning
TPM_RC_HANDLE	handle type does not match
TPM_RC_REFERENCE_Hx	entity is not present
TPM_RC_HIERARCHY	entity belongs to a disabled hierarchy
TPM_RC_OBJECT_MEMORY	handle is an evict object but there is no space to load it to RAM

```
2   TPM_RC
3   EntityGetLoadStatus(
4       COMMAND          *command        // IN/OUT: command parsing structure
5       )
6   {
7       UINT32                  i;
8       TPM_RC                  result = TPM_RC_SUCCESS;
9   //
10      for(i = 0; i < command->handleNum; i++)
11      {
12          TPM_HANDLE       handle = command->handles[i];
13          switch(HandleGetType(handle))
14          {
15              // For handles associated with hierarchies, the entity is present
16              // only if the associated enable is SET.
17              case TPM_HT_PERMANENT:
18                  switch(handle)
19                  {
20                      case TPM_RH_OWNER:
21                          if(!gc.shEnable)
22                              result = TPM_RC_HIERARCHY;
23                          break;
24
25  #ifdef  VENDOR_PERMANENT
26                      case VENDOR_PERMANENT:
27  #endif
28                      case TPM_RH_ENDORSEMENT:
29                          if(!gc.ehEnable)
30                              result = TPM_RC_HIERARCHY;
31                          break;
32                      case TPM_RH_PLATFORM:
33                          if(!g_phEnable)
34                              result = TPM_RC_HIERARCHY;
35                          break;
```

 Copyright © TCG 2006-2020

```
36                          // null handle, PW session handle and lockout
37                          // handle are always available
38                      case TPM_RH_NULL:
39                      case TPM_RS_PW:
40                          // Need to be careful for lockout. Lockout is always available
41                          // for policy checks but not always available when authValue
42                          // is being checked.
43                      case TPM_RH_LOCKOUT:
44                      // Rather than have #ifdefs all over the code,
45                      // CASE_ACT_HANDLE is defined in ACT.h. It is 'case TPM_RH_ACT_x:'
46                      // FOR_EACH_ACT(CASE_ACT_HANDLE) creates a simple
47                      // case TPM_RH_ACT_x: // for each of the implemented ACT.
48                      FOR_EACH_ACT(CASE_ACT_HANDLE)
49                          break;
50                      default:
51                          // If the implementation has a manufacturer-specific value
52                          // then test for it here. Since this implementation does
53                          // not have any, this implementation returns the same failure
54                          // that unmarshaling of a bad handle would produce.
55                          if(((TPM_RH)handle >= TPM_RH_AUTH_00)
56                             && ((TPM_RH)handle <= TPM_RH_AUTH_FF))
57                              // if the implementation has a manufacturer-specific value
58                              result = TPM_RC_VALUE;
59                          else
60                              // The handle is in the range of reserved handles but is
61                              // not implemented in this TPM.
62                              result = TPM_RC_VALUE;
63                          break;
64                  }
65                  break;
66              case TPM_HT_TRANSIENT:
67                  // For a transient object, check if the handle is associated
68                  // with a loaded object.
69                  if(!IsObjectPresent(handle))
70                      result = TPM_RC_REFERENCE_H0;
71                  break;
72              case TPM_HT_PERSISTENT:
73                  // Persistent object
74                  // Copy the persistent object to RAM and replace the handle with the
75                  // handle of the assigned slot.  A TPM_RC_OBJECT_MEMORY,
76                  // TPM_RC_HIERARCHY or TPM_RC_REFERENCE_H0 error may be returned by
77                  // ObjectLoadEvict()
78                  result = ObjectLoadEvict(&command->handles[i], command->index);
79                  break;
80              case TPM_HT_HMAC_SESSION:
81                  // For an HMAC session, see if the session is loaded
82                  // and if the session in the session slot is actually
83                  // an HMAC session.
84                  if(SessionIsLoaded(handle))
85                  {
86                      SESSION             *session;
87                      session = SessionGet(handle);
88                      // Check if the session is a HMAC session
89                      if(session->attributes.isPolicy == SET)
90                          result = TPM_RC_HANDLE;
91                  }
92                  else
93                      result = TPM_RC_REFERENCE_H0;
94                  break;
95              case TPM_HT_POLICY_SESSION:
96                  // For a policy session, see if the session is loaded
97                  // and if the session in the session slot is actually
98                  // a policy session.
99                  if(SessionIsLoaded(handle))
100                 {
101                     SESSION             *session;
```

```
102                         session = SessionGet(handle);
103                         // Check if the session is a policy session
104                         if(session->attributes.isPolicy == CLEAR)
105                             result = TPM_RC_HANDLE;
106                     }
107                     else
108                         result = TPM_RC_REFERENCE_H0;
109                     break;
110                 case TPM_HT_NV_INDEX:
111                     // For an NV Index, use the TPM-specific routine
112                     // to search the IN Index space.
113                     result = NvIndexIsAccessible(handle);
114                     break;
115                 case TPM_HT_PCR:
116                     // Any PCR handle that is unmarshaled successfully referenced
117                     // a PCR that is defined.
118                     break;
119 #if CC_AC_Send
120                 case TPM_HT_AC:
121                     // Use the TPM-specific routine to search for the AC
122                     result = AcIsAccessible(handle);
123                     break;
124 #endif
125                 default:
126                     // Any other handle type is a defect in the unmarshaling code.
127                     FAIL(FATAL_ERROR_INTERNAL);
128                     break;
129             }
130             if(result != TPM_RC_SUCCESS)
131             {
132                 if(result == TPM_RC_REFERENCE_H0)
133                     result = result + i;
134                 else
135                     result = RcSafeAddToResult(result, TPM_RC_H + g_rcIndex[i]);
136                 break;
137             }
138         }
139         return result;
140 }
```

9.4.3.2 EntityGetAuthValue()

This function is used to access the *authValue* associated with a handle. This function assumes that the handle references an entity that is accessible and the handle is not for a persistent objects. That is EntityGetLoadStatus() should have been called. Also, the accessibility of the *authValue* should have been verified by IsAuthValueAvailable().

This function copies the authorization value of the entity to *auth*.

Return Value	Meaning
count	number of bytes in the *authValue* with 0's stripped

```
141 UINT16
142 EntityGetAuthValue(
143     TPMI_DH_ENTITY    handle,        // IN: handle of entity
144     TPM2B_AUTH       *auth           // OUT: authValue of the entity
145     )
146 {
147     TPM2B_AUTH       *pAuth = NULL;
148
149     auth->t.size = 0;
150
151     switch(HandleGetType(handle))
```

```
152         {
153             case TPM_HT_PERMANENT:
154             {
155                 switch(handle)
156                 {
157                     case TPM_RH_OWNER:
158                         // ownerAuth for TPM_RH_OWNER
159                         pAuth = &gp.ownerAuth;
160                         break;
161                     case TPM_RH_ENDORSEMENT:
162                         // endorsementAuth for TPM_RH_ENDORSEMENT
163                         pAuth = &gp.endorsementAuth;
164                         break;
165                     // The ACT use platformAuth for auth
166                     FOR_EACH_ACT(CASE_ACT_HANDLE)
167                     case TPM_RH_PLATFORM:
168                         // platformAuth for TPM_RH_PLATFORM
169                         pAuth = &gc.platformAuth;
170                         break;
171                     case TPM_RH_LOCKOUT:
172                         // lockoutAuth for TPM_RH_LOCKOUT
173                         pAuth = &gp.lockoutAuth;
174                         break;
175                     case TPM_RH_NULL:
176                         // nullAuth for TPM_RH_NULL. Return 0 directly here
177                         return 0;
178                         break;
179 #ifdef  VENDOR_PERMANENT
180                     case VENDOR_PERMANENT:
181                         // vendor authorization value
182                         pAauth = &g_platformUniqueDetails;
183 #endif
184                     default:
185                         // If any other permanent handle is present it is
186                         // a code defect.
187                         FAIL(FATAL_ERROR_INTERNAL);
188                         break;
189                 }
190                 break;
191             }
192             case TPM_HT_TRANSIENT:
193                 // authValue for an object
194                 // A persistent object would have been copied into RAM
195                 // and would have an transient object handle here.
196             {
197                 OBJECT          *object;
198
199                 object = HandleToObject(handle);
200                 // special handling if this is a sequence object
201                 if(ObjectIsSequence(object))
202                 {
203                     pAuth = &((HASH_OBJECT *)object)->auth;
204                 }
205                 else
206                 {
207                     // Authorization is available only when the private portion of
208                     // the object is loaded.  The check should be made before
209                     // this function is called
210                     pAssert(object->attributes.publicOnly == CLEAR);
211                     pAuth = &object->sensitive.authValue;
212                 }
213             }
214             break;
215             case TPM_HT_NV_INDEX:
216                 // authValue for an NV index
217             {
```

```
218                 NV_INDEX        *nvIndex = NvGetIndexInfo(handle, NULL);
219                 pAssert(nvIndex != NULL);
220                 pAuth = &nvIndex->authValue;
221             }
222             break;
223         case TPM_HT_PCR:
224             // authValue for PCR
225             pAuth = PCRGetAuthValue(handle);
226             break;
227         default:
228             // If any other handle type is present here, then there is a defect
229             // in the unmarshaling code.
230             FAIL(FATAL_ERROR_INTERNAL);
231             break;
232     }
233     // Copy the authValue
234     MemoryCopy2B((TPM2B *)auth, (TPM2B *)pAuth, sizeof(auth->t.buffer));
235     MemoryRemoveTrailingZeros(auth);
236     return auth->t.size;
237 }
```

9.4.3.3 EntityGetAuthPolicy()

This function is used to access the *authPolicy* associated with a handle. This function assumes that the handle references an entity that is accessible and the handle is not for a persistent objects. That is EntityGetLoadStatus() should have been called. Also, the accessibility of the *authPolicy* should have been verified by IsAuthPolicyAvailable().

This function copies the authorization policy of the entity to *authPolicy*.

The return value is the hash algorithm for the policy.

```
238 TPMI_ALG_HASH
239 EntityGetAuthPolicy(
240     TPMI_DH_ENTITY    handle,        // IN: handle of entity
241     TPM2B_DIGEST     *authPolicy     // OUT: authPolicy of the entity
242     )
243 {
244     TPMI_ALG_HASH       hashAlg = TPM_ALG_NULL;
245     authPolicy->t.size = 0;
246
247     switch(HandleGetType(handle))
248     {
249         case TPM_HT_PERMANENT:
250             switch(handle)
251             {
252                 case TPM_RH_OWNER:
253                     // ownerPolicy for TPM_RH_OWNER
254                     *authPolicy = gp.ownerPolicy;
255                     hashAlg = gp.ownerAlg;
256                     break;
257                 case TPM_RH_ENDORSEMENT:
258                     // endorsementPolicy for TPM_RH_ENDORSEMENT
259                     *authPolicy = gp.endorsementPolicy;
260                     hashAlg = gp.endorsementAlg;
261                     break;
262                 case TPM_RH_PLATFORM:
263                     // platformPolicy for TPM_RH_PLATFORM
264                     *authPolicy = gc.platformPolicy;
265                     hashAlg = gc.platformAlg;
266                     break;
267                 case TPM_RH_LOCKOUT:
268                     // lockoutPolicy for TPM_RH_LOCKOUT
269                     *authPolicy = gp.lockoutPolicy;
```


 Copyright © TCG 2006-2020

```
270                         hashAlg = gp.lockoutAlg;
271                         break;
272 #define ACT_GET_POLICY(N)                                                          \
273                     case TPM_RH_ACT_##N:                                           \
274                         *authPolicy = go.ACT_##N.authPolicy;                       \
275                         hashAlg = go.ACT_##N.hashAlg;                              \
276                         break;
277                         // Get the policy for each implemented ACT
278                         FOR_EACH_ACT(ACT_GET_POLICY)
279                     default:
280                         hashAlg = TPM_ALG_ERROR;
281                         break;
282                 }
283                 break;
284         case TPM_HT_TRANSIENT:
285             // authPolicy for an object
286         {
287             OBJECT *object = HandleToObject(handle);
288             *authPolicy = object->publicArea.authPolicy;
289             hashAlg = object->publicArea.nameAlg;
290         }
291         break;
292         case TPM_HT_NV_INDEX:
293             // authPolicy for a NV index
294         {
295             NV_INDEX        *nvIndex = NvGetIndexInfo(handle, NULL);
296             pAssert(nvIndex != 0);
297             *authPolicy = nvIndex->publicArea.authPolicy;
298             hashAlg = nvIndex->publicArea.nameAlg;
299         }
300         break;
301         case TPM_HT_PCR:
302             // authPolicy for a PCR
303             hashAlg = PCRGetAuthPolicy(handle, authPolicy);
304             break;
305         default:
306             // If any other handle type is present it is a code defect.
307             FAIL(FATAL_ERROR_INTERNAL);
308             break;
309     }
310     return hashAlg;
311 }
```

9.4.3.4 EntityGetName()

This function returns the Name associated with a handle.

```
312 TPM2B_NAME *
313 EntityGetName(
314     TPMI_DH_ENTITY   handle,         // IN: handle of entity
315     TPM2B_NAME      *name            // OUT: name of entity
316     )
317 {
318     switch(HandleGetType(handle))
319     {
320         case TPM_HT_TRANSIENT:
321         {
322             // Name for an object
323             OBJECT      *object = HandleToObject(handle);
324             // an object with no nameAlg has no name
325             if(object->publicArea.nameAlg == TPM_ALG_NULL)
326                 name->b.size = 0;
327             else
328                 *name = object->name;
```

```
329                 break;
330             }
331         case TPM_HT_NV_INDEX:
332             // Name for a NV index
333             NvGetNameByIndexHandle(handle, name);
334             break;
335         default:
336             // For all other types, the handle is the Name
337             name->t.size = sizeof(TPM_HANDLE);
338             UINT32_TO_BYTE_ARRAY(handle, name->t.name);
339             break;
340     }
341     return name;
342 }
```

9.4.3.5 EntityGetHierarchy()

This function returns the hierarchy handle associated with an entity.

a) A handle that is a hierarchy handle is associated with itself.

b) An NV index belongs to TPM_RH_PLATFORM if TPMA_NV_PLATFORMCREATE, is SET, otherwise it belongs to TPM_RH_OWNER

c) An object handle belongs to its hierarchy.

```
343 TPMI_RH_HIERARCHY
344 EntityGetHierarchy(
345     TPMI_DH_ENTITY    handle          // IN :handle of entity
346     )
347 {
348     TPMI_RH_HIERARCHY        hierarchy = TPM_RH_NULL;
349
350     switch(HandleGetType(handle))
351     {
352         case TPM_HT_PERMANENT:
353             // hierarchy for a permanent handle
354             switch(handle)
355             {
356                 case TPM_RH_PLATFORM:
357                 case TPM_RH_ENDORSEMENT:
358                 case TPM_RH_NULL:
359                     hierarchy = handle;
360                     break;
361                 // all other permanent handles are associated with the owner
362                 // hierarchy. (should only be TPM_RH_OWNER and TPM_RH_LOCKOUT)
363                 default:
364                     hierarchy = TPM_RH_OWNER;
365                     break;
366             }
367             break;
368         case TPM_HT_NV_INDEX:
369             // hierarchy for NV index
370         {
371             NV_INDEX        *nvIndex = NvGetIndexInfo(handle, NULL);
372             pAssert(nvIndex != NULL);
373
374             // If only the platform can delete the index, then it is
375             // considered to be in the platform hierarchy, otherwise it
376             // is in the owner hierarchy.
377             if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV,
378                             PLATFORMCREATE))
379                 hierarchy = TPM_RH_PLATFORM;
380             else
381                 hierarchy = TPM_RH_OWNER;
```

 Copyright © TCG 2006-2020

```
382             }
383             break;
384             case TPM_HT_TRANSIENT:
385                 // hierarchy for an object
386             {
387                 OBJECT          *object;
388                 object = HandleToObject(handle);
389                 if(object->attributes.ppsHierarchy)
390                 {
391                     hierarchy = TPM_RH_PLATFORM;
392                 }
393                 else if(object->attributes.epsHierarchy)
394                 {
395                     hierarchy = TPM_RH_ENDORSEMENT;
396                 }
397                 else if(object->attributes.spsHierarchy)
398                 {
399                     hierarchy = TPM_RH_OWNER;
400                 }
401             }
402             break;
403             case TPM_HT_PCR:
404                 hierarchy = TPM_RH_OWNER;
405                 break;
406             default:
407                 FAIL(FATAL_ERROR_INTERNAL);
408                 break;
409         }
410         // this is unreachable but it provides a return value for the default
411         // case which makes the complier happy
412         return hierarchy;
413     }
```

9.5 Global.c

9.5.1 Description

This file will instance the TPM variables that are not stack allocated. Descriptions of global variables are in Global.h. There macro macro definitions that allows a variable to be instanced or simply defined as an external variable. When global.h is included from this .c file, GLOBAL_C is defined and values are instanced (and possibly initialized), but when global.h is included by any other file, they are simply defined as external values. DO NOT DEFINE GLOBAL_C IN ANY OTHER FILE.

NOTE: This is a change from previous implementations where Global.h just contained the extern declaration and values were instanced in this file. This change keeps the definition and instance in one file making maintenance easier. The instanced data will still be in the global.obj file.

The OIDs.h file works in a way that is similar to the Global.h with the definition of the values in OIDs.h such that they are instanced in global.obj. The macros that are defined in Global.h are used in OIDs.h in the same way as they are in Global.h.

9.5.2 Defines and Includes

```
1   #define GLOBAL_C
2   #include "Tpm.h"
3   #include "OIDs.h"
4   #if CC_CertifyX509
5   #   include "X509.h"
6   #endif // CC_CertifyX509
```

 Copyright © TCG 2006-2020

9.6 Handle.c

9.6.1 Description

This file contains the functions that return the type of a handle.

9.6.2 Includes

```
1 #include "Tpm.h"
```

9.6.3 Functions

9.6.3.1 HandleGetType()

This function returns the type of a handle which is the MSO of the handle.

```
2  TPM_HT
3  HandleGetType(
4      TPM_HANDLE       handle         // IN: a handle to be checked
5      )
6  {
7      // return the upper bytes of input data
8      return (TPM_HT)((handle & HR_RANGE_MASK) >> HR_SHIFT);
9  }
```

9.6.3.2 NextPermanentHandle()

This function returns the permanent handle that is equal to the input value or is the next higher value. If there is no handle with the input value and there is no next higher value, it returns 0:

```
10 TPM_HANDLE
11 NextPermanentHandle(
12     TPM_HANDLE       inHandle       // IN: the handle to check
13     )
14 {
15     // If inHandle is below the start of the range of permanent handles
16     // set it to the start and scan from there
17     if(inHandle < TPM_RH_FIRST)
18         inHandle = TPM_RH_FIRST;
19     // scan from input value until we find an implemented permanent handle
20     // or go out of range
21     for(; inHandle <= TPM_RH_LAST; inHandle++)
22     {
23         switch(inHandle)
24         {
25             case TPM_RH_OWNER:
26             case TPM_RH_NULL:
27             case TPM_RS_PW:
28             case TPM_RH_LOCKOUT:
29             case TPM_RH_ENDORSEMENT:
30             case TPM_RH_PLATFORM:
31             case TPM_RH_PLATFORM_NV:
32 #ifdef  VENDOR_PERMANENT
33             case VENDOR_PERMANENT:
34 #endif
35 // Each of the implemented ACT
36 #define ACT_IMPLEMENTED_CASE(N)                                              \
37             case TPM_RH_ACT_##N:
38
```

```
39                  FOR_EACH_ACT(ACT_IMPLEMENTED_CASE)
40
41                      return inHandle;
42                      break;
43                  default:
44                      break;
45              }
46          }
47          // Out of range on the top
48          return 0;
49      }
```

9.6.3.3 PermanentCapGetHandles()

This function returns a list of the permanent handles of PCR, started from *handle*. If *handle* is larger than the largest permanent handle, an empty list will be returned with *more* set to NO.

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
50      TPMI_YES_NO
51      PermanentCapGetHandles(
52          TPM_HANDLE        handle,         // IN: start handle
53          UINT32            count,          // IN: count of returned handles
54          TPML_HANDLE      *handleList      // OUT: list of handle
55          )
56      {
57          TPMI_YES_NO      more = NO;
58          UINT32           i;
59
60          pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);
61
62          // Initialize output handle list
63          handleList->count = 0;
64
65          // The maximum count of handles we may return is MAX_CAP_HANDLES
66          if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
67
68          // Iterate permanent handle range
69          for(i = NextPermanentHandle(handle);
70              i != 0; i = NextPermanentHandle(i + 1))
71          {
72              if(handleList->count < count)
73              {
74                  // If we have not filled up the return list, add this permanent
75                  // handle to it
76                  handleList->handle[handleList->count] = i;
77                  handleList->count++;
78              }
79              else
80              {
81                  // If the return list is full but we still have permanent handle
82                  // available, report this and stop iterating
83                  more = YES;
84                  break;
85              }
86          }
87          return more;
88      }
```

9.6.3.4 PermanentHandleGetPolicy()

This function returns a list of the permanent handles of PCR, started from *handle*. If *handle* is larger than the largest permanent handle, an empty list will be returned with *more* set to NO.

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
89  TPMI_YES_NO
90  PermanentHandleGetPolicy(
91      TPM_HANDLE            handle,          // IN: start handle
92      UINT32                count,           // IN: max count of returned handles
93      TPML_TAGGED_POLICY   *policyList       // OUT: list of handle
94      )
95  {
96      TPMI_YES_NO      more = NO;
97
98      pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);
99
100     // Initialize output handle list
101     policyList->count = 0;
102
103     // The maximum count of policies we may return is MAX_TAGGED_POLICIES
104     if(count > MAX_TAGGED_POLICIES)
105         count = MAX_TAGGED_POLICIES;
106
107     // Iterate permanent handle range
108     for(handle = NextPermanentHandle(handle);
109         handle != 0;
110         handle = NextPermanentHandle(handle + 1))
111     {
112         TPM2B_DIGEST     policyDigest;
113         TPM_ALG_ID       policyAlg;
114         // Check to see if this permanent handle has a policy
115         policyAlg = EntityGetAuthPolicy(handle, &policyDigest);
116         if(policyAlg == TPM_ALG_ERROR)
117            continue;
118         if(policyList->count < count)
119         {
120             // If we have not filled up the return list, add this
121             // policy to the list;
122             policyList->policies[policyList->count].handle = handle;
123             policyList->policies[policyList->count].policyHash.hashAlg = policyAlg;
124             MemoryCopy(&policyList->policies[policyList->count].policyHash.digest,
125                        policyDigest.t.buffer, policyDigest.t.size);
126             policyList->count++;
127         }
128         else
129         {
130             // If the return list is full but we still have permanent handle
131             // available, report this and stop iterating
132             more = YES;
133             break;
134         }
135     }
136     return more;
137 }
```

9.7 IoBuffers.c

9.7.1 Includes and Data Definitions

This definition allows this module to **see** the values that are private to this module but kept in Global.c for ease of state migration.

```
1  #define IO_BUFFER_C
2  #include "Tpm.h"
3  #include "IoBuffers_fp.h"
```

9.7.2 Buffers and Functions

These buffers are set aside to hold command and response values. In this implementation, it is not guaranteed that the code will stop accessing the *s_actionInputBuffer* before starting to put values in the *s_actionOutputBuffer* so different buffers are required.

9.7.2.1 MemoryIoBufferAllocationReset()

This function is used to reset the allocation of buffers.

```
4  void
5  MemoryIoBufferAllocationReset(
6      void
7  )
8  {
9      s_actionIoAllocation = 0;
10 }
```

9.7.2.2 MemoryIoBufferZero()

Function zeros the action I/O buffer at the end of a command. Calling this is not mandatory for proper functionality.

```
11 void
12 MemoryIoBufferZero(
13     void
14 )
15 {
16     memset(s_actionIoBuffer, 0, s_actionIoAllocation);
17 }
```

9.7.2.3 MemoryGetInBuffer()

This function returns the address of the buffer into which the command parameters will be unmarshaled in preparation for calling the command actions.

```
18 BYTE *
19 MemoryGetInBuffer(
20     UINT32           size           // Size, in bytes, required for the input
21                                     // unmarshaling
22     )
23 {
24     pAssert(size <= sizeof(s_actionIoBuffer));
25     // In this implementation, a static buffer is set aside for the command action
26     // buffers. The buffer is shared between input and output. This is because
27     // there is no need to allocate for the worst case input and worst case output
```

 Copyright © TCG 2006-2020

```
28      // at the same time.
29      // Round size up
30      #define UoM  (sizeof(s_actionIoBuffer[0]))
31      size = (size + (UoM - 1)) & (UINT32_MAX - (UoM - 1));
32      memset(s_actionIoBuffer, 0, size);
33      s_actionIoAllocation = size;
34      return (BYTE *)&s_actionIoBuffer[0];
35  }
```

9.7.2.4 MemoryGetOutBuffer()

This function returns the address of the buffer into which the command action code places its output values.

```
36  BYTE *
37  MemoryGetOutBuffer(
38      UINT32          size            // required size of the buffer
39      )
40  {
41      BYTE        *retVal = (BYTE *)(&s_actionIoBuffer[s_actionIoAllocation / UoM]);
42      pAssert((size + s_actionIoAllocation) < (sizeof(s_actionIoBuffer)));
43      // In this implementation, a static buffer is set aside for the command action
44      // output buffer.
45      memset(retVal, 0, size);
46      s_actionIoAllocation += size;
47      return retVal;
48  }
```

9.7.2.5 IsLabelProperlyFormatted()

This function checks that a label is a null-terminated string.

NOTE: this function is here because there was no better place for it.

Return Value	Meaning
TRUE(1)	string is null terminated
FALSE(0)	string is not null terminated

```
49  BOOL
50  IsLabelProperlyFormatted(
51      TPM2B           *x
52      )
53  {
54      return (((x)->size == 0) || ((x)->buffer[(x)->size - 1] == 0));
55  }
```

 Copyright © TCG 2006-2020

9.8 Locality.c

9.8.1 Includes

```
1    #include "Tpm.h"
```

9.8.2 LocalityGetAttributes()

This function will convert a locality expressed as an integer into TPMA_LOCALITY form.

The function returns the locality attribute.

```
2    TPMA_LOCALITY
3    LocalityGetAttributes(
4        UINT8           locality        // IN: locality value
5        )
6    {
7        TPMA_LOCALITY           locality_attributes;
8        BYTE                   *localityAsByte = (BYTE *)&locality_attributes;
9
10       MemorySet(&locality_attributes, 0, sizeof(TPMA_LOCALITY));
11       switch(locality)
12       {
13           case 0:
14               SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_ZERO);
15               break;
16           case 1:
17               SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_ONE);
18               break;
19           case 2:
20               SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_TWO);
21               break;
22           case 3:
23               SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_THREE);
24               break;
25           case 4:
26               SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_FOUR);
27               break;
28           default:
29               pAssert(locality > 31);
30               *localityAsByte = locality;
31               break;
32       }
33       return locality_attributes;
34   }
```

 Copyright © TCG 2006-2020

9.9 Manufacture.c

9.9.1 Description

This file contains the function that performs the **manufacturing** of the TPM in a simulated environment. These functions should not be used outside of a manufacturing or simulation environment.

9.9.2 Includes and Data Definitions

```
1  #define MANUFACTURE_C
2  #include "Tpm.h"
3  #include "TpmSizeChecks_fp.h"
```

9.9.3 Functions

9.9.3.1 TPM_Manufacture()

This function initializes the TPM values in preparation for the TPM's first use. This function will fail if previously called. The TPM can be re-manufactured by calling TPM_Teardown() first and then calling this function again.

Return Value	Meaning
-1	failure
0	success
1	manufacturing process previously performed

```
4  LIB_EXPORT int
5  TPM_Manufacture(
6      int                firstTime        // IN: indicates if this is the first call from
7                                          //     main()
8      )
9  {
10     TPM_SU             orderlyShutdown;
11
12 #if RUNTIME_SIZE_CHECKS
13     // Call the function to verify the sizes of values that result from different
14     // compile options.
15     if(!TpmSizeChecks())
16         return -1;
17 #endif
18 #if LIBRARY_COMPATIBILITY_CHECK
19 // Make sure that the attached library performs as expected.
20     if(!MathLibraryCompatibilityCheck())
21         return -1;
22 #endif
23
24     // If TPM has been manufactured, return indication.
25     if(!firstTime && g_manufactured)
26         return 1;
27
28     // Do power on initializations of the cryptographic libraries.
29     CryptInit();
30
31     s_DAPendingOnNV = FALSE;
32
33     // initialize NV
34     NvManufacture();
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
35
36      // Clear the magic value in the DRBG state
37      go.drbgState.magic = 0;
38
39      CryptStartup(SU_RESET);
40
41      // default configuration for PCR
42      PCRSimStart();
43
44      // initialize pre-installed hierarchy data
45      // This should happen after NV is initialized because hierarchy data is
46      // stored in NV.
47      HierarchyPreInstall_Init();
48
49      // initialize dictionary attack parameters
50      DAPreInstall_Init();
51
52      // initialize PP list
53      PhysicalPresencePreInstall_Init();
54
55      // initialize command audit list
56      CommandAuditPreInstall_Init();
57
58      // first start up is required to be Startup(CLEAR)
59      orderlyShutdown = TPM_SU_CLEAR;
60      NV_WRITE_PERSISTENT(orderlyState, orderlyShutdown);
61
62      // initialize the firmware version
63      gp.firmwareV1 = FIRMWARE_V1;
64  #ifdef FIRMWARE_V2
65      gp.firmwareV2 = FIRMWARE_V2;
66  #else
67      gp.firmwareV2 = 0;
68  #endif
69      NV_SYNC_PERSISTENT(firmwareV1);
70      NV_SYNC_PERSISTENT(firmwareV2);
71
72      // initialize the total reset counter to 0
73      gp.totalResetCount = 0;
74      NV_SYNC_PERSISTENT(totalResetCount);
75
76      // initialize the clock stuff
77      go.clock = 0;
78      go.clockSafe = YES;
79
80      NvWrite(NV_ORDERLY_DATA, sizeof(ORDERLY_DATA), &go);
81
82      // Commit NV writes.  Manufacture process is an artificial process existing
83      // only in simulator environment and it is not defined in the specification
84      // that what should be the expected behavior if the NV write fails at this
85      // point.  Therefore, it is assumed the NV write here is always success and
86      // no return code of this function is checked.
87      NvCommit();
88
89      g_manufactured = TRUE;
90
91      return 0;
92  }
```

9.9.3.2 TPM_TearDown()

This function prepares the TPM for re-manufacture. It should not be implemented in anything other than a simulated TPM.

In this implementation, all that is needs is to stop the cryptographic units and set a flag to indicate that the TPM can be re-manufactured. This should be all that is necessary to start the manufacturing process again.

Return Value	Meaning
0	success
1	TPM not previously manufactured

```
93   LIB_EXPORT int
94   TPM_TearDown(
95       void
96       )
97   {
98       g_manufactured = FALSE;
99       return 0;
100  }
```

9.9.3.3 TpmEndSimulation()

This function is called at the end of the simulation run. It is used to provoke printing of any statistics that might be needed.

```
101  LIB_EXPORT void
102  TpmEndSimulation(
103      void
104      )
105  {
106  #if SIMULATION
107      HashLibSimulationEnd();
108      SymLibSimulationEnd();
109      MathLibSimulationEnd();
110  #if ALG_RSA
111      RsaSimulationEnd();
112  #endif
113  #if ALG_ECC
114      EccSimulationEnd();
115  #endif
116  #endif // SIMULATION
117  }
```

9.10 Marshal.c

9.10.1 Introduction

This file contains the marshaling and unmarshaling code.

The marshaling and unmarshaling code and function prototypes are not listed, as the code is repetitive, long, and not very useful to read. Examples of a few unmarshaling routines are provided. Most of the others are similar.

Depending on the table header flags, a type will have an unmarshaling routine and a marshaling routine The table header flags that control the generation of the unmarshaling and marshaling code are delimited by angle brackets ("<>") in the table header. If no brackets are present, then both unmarshaling and marshaling code is generated (i.e., generation of both marshaling and unmarshaling code is the default).

9.10.2 Unmarshal and Marshal a Value

In TPM 2.0 Part 2, a TPMI_DI_OBJECT is defined by this table:

Table xxx — Definition of (TPM_HANDLE) TPMI_DH_OBJECT Type

Values	Comments
{TRANSIENT_FIRST:TRANSIENT_LAST}	allowed range for transient objects
{PERSISTENT_FIRST:PERSISTENT_LAST}	allowed range for persistent objects
+TPM_RH_NULL	the null handle
#TPM_RC_VALUE	

This generates the following unmarshaling code:

```
1   TPM_RC
2   TPMI_DH_OBJECT_Unmarshal(TPMI_DH_OBJECT *target, BYTE **buffer, INT32 *size,
3                            BOOL flag)
4   {
5       TPM_RC    result;
6       result = TPM_HANDLE_Unmarshal((TPM_HANDLE *)target, buffer, size);
7       if(result != TPM_RC_SUCCESS)
8           return result;
9       if(*target == TPM_RH_NULL)
10      {
11          if(flag)
12              return TPM_RC_SUCCESS;
13          else
14              return TPM_RC_VALUE;
15      }
16      if(((*target < TRANSIENT_FIRST) || (*target > TRANSIENT_LAST))
17        &&((*target < PERSISTENT_FIRST) || (*target > PERSISTENT_LAST)))
18              return TPM_RC_VALUE;
19      return TPM_RC_SUCCESS;
20  }
```

and the following marshaling code:

NOTE The marshaling code does not do parameter checking, as the TPM is the source of the marshaling data.

```
1   UINT16
2   TPMI_DH_OBJECT_Marshal(TPMI_DH_OBJECT *source, BYTE **buffer, INT32 *size)
```

 Copyright © TCG 2006-2020

```
3   {
4       return UINT32_Marshal((UINT32 *)source, buffer, size);
5   }
```

An additional script is used to do the work that might be done by a linker or globally optimizing compiler. It searches for functions like TPMI_DH_OBJECT_Marshal() that do nothing but call another function and replaces the function with a #define.

```
6   #define TPMI_DH_OBJECT_Marshal(source, buffer, size)      \
7         UINT32_Marshal((UINT32 *)source, buffer, size)
```

When replacing the function with a #define, the #define is placed in marshal_fp.h and the function body is removed from marshal.c.

9.10.3 Unmarshal and Marshal a Union

In TPM 2.0 Part 2, a TPMU_PUBLIC_PARMS union is defined by:

Table xxx — Definition of TPMU_PUBLIC_PARMS Union <IN/OUT, S>

Parameter	Type	Selector	Description
keyedHash	TPMS_KEYEDHASH_PARMS	TPM_ALG_KEYEDHASH	sign | encrypt | neither
symDetail	TPMT_SYM_DEF_OBJECT	TPM_ALG_SYMCIPHER	a symmetric block cipher
rsaDetail	TPMS_RSA_PARMS	TPM_ALG_RSA	decrypt + sign
eccDetail	TPMS_ECC_PARMS	TPM_ALG_ECC	decrypt + sign
asymDetail	TPMS_ASYM_PARMS		common scheme structure for RSA and ECC keys
NOTE The Description column indicates which of TPMA_OBJECT.*decrypt* or TPMA_OBJECT.*sign* may be set. "+" indicates that both may be set but one shall be set. "|" indicates the optional settings.			

From this table, the following unmarshaling code is generated.

```
1   TPM_RC
2   TPMU_PUBLIC_PARMS_Unmarshal(TPMU_PUBLIC_PARMS *target, BYTE **buffer, INT32 *size,
3                               UINT32 selector)
4   {
5       switch(selector) {
6   #if ALG_KEYEDHASH
7           case TPM_ALG_KEYEDHASH:
8               return TPMS_KEYEDHASH_PARMS_Unmarshal(
9                          (TPMS_KEYEDHASH_PARMS *)&(target->keyedHash), buffer, size);
10  #endif
11  #if ALG_SYMCIPHER
12          case TPM_ALG_SYMCIPHER:
13              return TPMT_SYM_DEF_OBJECT_Unmarshal(
14                     (TPMT_SYM_DEF_OBJECT *)&(target->symDetail), buffer, size, FALSE);
15  #endif
16  #if ALG_RSA
17          case TPM_ALG_RSA:
18              return TPMS_RSA_PARMS_Unmarshal(
19                              (TPMS_RSA_PARMS *)&(target->rsaDetail), buffer, size);
20  #endif
21  #if ALG_ECC
22          case TPM_ALG_ECC:
23              return TPMS_ECC_PARMS_Unmarshal(
24                              (TPMS_ECC_PARMS *)&(target->eccDetail), buffer, size);
25  #endif
26      }
```

 Copyright © TCG 2006-2020

```
27      return TPM_RC_SELECTOR;
28  }
```

NOTE The #if/#endif directives are added whenever a value is dependent on an algorithm ID so that removing the algorithm definition will remove the related code.

The marshaling code for the union is:

```
1   UINT16
2   TPMU_PUBLIC_PARMS_Marshal(TPMU_PUBLIC_PARMS *source, BYTE **buffer, INT32 *size,
3                              UINT32 selector)
4   {
5       switch(selector) {
6   #if ALG_KEYEDHASH
7           case TPM_ALG_KEYEDHASH:
8               return TPMS_KEYEDHASH_PARMS_Marshal(
9                             (TPMS_KEYEDHASH_PARMS *)&(source->keyedHash), buffer, size);
10  #endif
11  #if ALG_SYMCIPHER
12          case TPM_ALG_SYMCIPHER:
13              return TPMT_SYM_DEF_OBJECT_Marshal(
14                             (TPMT_SYM_DEF_OBJECT *)&(source->symDetail), buffer, size);
15  #endif
16  #if ALG_RSA
17          case TPM_ALG_RSA:
18              return TPMS_RSA_PARMS_Marshal(
19                              (TPMS_RSA_PARMS *)&(source->rsaDetail), buffer, size);
20  #endif
21  #if ALG_ECC
22          case TPM_ALG_ECC:
23              return TPMS_ECC_PARMS_Marshal(
24                               (TPMS_ECC_PARMS *)&(source->eccDetail), buffer, size);
25  #endif
26      }
27      assert(1);
28      return 0;
29  }
```

For the marshaling and unmarshaling code, a value in the structure containing the union provides the value used for *selector*. The example in the next section illustrates this.

 Copyright © TCG 2006-2020

9.10.4 Unmarshal and Marshal a Structure

In TPM 2.0 Part 2, the TPMT_PUBLIC structure is defined by:

Table xxx — Definition of TPMT_PUBLIC Structure

Parameter	Type	Description
type	TPMI_ALG_PUBLIC	"algorithm" associated with this object
nameAlg	+TPMI_ALG_HASH	algorithm used for computing the Name of the object NOTE The "+" indicates that the instance of a TPMT_PUBLIC may have a "+" to indicate that the nameAlg may be TPM_ALG_NULL.
objectAttributes	TPMA_OBJECT	attributes that, along with *type*, determine the manipulations of this object
authPolicy	TPM2B_DIGEST	optional policy for using this key The policy is computed using the *nameAlg* of the object. NOTE shall be the Empty Buffer if no authorization policy is present
[type]parameters	TPMU_PUBLIC_PARMS	the algorithm or structure details
[type]unique	TPMU_PUBLIC_ID	the unique identifier of the structure For an asymmetric key, this would be the public key.

This structure is tagged (the first value indicates the structure type), and that tag is used to determine how the parameters and unique fields are unmarshaled and marshaled. The use of the type for specifying the union selector is emphasized below.

The unmarshaling code for the structure in the table above is:

```
1  TPM_RC
2  TPMT_PUBLIC_Unmarshal(TPMT_PUBLIC *target, BYTE **buffer, INT32 *size, BOOL flag)
3  {
4      TPM_RC     result;
5      result = TPMI_ALG_PUBLIC_Unmarshal((TPMI_ALG_PUBLIC *)&(target->type),
6                                         buffer, size);
7      if(result != TPM_RC_SUCCESS)
8          return result;
9      result = TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH *)&(target->nameAlg),
10                                      buffer, size, flag);
11     if(result != TPM_RC_SUCCESS)
12         return result;
13     result = TPMA_OBJECT_Unmarshal((TPMA_OBJECT *)&(target->objectAttributes),
14                                    buffer, size);
15     if(result != TPM_RC_SUCCESS)
16         return result;
17     result = TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST *)&(target->authPolicy),
18                                     buffer, size);
19     if(result != TPM_RC_SUCCESS)
20         return result;
21
22     result = TPMU_PUBLIC_PARMS_Unmarshal((TPMU_PUBLIC_PARMS *)&(target->parameters),
23                                          buffer, size, (UINT32)target->type);
24     if(result != TPM_RC_SUCCESS)
25         return result;
26
27     result = TPMU_PUBLIC_ID_Unmarshal((TPMU_PUBLIC_ID *)&(target->unique),
28                                       buffer, size, (UINT32)target->type);
29     if(result != TPM_RC_SUCCESS)
30         return result;
31
32     return TPM_RC_SUCCESS;
33 }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

The marshaling code for the TPMT_PUBLIC structure is:

```
1   UINT16
2   TPMT_PUBLIC_Marshal(TPMT_PUBLIC *source, BYTE **buffer, INT32 *size)
3   {
4       UINT16    result = 0;
5       result = (UINT16)(result + TPMI_ALG_PUBLIC_Marshal(
6                                  (TPMI_ALG_PUBLIC *)&(source->type), buffer, size));
7       result = (UINT16)(result + TPMI_ALG_HASH_Marshal(
8                                 (TPMI_ALG_HASH *)&(source->nameAlg), buffer, size))
9   ;
10      result = (UINT16)(result + TPMA_OBJECT_Marshal(
11                            (TPMA_OBJECT *)&(source->objectAttributes), buffer, size));
12
13      result = (UINT16)(result + TPM2B_DIGEST_Marshal(
14                                (TPM2B_DIGEST *)&(source->authPolicy), buffer, size));
15
16      result = (UINT16)(result + TPMU_PUBLIC_PARMS_Marshal(
17                            (TPMU_PUBLIC_PARMS *)&(source->parameters), buffer, size,
18                                                           (UINT32)source->type));
19
20      result = (UINT16)(result + TPMU_PUBLIC_ID_Marshal(
21                                (TPMU_PUBLIC_ID *)&(source->unique), buffer, size,
22                                                           (UINT32)source->type));
23
24      return result;
25  }
```

9.10.5 Unmarshal and Marshal an Array

In TPM 2.0 Part 2, the TPML_DIGEST is defined by:

Table xxx — Definition of TPML_DIGEST Structure

Parameter	Type	Description
count {2:}	UINT32	number of digests in the list, minimum is two
digests[count]{:8}	TPM2B_DIGEST	a list of digests For TPM2_PolicyOR(), all digests will have been computed using the digest of the policy session. For TPM2_PCR_Read(), each digest will be the size of the digest for the bank containing the PCR.
#TPM_RC_SIZE		response code when count is not at least two or is greater than 8

The *digests* parameter is an array of up to *count* structures (TPM2B_DIGESTS). The auto-generated code to Unmarshal this structure is:

```
1   TPM_RC
2   TPML_DIGEST_Unmarshal(TPML_DIGEST *target, BYTE **buffer, INT32 *size)
3   {
4       TPM_RC    result;
5       result = UINT32_Unmarshal((UINT32 *)&(target->count), buffer, size);
6       if(result != TPM_RC_SUCCESS)
7           return result;
8
9       if( (target->count < 2))     // This check is triggered by the {2:} notation
10                                    // on 'count'
11          return TPM_RC_SIZE;
12
13      if((target->count) > 8)      // This check is triggered by the {:8} notation
```

 Copyright © TCG 2006-2020

```
14                                     // on 'digests'.
15          return TPM_RC_SIZE;
16
17      result = TPM2B_DIGEST_Array_Unmarshal((TPM2B_DIGEST *)(target->digests),
18                                             buffer, size, (INT32)(target->count));
19      if(result != TPM_RC_SUCCESS)
20          return result;
21
22      return TPM_RC_SUCCESS;
23  }
```

The routine unmarshals a *count* value and passes that value to a routine that unmarshals an array of TPM2B_DIGEST values. The unmarshaling code for the array is:

```
1   TPM_RC
2   TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
3                                INT32 count)
4   {
5       TPM_RC    result;
6       INT32 i;
7       for(i = 0; i < count; i++) {
8           result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
9           if(result != TPM_RC_SUCCESS)
10              return result;
11      }
12      return TPM_RC_SUCCESS;
13  }
14
```

Marshaling of the TPML_DIGEST uses a similar scheme with a structure specifying the number of elements in an array and a subsequent call to a routine to marshal an array of that type.

```
1   UINT16
2   TPML_DIGEST_Marshal(TPML_DIGEST *source, BYTE **buffer, INT32 *size)
3   {
4       UINT16    result = 0;
5       result = (UINT16)(result + UINT32_Marshal((UINT32 *)&(source->count), buffer,
6                                                                              size));
7       result = (UINT16)(result + TPM2B_DIGEST_Array_Marshal(
8                                  (TPM2B_DIGEST *)(source->digests), buffer, size,
9                                  (INT32)(source->count)));
10
11      return result;
12  }
```

The marshaling code for the array is:

```
1   TPM_RC
2   TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
3                               INT32 count)
4   {
5       TPM_RC    result;
6       INT32 i;
7       for(i = 0; i < count; i++) {
8           result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
9           if(result != TPM_RC_SUCCESS)
10              return result;
11      }
12      return TPM_RC_SUCCESS;
13  }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

9.10.6 TPM2B Handling

A TPM2B structure is handled as a special case. The unmarshaling code is similar to what is shown in 9.10.5 but the unmarshaling/marshaling is to a union element. Each TPM2B is a union of two sized buffers, one of which is type specific (the 't' element) and the other is a generic value (the 'b' element). This allows each of the TPM2B structures to have some inheritance property with all other TPM2B. The purpose is to allow functions that have parameters that can be any TPM2B structure while allowing other functions to be specific about the type of the TPM2B that is used. When the generic structure is allowed, the input parameter would use the 'b' element and when the type-specific structure is required, the 't' element is used.

When marshaling a TPM2B where the second member is a BYTE array, the size parameter indicates the size of the array. The second member can also be a structure. In this case, the caller does not prefill the size member. The marshaling code must marshal the structure and then back fill the calculated size.

Table xxx — Definition of TPM2B_EVENT Structure

Parameter	Type	Description
size	UINT16	Size of the operand
buffer [size] {:1024}	BYTE	The operand

```
1   TPM_RC
2   TPM2B_EVENT_Unmarshal(TPM2B_EVENT *target, BYTE **buffer, INT32 *size)
3   {
4       TPM_RC    result;
5       result = UINT16_Unmarshal((UINT16 *)&(target->t.size), buffer, size);
6       if(result != TPM_RC_SUCCESS)
7           return result;
8       // if size equal to 0, the rest of the structure is a zero buffer
9       //  so stop processing
10      if(target->t.size == 0)
11          return TPM_RC_SUCCESS;
12      if((target->t.size) > 1024)     // This check is triggered by the {:1024}
13                                      // notation on 'buffer'
14          return TPM_RC_SIZE;
15      result = BYTE_Array_Unmarshal((BYTE *)(target->t.buffer), buffer, size,
16                                    (INT32)(target->t.size));
17      if(result != TPM_RC_SUCCESS)
18          return result;
19      return TPM_RC_SUCCESS;
20  }
```

using these structure definitions:

```
1   typedef union {
2       struct {
3           UINT16         size;
4           BYTE           buffer[1024];
5       }            t;
6       TPM2B        b;
7   } TPM2B_EVENT;
```

9.10.7 Table Marshal Headers

9.10.7.1 TableMarshal.h

```
1   #ifndef _TABLE_DRIVEN_MARSHAL_H_
2   #define _TABLE_DRIVEN_MARSHAL_H_
```

 Copyright © TCG 2006-2020

These are the basic unmarshaling types. This is in the first byte of each structure descriptor that is passed to Marshal()/Unmarshal() for processing.

```
 3  #define UINT_MTYPE          0
 4  #define VALUES_MTYPE        (UINT_MTYPE + 1)
 5  #define TABLE_MTYPE         (VALUES_MTYPE + 1)
 6  #define MIN_MAX_MTYPE       (TABLE_MTYPE + 1)
 7  #define ATTRIBUTES_MTYPE    (MIN_MAX_MTYPE + 1)
 8  #define STRUCTURE_MTYPE     (ATTRIBUTES_MTYPE + 1)
 9  #define TPM2B_MTYPE         (STRUCTURE_MTYPE + 1)
10  #define TPM2BS_MTYPE        (TPM2B_MTYPE + 1)
11  #define LIST_MTYPE          (TPM2BS_MTYPE + 1) // TPML
12  #define ERROR_MTYPE         (LIST_MTYPE + 1)
13  #define NULL_MTYPE          (ERROR_MTYPE + 1)
14  #define COMPOSITE_MTYPE     (NULL_MTYPE + 1)
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

9.10.7.1.1.1 The Marshal Index

A structure is used to hold the values that guide the marshaling/unmarshaling of each of the types. Each structure has a name and an address. For a structure to define a TPMS_name, the structure is a TPMS_name_MARSHAL_STRUCT and its index is TPMS_name_MARSHAL_INDEX. So, to get the proper structure, use the associated marshal index. The marshal index is passed to Marshal() or Unmarshal() and those functions look up the proper structure.

To handle structures that allow a null value, the upper bit of each marshal index indicates if the null value is allowed. This is the NULL_FLAG. It is defined in TableMarshalIndex.h because it is needed by code outside of the marshaling code. A structure will have a list of marshal indexes to indicate what to unmarshal. When that index appears in a structure/union, the value will contain a flag to indicate that the NULL_FLAG should be SET on the call to Unmarshal() to unmarshal the type. The caller simply takes the entry and passes it to Unmarshal() to indicate that the NULL_FLAG is SET. There is also the opportunity to SET the NULL_FLAG in the called structure if the NULL_FLAG was set in the call to the calling structure. This is indicated by:

```
15  #define NULL_MASK       ~(NULL_FLAG)
```

When looking up the value to marshal, the upper two bits of the marshal index are masked to yield the actual index.

```
16  typedef unsigned int        uint;
```

 Copyright © TCG 2006-2020

9.10.7.1.1.2 Modifier Octet Values

These are in used in anything that is an integer value. Theses would not be in structure modifier bytes (they would be used in values in structures but not the STRUCTURE_MTYPE header.

```
17  #define ONE_BYTES           (0)
18  #define TWO_BYTES           (1)
19  #define FOUR_BYTES          (2)
20  #define EIGHT_BYTES         (3)
21  #define SIZE_MASK           (0x3)
22  #define IS_SIGNED           (1 << 2)     // when the unmarshaled type is a signed value
23  #define SIGNED_MASK         (SIZE_MASK | IS_SIGNED)
```

This may be used for any type except a UINT_MTYPE

```
24  #define TAKES_NULL          (1 << 7)     // when the type takes a null
```

When referencing a structure, this flag indicates if a null is to be propagated to the referenced structure or type.

```
25  #define PROPAGATE_SHIFT     7
26  #define PROPAGATE_NULL      (1 << PROPAGATE_SHIFT)
```

Can be used in min-max or table structures.

```
27  #define HAS_BITS            (1 << 6)     // when bit mask is present
```

In a union, we need to know if this is a union of constant arrays.

```
28  #define IS_ARRAY_UNION      (1 << 6)
```

In a TPM2BS_MTYPE

```
29  #define SIZE_EQUAL          (1 << 6)
```

Right now, there are two spare bits in the modifiers field. Within the descriptor word of each entry in a StructMarsh_mst(), there is a selector field to determine which of the sub-types the entry represents and a field that is used to reference another structure entry. This is a 6-bit field allowing a structure to have 64 entries. This should be more than enough as the structures are not that long. As of now, only 10-bits of the descriptor word leaving room for expansion. These are the values used in a STRUCTURE_MTYPE to identify the sub-type of the thing being processed

```
30  #define SIMPLE_STYPE                0
31  #define UNION_STYPE                 1
32  #define ARRAY_STYPE                 2
```

The code used GET_ to get the element type and the compiler uses SET_ to initialize the value. The element type is the three bits (2:0).

```
33  #define GET_ELEMENT_TYPE(val)       (val & 7)
34  #define SET_ELEMENT_TYPE(val)       (val & 7)
```

When an entry is an array or union, this references the structure entry that contains the dimension or selector value. The code then uses this number to look up the structure entry for that element to find out what it and where is it in memory. When this is not a reference, it is a simple type and it could be used as an array value or a union selector. When a simple value, this field contains the size of the associated value (ONE_BYTES, TWO_BYTES ...) The entry size/number is 6 bits (13:8).

 Copyright © TCG 2006-2020

```
35  #define GET_ELEMENT_NUMBER(val)      (((val) >> 8) & 0x3F)
36  #define SET_ELEMENT_NUMBER(val)      (((val) & 0x3F) << 8)
37  #define GET_ELEMENT_SIZE(val)        GET_ELEMENT_NUMBER(val)
38  #define SET_ELEMENT_SIZE(val)        SET_ELEMENT_NUMBER(val)
```

This determines if the null flag is propagated to this type. If generate, the NULL_FLAG is SET in the index value. This flag is one bit (7)

```
39  #define ELEMENT_PROPAGATE            (1 << PROPAGATE_SHIFT)
40  #define INDEX_MASK                   ((UINT16)NULL_MASK)
```

This is used in all bit-field checks. These are used when a value that is checked is conditional (dependent on the compilation). For example, if AES_128 is (NO), then the bit associated with AES_128 will be 0. In some cases, the bit value is found by checking that the input is within the range of the table, and then using the (val - min) value to index the bit. This would be used when verifying that a particular algorithm is implemented. In other cases, there is a bit for each value in a table. For example, if checking the key sizes, there is a list of possible key sizes allowed by the algorithm registry and a bit field to indicate if that key size is allowed in the implementation. The smallest bit field has 32-bits 32-bits because it is implemented as part of the *values* array of the structures that allow bit fields.

```
41  #define IS_BIT_SET32(bit, bits)                                              \
42                          ((((UINT32 *)bits)[bit >> 5] & (1 << (bit & 0x1F))) != 0)
```

For a COMPOSITE_MTYPE, the qualifiers byte has an element size and count.

```
43  #define SET_ELEMENT_COUNT(count)     ((count & 0x1F) << 3)
44  #define GET_ELEMENT_COUNT(val)       ((val  >> 3) & 0x1F)
45  #endif // _TABLE_DRIVEN_MARSHAL_H_
```

9.10.7.2 TableMarshalData.h

```
1   #ifndef _Table_Marshal_Data_
2   #define _Table_Marshal_Data_
```

The datatype descriptions for each type if needed in addition to the default types.

```
3   typedef const struct TPM_ECC_CURVE_mst {
4       UINT8       marshalType;
5       UINT8       modifiers;
6       UINT8       errorCode;
7       UINT32      values[4];
8   } TPM_ECC_CURVE_mst;
9   typedef const struct TPM_CLOCK_ADJUST_mst {
10      UINT8       marshalType;
11      UINT8       modifiers;
12      UINT8       errorCode;
13      UINT32      values[2];
14  } TPM_CLOCK_ADJUST_mst;
15  typedef const struct TPM_EO_mst {
16      UINT8       marshalType;
17      UINT8       modifiers;
18      UINT8       errorCode;
19      UINT32      values[2];
20  } TPM_EO_mst;
21  typedef const struct TPM_SU_mst {
22      UINT8         marshalType;
23      UINT8         modifiers;
24      UINT8         errorCode;
25      UINT8         entries;
26      UINT32        values[2];
27  } TPM_SU_mst;
```

 Copyright © TCG 2006-2020

```
28   typedef const struct TPM_SE_mst {
29       UINT8           marshalType;
30       UINT8           modifiers;
31       UINT8           errorCode;
32       UINT8           entries;
33       UINT32          values[3];
34   } TPM_SE_mst;
35   typedef const struct TPM_CAP_mst {
36       UINT8           marshalType;
37       UINT8           modifiers;
38       UINT8           errorCode;
39       UINT8           ranges;
40       UINT8           singles;
41       UINT32          values[3];
42   } TPM_CAP_mst;
43   typedef const struct TPMI_YES_NO_mst {
44       UINT8           marshalType;
45       UINT8           modifiers;
46       UINT8           errorCode;
47       UINT8           entries;
48       UINT32          values[2];
49   } TPMI_YES_NO_mst;
50   typedef const struct TPMI_DH_OBJECT_mst {
51       UINT8           marshalType;
52       UINT8           modifiers;
53       UINT8           errorCode;
54       UINT8           ranges;
55       UINT8           singles;
56       UINT32          values[5];
57   } TPMI_DH_OBJECT_mst;
58   typedef const struct TPMI_DH_PARENT_mst {
59       UINT8           marshalType;
60       UINT8           modifiers;
61       UINT8           errorCode;
62       UINT8           ranges;
63       UINT8           singles;
64       UINT32          values[8];
65   } TPMI_DH_PARENT_mst;
66   typedef const struct TPMI_DH_PERSISTENT_mst {
67       UINT8         marshalType;
68       UINT8         modifiers;
69       UINT8         errorCode;
70       UINT32        values[2];
71   } TPMI_DH_PERSISTENT_mst;
72   typedef const struct TPMI_DH_ENTITY_mst {
73       UINT8           marshalType;
74       UINT8           modifiers;
75       UINT8           errorCode;
76       UINT8           ranges;
77       UINT8           singles;
78       UINT32          values[15];
79   } TPMI_DH_ENTITY_mst;
80   typedef const struct TPMI_DH_PCR_mst {
81       UINT8         marshalType;
82       UINT8         modifiers;
83       UINT8         errorCode;
84       UINT32        values[3];
85   } TPMI_DH_PCR_mst;
86   typedef const struct TPMI_SH_AUTH_SESSION_mst {
87       UINT8           marshalType;
88       UINT8           modifiers;
89       UINT8           errorCode;
90       UINT8           ranges;
91       UINT8           singles;
92       UINT32          values[5];
93   } TPMI_SH_AUTH_SESSION_mst;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
94   typedef const struct TPMI_SH_HMAC_mst {
95       UINT8        marshalType;
96       UINT8        modifiers;
97       UINT8        errorCode;
98       UINT32       values[2];
99   } TPMI_SH_HMAC_mst;
100  typedef const struct TPMI_SH_POLICY_mst {
101      UINT8        marshalType;
102      UINT8        modifiers;
103      UINT8        errorCode;
104      UINT32       values[2];
105  } TPMI_SH_POLICY_mst;
106  typedef const struct TPMI_DH_CONTEXT_mst {
107      UINT8          marshalType;
108      UINT8          modifiers;
109      UINT8          errorCode;
110      UINT8          ranges;
111      UINT8          singles;
112      UINT32         values[6];
113  } TPMI_DH_CONTEXT_mst;
114  typedef const struct TPMI_DH_SAVED_mst {
115      UINT8          marshalType;
116      UINT8          modifiers;
117      UINT8          errorCode;
118      UINT8          ranges;
119      UINT8          singles;
120      UINT32         values[7];
121  } TPMI_DH_SAVED_mst;
122  typedef const struct TPMI_RH_HIERARCHY_mst {
123      UINT8          marshalType;
124      UINT8          modifiers;
125      UINT8          errorCode;
126      UINT8          entries;
127      UINT32         values[4];
128  } TPMI_RH_HIERARCHY_mst;
129  typedef const struct TPMI_RH_ENABLES_mst {
130      UINT8          marshalType;
131      UINT8          modifiers;
132      UINT8          errorCode;
133      UINT8          entries;
134      UINT32         values[5];
135  } TPMI_RH_ENABLES_mst;
136  typedef const struct TPMI_RH_HIERARCHY_AUTH_mst {
137      UINT8          marshalType;
138      UINT8          modifiers;
139      UINT8          errorCode;
140      UINT8          entries;
141      UINT32         values[4];
142  } TPMI_RH_HIERARCHY_AUTH_mst;
143  typedef const struct TPMI_RH_PLATFORM_mst {
144      UINT8          marshalType;
145      UINT8          modifiers;
146      UINT8          errorCode;
147      UINT8          entries;
148      UINT32         values[1];
149  } TPMI_RH_PLATFORM_mst;
150  typedef const struct TPMI_RH_OWNER_mst {
151      UINT8          marshalType;
152      UINT8          modifiers;
153      UINT8          errorCode;
154      UINT8          entries;
155      UINT32         values[2];
156  } TPMI_RH_OWNER_mst;
157  typedef const struct TPMI_RH_ENDORSEMENT_mst {
158      UINT8          marshalType;
159      UINT8          modifiers;
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
160     UINT8         errorCode;
161     UINT8         entries;
162     UINT32        values[2];
163 } TPMI_RH_ENDORSEMENT_mst;
164 typedef const struct TPMI_RH_PROVISION_mst {
165     UINT8         marshalType;
166     UINT8         modifiers;
167     UINT8         errorCode;
168     UINT8         entries;
169     UINT32        values[2];
170 } TPMI_RH_PROVISION_mst;
171 typedef const struct TPMI_RH_CLEAR_mst {
172     UINT8         marshalType;
173     UINT8         modifiers;
174     UINT8         errorCode;
175     UINT8         entries;
176     UINT32        values[2];
177 } TPMI_RH_CLEAR_mst;
178 typedef const struct TPMI_RH_NV_AUTH_mst {
179     UINT8         marshalType;
180     UINT8         modifiers;
181     UINT8         errorCode;
182     UINT8         ranges;
183     UINT8         singles;
184     UINT32        values[4];
185 } TPMI_RH_NV_AUTH_mst;
186 typedef const struct TPMI_RH_LOCKOUT_mst {
187     UINT8         marshalType;
188     UINT8         modifiers;
189     UINT8         errorCode;
190     UINT8         entries;
191     UINT32        values[1];
192 } TPMI_RH_LOCKOUT_mst;
193 typedef const struct TPMI_RH_NV_INDEX_mst {
194     UINT8       marshalType;
195     UINT8       modifiers;
196     UINT8       errorCode;
197     UINT32      values[2];
198 } TPMI_RH_NV_INDEX_mst;
199 typedef const struct TPMI_RH_AC_mst {
200     UINT8       marshalType;
201     UINT8       modifiers;
202     UINT8       errorCode;
203     UINT32      values[2];
204 } TPMI_RH_AC_mst;
205 typedef const struct TPMI_ALG_HASH_mst {
206     UINT8       marshalType;
207     UINT8       modifiers;
208     UINT8       errorCode;
209     UINT32      values[5];
210 } TPMI_ALG_HASH_mst;
211 typedef const struct TPMI_ALG_ASYM_mst {
212     UINT8       marshalType;
213     UINT8       modifiers;
214     UINT8       errorCode;
215     UINT32      values[5];
216 } TPMI_ALG_ASYM_mst;
217 typedef const struct TPMI_ALG_SYM_mst {
218     UINT8       marshalType;
219     UINT8       modifiers;
220     UINT8       errorCode;
221     UINT32      values[5];
222 } TPMI_ALG_SYM_mst;
223 typedef const struct TPMI_ALG_SYM_OBJECT_mst {
224     UINT8       marshalType;
225     UINT8       modifiers;
```

```
226         UINT8         errorCode;
227         UINT32        values[5];
228     } TPMI_ALG_SYM_OBJECT_mst;
229     typedef const struct TPMI_ALG_SYM_MODE_mst {
230         UINT8         marshalType;
231         UINT8         modifiers;
232         UINT8         errorCode;
233         UINT32        values[4];
234     } TPMI_ALG_SYM_MODE_mst;
235     typedef const struct TPMI_ALG_KDF_mst {
236         UINT8         marshalType;
237         UINT8         modifiers;
238         UINT8         errorCode;
239         UINT32        values[4];
240     } TPMI_ALG_KDF_mst;
241     typedef const struct TPMI_ALG_SIG_SCHEME_mst {
242         UINT8         marshalType;
243         UINT8         modifiers;
244         UINT8         errorCode;
245         UINT32        values[4];
246     } TPMI_ALG_SIG_SCHEME_mst;
247     typedef const struct TPMI_ECC_KEY_EXCHANGE_mst {
248         UINT8         marshalType;
249         UINT8         modifiers;
250         UINT8         errorCode;
251         UINT32        values[4];
252     } TPMI_ECC_KEY_EXCHANGE_mst;
253     typedef const struct TPMI_ST_COMMAND_TAG_mst {
254         UINT8           marshalType;
255         UINT8           modifiers;
256         UINT8           errorCode;
257         UINT8           entries;
258         UINT32          values[2];
259     } TPMI_ST_COMMAND_TAG_mst;
260     typedef const struct TPMI_ALG_MAC_SCHEME_mst {
261         UINT8         marshalType;
262         UINT8         modifiers;
263         UINT8         errorCode;
264         UINT32        values[5];
265     } TPMI_ALG_MAC_SCHEME_mst;
266     typedef const struct TPMI_ALG_CIPHER_MODE_mst {
267         UINT8         marshalType;
268         UINT8         modifiers;
269         UINT8         errorCode;
270         UINT32        values[4];
271     } TPMI_ALG_CIPHER_MODE_mst;
272     typedef const struct TPMS_EMPTY_mst
273     {
274         UINT8       marshalType;
275         UINT8       elements;
276         UINT16      values[3];
277     } TPMS_EMPTY_mst;
278     typedef const struct TPMS_ALGORITHM_DESCRIPTION_mst
279     {
280         UINT8       marshalType;
281         UINT8       elements;
282         UINT16      values[6];
283     } TPMS_ALGORITHM_DESCRIPTION_mst;
284     typedef struct TPMU_HA_mst
285     {
286         BYTE                countOfselectors;
287         BYTE                modifiers;
288         UINT16            offsetOfUnmarshalTypes;
289         UINT32            selectors[9];
290         UINT16            marshalingTypes[9];
291     } TPMU_HA_mst;
```

```
292     typedef const struct TPMT_HA_mst
293     {
294         UINT8     marshalType;
295         UINT8     elements;
296         UINT16    values[6];
297     } TPMT_HA_mst;
298     typedef const struct TPMS_PCR_SELECT_mst
299     {
300         UINT8     marshalType;
301         UINT8     elements;
302         UINT16    values[6];
303     } TPMS_PCR_SELECT_mst;
304     typedef const struct TPMS_PCR_SELECTION_mst
305     {
306         UINT8     marshalType;
307         UINT8     elements;
308         UINT16    values[9];
309     } TPMS_PCR_SELECTION_mst;
310     typedef const struct TPMT_TK_CREATION_mst
311     {
312         UINT8     marshalType;
313         UINT8     elements;
314         UINT16    values[9];
315     } TPMT_TK_CREATION_mst;
316     typedef const struct TPMT_TK_VERIFIED_mst
317     {
318         UINT8     marshalType;
319         UINT8     elements;
320         UINT16    values[9];
321     } TPMT_TK_VERIFIED_mst;
322     typedef const struct TPMT_TK_AUTH_mst
323     {
324         UINT8     marshalType;
325         UINT8     elements;
326         UINT16    values[9];
327     } TPMT_TK_AUTH_mst;
328     typedef const struct TPMT_TK_HASHCHECK_mst
329     {
330         UINT8     marshalType;
331         UINT8     elements;
332         UINT16    values[9];
333     } TPMT_TK_HASHCHECK_mst;
334     typedef const struct TPMS_ALG_PROPERTY_mst
335     {
336         UINT8     marshalType;
337         UINT8     elements;
338         UINT16    values[6];
339     } TPMS_ALG_PROPERTY_mst;
340     typedef const struct TPMS_TAGGED_PROPERTY_mst
341     {
342         UINT8     marshalType;
343         UINT8     elements;
344         UINT16    values[6];
345     } TPMS_TAGGED_PROPERTY_mst;
346     typedef const struct TPMS_TAGGED_PCR_SELECT_mst
347     {
348         UINT8     marshalType;
349         UINT8     elements;
350         UINT16    values[9];
351     } TPMS_TAGGED_PCR_SELECT_mst;
352     typedef const struct TPMS_TAGGED_POLICY_mst
353     {
354         UINT8     marshalType;
355         UINT8     elements;
356         UINT16    values[6];
357     } TPMS_TAGGED_POLICY_mst;
```

```
358     typedef struct TPMU_CAPABILITIES_mst
359     {
360         BYTE            countOfselectors;
361         BYTE            modifiers;
362         UINT16        offsetOfUnmarshalTypes;
363         UINT32        selectors[10];
364         UINT16        marshalingTypes[10];
365     } TPMU_CAPABILITIES_mst;
366     typedef const struct TPMS_CAPABILITY_DATA_mst
367     {
368         UINT8     marshalType;
369         UINT8     elements;
370         UINT16    values[6];
371     } TPMS_CAPABILITY_DATA_mst;
372     typedef const struct TPMS_CLOCK_INFO_mst
373     {
374         UINT8     marshalType;
375         UINT8     elements;
376         UINT16    values[12];
377     } TPMS_CLOCK_INFO_mst;
378     typedef const struct TPMS_TIME_INFO_mst
379     {
380         UINT8     marshalType;
381         UINT8     elements;
382         UINT16    values[6];
383     } TPMS_TIME_INFO_mst;
384     typedef const struct TPMS_TIME_ATTEST_INFO_mst
385     {
386         UINT8     marshalType;
387         UINT8     elements;
388         UINT16    values[6];
389     } TPMS_TIME_ATTEST_INFO_mst;
390     typedef const struct TPMS_CERTIFY_INFO_mst
391     {
392         UINT8     marshalType;
393         UINT8     elements;
394         UINT16    values[6];
395     } TPMS_CERTIFY_INFO_mst;
396     typedef const struct TPMS_QUOTE_INFO_mst
397     {
398         UINT8     marshalType;
399         UINT8     elements;
400         UINT16    values[6];
401     } TPMS_QUOTE_INFO_mst;
402     typedef const struct TPMS_COMMAND_AUDIT_INFO_mst
403     {
404         UINT8     marshalType;
405         UINT8     elements;
406         UINT16    values[12];
407     } TPMS_COMMAND_AUDIT_INFO_mst;
408     typedef const struct TPMS_SESSION_AUDIT_INFO_mst
409     {
410         UINT8     marshalType;
411         UINT8     elements;
412         UINT16    values[6];
413     } TPMS_SESSION_AUDIT_INFO_mst;
414     typedef const struct TPMS_CREATION_INFO_mst
415     {
416         UINT8     marshalType;
417         UINT8     elements;
418         UINT16    values[6];
419     } TPMS_CREATION_INFO_mst;
420     typedef const struct TPMS_NV_CERTIFY_INFO_mst
421     {
422         UINT8     marshalType;
423         UINT8     elements;
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
424     UINT16    values[9];
425 } TPMS_NV_CERTIFY_INFO_mst;
426 typedef const struct TPMS_NV_DIGEST_CERTIFY_INFO_mst
427 {
428     UINT8     marshalType;
429     UINT8     elements;
430     UINT16    values[6];
431 } TPMS_NV_DIGEST_CERTIFY_INFO_mst;
432 typedef const struct TPMI_ST_ATTEST_mst {
433     UINT8         marshalType;
434     UINT8         modifiers;
435     UINT8         errorCode;
436     UINT8         ranges;
437     UINT8         singles;
438     UINT32        values[3];
439 } TPMI_ST_ATTEST_mst;
440 typedef struct TPMU_ATTEST_mst
441 {
442     BYTE            countOfselectors;
443     BYTE            modifiers;
444     UINT16        offsetOfUnmarshalTypes;
445     UINT32        selectors[8];
446     UINT16        marshalingTypes[8];
447 } TPMU_ATTEST_mst;
448 typedef const struct TPMS_ATTEST_mst
449 {
450     UINT8     marshalType;
451     UINT8     elements;
452     UINT16    values[21];
453 } TPMS_ATTEST_mst;
454 typedef const struct TPMS_AUTH_COMMAND_mst
455 {
456     UINT8     marshalType;
457     UINT8     elements;
458     UINT16    values[12];
459 } TPMS_AUTH_COMMAND_mst;
460 typedef const struct TPMS_AUTH_RESPONSE_mst
461 {
462     UINT8     marshalType;
463     UINT8     elements;
464     UINT16    values[9];
465 } TPMS_AUTH_RESPONSE_mst;
466 typedef const struct TPMI_TDES_KEY_BITS_mst {
467     UINT8         marshalType;
468     UINT8         modifiers;
469     UINT8         errorCode;
470     UINT8         entries;
471     UINT32        values[3];
472 } TPMI_TDES_KEY_BITS_mst;
473 typedef const struct TPMI_AES_KEY_BITS_mst {
474     UINT8         marshalType;
475     UINT8         modifiers;
476     UINT8         errorCode;
477     UINT8         entries;
478     UINT32        values[4];
479 } TPMI_AES_KEY_BITS_mst;
480 typedef const struct TPMI_SM4_KEY_BITS_mst {
481     UINT8         marshalType;
482     UINT8         modifiers;
483     UINT8         errorCode;
484     UINT8         entries;
485     UINT32        values[2];
486 } TPMI_SM4_KEY_BITS_mst;
487 typedef const struct TPMI_CAMELLIA_KEY_BITS_mst {
488     UINT8         marshalType;
489     UINT8         modifiers;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
490     UINT8         errorCode;
491     UINT8         entries;
492     UINT32        values[4];
493 } TPMI_CAMELLIA_KEY_BITS_mst;
494 typedef struct TPMU_SYM_KEY_BITS_mst
495 {
496     BYTE            countOfselectors;
497     BYTE            modifiers;
498     UINT16        offsetOfUnmarshalTypes;
499     UINT32        selectors[6];
500     UINT16        marshalingTypes[6];
501 } TPMU_SYM_KEY_BITS_mst;
502 typedef struct TPMU_SYM_MODE_mst
503 {
504     BYTE            countOfselectors;
505     BYTE            modifiers;
506     UINT16        offsetOfUnmarshalTypes;
507     UINT32        selectors[6];
508     UINT16        marshalingTypes[6];
509 } TPMU_SYM_MODE_mst;
510 typedef const struct TPMT_SYM_DEF_mst
511 {
512     UINT8     marshalType;
513     UINT8     elements;
514     UINT16    values[9];
515 } TPMT_SYM_DEF_mst;
516 typedef const struct TPMT_SYM_DEF_OBJECT_mst
517 {
518     UINT8     marshalType;
519     UINT8     elements;
520     UINT16    values[9];
521 } TPMT_SYM_DEF_OBJECT_mst;
522 typedef const struct TPMS_SYMCIPHER_PARMS_mst
523 {
524     UINT8     marshalType;
525     UINT8     elements;
526     UINT16    values[3];
527 } TPMS_SYMCIPHER_PARMS_mst;
528 typedef const struct TPMS_DERIVE_mst
529 {
530     UINT8     marshalType;
531     UINT8     elements;
532     UINT16    values[6];
533 } TPMS_DERIVE_mst;
534 typedef const struct TPMS_SENSITIVE_CREATE_mst
535 {
536     UINT8     marshalType;
537     UINT8     elements;
538     UINT16    values[6];
539 } TPMS_SENSITIVE_CREATE_mst;
540 typedef const struct TPMS_SCHEME_HASH_mst
541 {
542     UINT8     marshalType;
543     UINT8     elements;
544     UINT16    values[3];
545 } TPMS_SCHEME_HASH_mst;
546 typedef const struct TPMS_SCHEME_ECDAA_mst
547 {
548     UINT8     marshalType;
549     UINT8     elements;
550     UINT16    values[6];
551 } TPMS_SCHEME_ECDAA_mst;
552 typedef const struct TPMI_ALG_KEYEDHASH_SCHEME_mst {
553     UINT8       marshalType;
554     UINT8       modifiers;
555     UINT8       errorCode;
```

```
556         UINT32       values[4];
557     } TPMI_ALG_KEYEDHASH_SCHEME_mst;
558     typedef const struct TPMS_SCHEME_XOR_mst
559     {
560         UINT8      marshalType;
561         UINT8      elements;
562         UINT16     values[6];
563     } TPMS_SCHEME_XOR_mst;
564     typedef struct TPMU_SCHEME_KEYEDHASH_mst
565     {
566         BYTE             countOfselectors;
567         BYTE             modifiers;
568         UINT16         offsetOfUnmarshalTypes;
569         UINT32         selectors[3];
570         UINT16         marshalingTypes[3];
571     } TPMU_SCHEME_KEYEDHASH_mst;
572     typedef const struct TPMT_KEYEDHASH_SCHEME_mst
573     {
574         UINT8      marshalType;
575         UINT8      elements;
576         UINT16     values[6];
577     } TPMT_KEYEDHASH_SCHEME_mst;
578     typedef struct TPMU_SIG_SCHEME_mst
579     {
580         BYTE             countOfselectors;
581         BYTE             modifiers;
582         UINT16         offsetOfUnmarshalTypes;
583         UINT32         selectors[8];
584         UINT16         marshalingTypes[8];
585     } TPMU_SIG_SCHEME_mst;
586     typedef const struct TPMT_SIG_SCHEME_mst
587     {
588         UINT8      marshalType;
589         UINT8      elements;
590         UINT16     values[6];
591     } TPMT_SIG_SCHEME_mst;
592     typedef struct TPMU_KDF_SCHEME_mst
593     {
594         BYTE             countOfselectors;
595         BYTE             modifiers;
596         UINT16         offsetOfUnmarshalTypes;
597         UINT32         selectors[5];
598         UINT16         marshalingTypes[5];
599     } TPMU_KDF_SCHEME_mst;
600     typedef const struct TPMT_KDF_SCHEME_mst
601     {
602         UINT8      marshalType;
603         UINT8      elements;
604         UINT16     values[6];
605     } TPMT_KDF_SCHEME_mst;
606     typedef const struct TPMI_ALG_ASYM_SCHEME_mst {
607         UINT8       marshalType;
608         UINT8       modifiers;
609         UINT8       errorCode;
610         UINT32      values[4];
611     } TPMI_ALG_ASYM_SCHEME_mst;
612     typedef struct TPMU_ASYM_SCHEME_mst
613     {
614         BYTE             countOfselectors;
615         BYTE             modifiers;
616         UINT16         offsetOfUnmarshalTypes;
617         UINT32         selectors[11];
618         UINT16         marshalingTypes[11];
619     } TPMU_ASYM_SCHEME_mst;
620     typedef const struct TPMT_ASYM_SCHEME_mst
621     {
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
622         UINT8     marshalType;
623         UINT8     elements;
624         UINT16    values[6];
625     } TPMT_ASYM_SCHEME_mst;
626     typedef const struct TPMI_ALG_RSA_SCHEME_mst {
627         UINT8       marshalType;
628         UINT8       modifiers;
629         UINT8       errorCode;
630         UINT32      values[4];
631     } TPMI_ALG_RSA_SCHEME_mst;
632     typedef const struct TPMT_RSA_SCHEME_mst
633     {
634         UINT8     marshalType;
635         UINT8     elements;
636         UINT16    values[6];
637     } TPMT_RSA_SCHEME_mst;
638     typedef const struct TPMI_ALG_RSA_DECRYPT_mst {
639         UINT8       marshalType;
640         UINT8       modifiers;
641         UINT8       errorCode;
642         UINT32      values[4];
643     } TPMI_ALG_RSA_DECRYPT_mst;
644     typedef const struct TPMT_RSA_DECRYPT_mst
645     {
646         UINT8     marshalType;
647         UINT8     elements;
648         UINT16    values[6];
649     } TPMT_RSA_DECRYPT_mst;
650     typedef const struct TPMI_RSA_KEY_BITS_mst {
651         UINT8         marshalType;
652         UINT8         modifiers;
653         UINT8         errorCode;
654         UINT8         entries;
655         UINT32        values[5];
656     } TPMI_RSA_KEY_BITS_mst;
657     typedef const struct TPMS_ECC_POINT_mst
658     {
659         UINT8     marshalType;
660         UINT8     elements;
661         UINT16    values[6];
662     } TPMS_ECC_POINT_mst;
663     typedef const struct TPMI_ALG_ECC_SCHEME_mst {
664         UINT8       marshalType;
665         UINT8       modifiers;
666         UINT8       errorCode;
667         UINT32      values[4];
668     } TPMI_ALG_ECC_SCHEME_mst;
669     typedef const struct TPMI_ECC_CURVE_mst {
670         UINT8       marshalType;
671         UINT8       modifiers;
672         UINT8       errorCode;
673         UINT32      values[3];
674     } TPMI_ECC_CURVE_mst;
675     typedef const struct TPMT_ECC_SCHEME_mst
676     {
677         UINT8     marshalType;
678         UINT8     elements;
679         UINT16    values[6];
680     } TPMT_ECC_SCHEME_mst;
681     typedef const struct TPMS_ALGORITHM_DETAIL_ECC_mst
682     {
683         UINT8     marshalType;
684         UINT8     elements;
685         UINT16    values[33];
686     } TPMS_ALGORITHM_DETAIL_ECC_mst;
687     typedef const struct TPMS_SIGNATURE_RSA_mst
```

 Copyright © TCG 2006-2020

```
688   {
689       UINT8     marshalType;
690       UINT8     elements;
691       UINT16    values[6];
692   } TPMS_SIGNATURE_RSA_mst;
693   typedef const struct TPMS_SIGNATURE_ECC_mst
694   {
695       UINT8     marshalType;
696       UINT8     elements;
697       UINT16    values[9];
698   } TPMS_SIGNATURE_ECC_mst;
699   typedef struct TPMU_SIGNATURE_mst
700   {
701       BYTE            countOfselectors;
702       BYTE            modifiers;
703       UINT16        offsetOfUnmarshalTypes;
704       UINT32        selectors[8];
705       UINT16        marshalingTypes[8];
706   } TPMU_SIGNATURE_mst;
707   typedef const struct TPMT_SIGNATURE_mst
708   {
709       UINT8     marshalType;
710       UINT8     elements;
711       UINT16    values[6];
712   } TPMT_SIGNATURE_mst;
713   typedef struct TPMU_ENCRYPTED_SECRET_mst
714   {
715       BYTE            countOfselectors;
716       BYTE            modifiers;
717       UINT16        offsetOfUnmarshalTypes;
718       UINT32        selectors[4];
719       UINT16        marshalingTypes[4];
720   } TPMU_ENCRYPTED_SECRET_mst;
721   typedef const struct TPMI_ALG_PUBLIC_mst {
722       UINT8       marshalType;
723       UINT8       modifiers;
724       UINT8       errorCode;
725       UINT32      values[4];
726   } TPMI_ALG_PUBLIC_mst;
727   typedef struct TPMU_PUBLIC_ID_mst
728   {
729       BYTE            countOfselectors;
730       BYTE            modifiers;
731       UINT16        offsetOfUnmarshalTypes;
732       UINT32        selectors[4];
733       UINT16        marshalingTypes[4];
734   } TPMU_PUBLIC_ID_mst;
735   typedef const struct TPMS_KEYEDHASH_PARMS_mst
736   {
737       UINT8     marshalType;
738       UINT8     elements;
739       UINT16    values[3];
740   } TPMS_KEYEDHASH_PARMS_mst;
741   typedef const struct TPMS_ASYM_PARMS_mst
742   {
743       UINT8     marshalType;
744       UINT8     elements;
745       UINT16    values[6];
746   } TPMS_ASYM_PARMS_mst;
747   typedef const struct TPMS_RSA_PARMS_mst
748   {
749       UINT8     marshalType;
750       UINT8     elements;
751       UINT16    values[12];
752   } TPMS_RSA_PARMS_mst;
753   typedef const struct TPMS_ECC_PARMS_mst
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
754   {
755       UINT8     marshalType;
756       UINT8     elements;
757       UINT16    values[12];
758   } TPMS_ECC_PARMS_mst;
759   typedef struct TPMU_PUBLIC_PARMS_mst
760   {
761       BYTE            countOfselectors;
762       BYTE            modifiers;
763       UINT16        offsetOfUnmarshalTypes;
764       UINT32        selectors[4];
765       UINT16        marshalingTypes[4];
766   } TPMU_PUBLIC_PARMS_mst;
767   typedef const struct TPMT_PUBLIC_PARMS_mst
768   {
769       UINT8     marshalType;
770       UINT8     elements;
771       UINT16    values[6];
772   } TPMT_PUBLIC_PARMS_mst;
773   typedef const struct TPMT_PUBLIC_mst
774   {
775       UINT8     marshalType;
776       UINT8     elements;
777       UINT16    values[18];
778   } TPMT_PUBLIC_mst;
779   typedef struct TPMU_SENSITIVE_COMPOSITE_mst
780   {
781       BYTE            countOfselectors;
782       BYTE            modifiers;
783       UINT16        offsetOfUnmarshalTypes;
784       UINT32        selectors[4];
785       UINT16        marshalingTypes[4];
786   } TPMU_SENSITIVE_COMPOSITE_mst;
787   typedef const struct TPMT_SENSITIVE_mst
788   {
789       UINT8     marshalType;
790       UINT8     elements;
791       UINT16    values[12];
792   } TPMT_SENSITIVE_mst;
793   typedef const struct _PRIVATE_mst
794   {
795       UINT8     marshalType;
796       UINT8     elements;
797       UINT16    values[9];
798   } _PRIVATE_mst;
799   typedef const struct TPMS_ID_OBJECT_mst
800   {
801       UINT8     marshalType;
802       UINT8     elements;
803       UINT16    values[6];
804   } TPMS_ID_OBJECT_mst;
805   typedef const struct TPMS_NV_PIN_COUNTER_PARAMETERS_mst
806   {
807       UINT8     marshalType;
808       UINT8     elements;
809       UINT16    values[6];
810   } TPMS_NV_PIN_COUNTER_PARAMETERS_mst;
811   typedef const struct TPMS_NV_PUBLIC_mst
812   {
813       UINT8     marshalType;
814       UINT8     elements;
815       UINT16    values[15];
816   } TPMS_NV_PUBLIC_mst;
817   typedef const struct TPMS_CONTEXT_DATA_mst
818   {
819       UINT8     marshalType;
```

 Copyright © TCG 2006-2020

```
820         UINT8      elements;
821         UINT16     values[6];
822     } TPMS_CONTEXT_DATA_mst;
823     typedef const struct TPMS_CONTEXT_mst
824     {
825         UINT8      marshalType;
826         UINT8      elements;
827         UINT16     values[12];
828     } TPMS_CONTEXT_mst;
829     typedef const struct TPMS_CREATION_DATA_mst
830     {
831         UINT8      marshalType;
832         UINT8      elements;
833         UINT16     values[21];
834     } TPMS_CREATION_DATA_mst;
835     typedef const struct TPM_AT_mst {
836         UINT8          marshalType;
837         UINT8          modifiers;
838         UINT8          errorCode;
839         UINT8          entries;
840         UINT32         values[4];
841     } TPM_AT_mst;
842     typedef const struct TPMS_AC_OUTPUT_mst
843     {
844         UINT8      marshalType;
845         UINT8      elements;
846         UINT16     values[6];
847     } TPMS_AC_OUTPUT_mst;
848     typedef const struct Type02_mst {
849         UINT8        marshalType;
850         UINT8        modifiers;
851         UINT8        errorCode;
852         UINT32       values[2];
853     } Type02_mst;
854     typedef const struct Type03_mst {
855         UINT8        marshalType;
856         UINT8        modifiers;
857         UINT8        errorCode;
858         UINT32       values[2];
859     } Type03_mst;
860     typedef const struct Type04_mst {
861         UINT8        marshalType;
862         UINT8        modifiers;
863         UINT8        errorCode;
864         UINT32       values[2];
865     } Type04_mst;
866     typedef const struct Type06_mst {
867         UINT8        marshalType;
868         UINT8        modifiers;
869         UINT8        errorCode;
870         UINT32       values[2];
871     } Type06_mst;
872     typedef const struct Type08_mst {
873         UINT8        marshalType;
874         UINT8        modifiers;
875         UINT8        errorCode;
876         UINT32       values[2];
877     } Type08_mst;
878     typedef const struct Type10_mst {
879         UINT8          marshalType;
880         UINT8          modifiers;
881         UINT8          errorCode;
882         UINT8          entries;
883         UINT32         values[1];
884     } Type10_mst;
885     typedef const struct Type11_mst {
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
886         UINT8           marshalType;
887         UINT8           modifiers;
888         UINT8           errorCode;
889         UINT8           entries;
890         UINT32          values[1];
891     } Type11_mst;
892     typedef const struct Type12_mst {
893         UINT8           marshalType;
894         UINT8           modifiers;
895         UINT8           errorCode;
896         UINT8           entries;
897         UINT32          values[2];
898     } Type12_mst;
899     typedef const struct Type13_mst {
900         UINT8           marshalType;
901         UINT8           modifiers;
902         UINT8           errorCode;
903         UINT8           entries;
904         UINT32          values[1];
905     } Type13_mst;
906     typedef const struct Type15_mst {
907         UINT8         marshalType;
908         UINT8         modifiers;
909         UINT8         errorCode;
910         UINT32        values[2];
911     } Type15_mst;
912     typedef const struct Type17_mst {
913         UINT8         marshalType;
914         UINT8         modifiers;
915         UINT8         errorCode;
916         UINT32        values[2];
917     } Type17_mst;
918     typedef const struct Type18_mst {
919         UINT8         marshalType;
920         UINT8         modifiers;
921         UINT8         errorCode;
922         UINT32        values[2];
923     } Type18_mst;
924     typedef const struct Type19_mst {
925         UINT8         marshalType;
926         UINT8         modifiers;
927         UINT8         errorCode;
928         UINT32        values[2];
929     } Type19_mst;
930     typedef const struct Type20_mst {
931         UINT8         marshalType;
932         UINT8         modifiers;
933         UINT8         errorCode;
934         UINT32        values[2];
935     } Type20_mst;
936     typedef const struct Type22_mst {
937         UINT8         marshalType;
938         UINT8         modifiers;
939         UINT8         errorCode;
940         UINT32        values[2];
941     } Type22_mst;
942     typedef const struct Type23_mst {
943         UINT8         marshalType;
944         UINT8         modifiers;
945         UINT8         errorCode;
946         UINT32        values[2];
947     } Type23_mst;
948     typedef const struct Type24_mst {
949         UINT8         marshalType;
950         UINT8         modifiers;
951         UINT8         errorCode;
```

```
952         UINT32      values[2];
953     } Type24_mst;
954     typedef const struct Type25_mst {
955         UINT8       marshalType;
956         UINT8       modifiers;
957         UINT8       errorCode;
958         UINT32      values[2];
959     } Type25_mst;
960     typedef const struct Type26_mst {
961         UINT8       marshalType;
962         UINT8       modifiers;
963         UINT8       errorCode;
964         UINT32      values[2];
965     } Type26_mst;
966     typedef const struct Type28_mst {
967         UINT8       marshalType;
968         UINT8       modifiers;
969         UINT8       errorCode;
970         UINT32      values[2];
971     } Type28_mst;
972     typedef const struct Type29_mst {
973         UINT8       marshalType;
974         UINT8       modifiers;
975         UINT8       errorCode;
976         UINT32      values[2];
977     } Type29_mst;
978     typedef const struct Type32_mst {
979         UINT8       marshalType;
980         UINT8       modifiers;
981         UINT8       errorCode;
982         UINT32      values[2];
983     } Type32_mst;
984     typedef const struct Type33_mst {
985         UINT8       marshalType;
986         UINT8       modifiers;
987         UINT8       errorCode;
988         UINT32      values[2];
989     } Type33_mst;
990     typedef const struct Type34_mst {
991         UINT8       marshalType;
992         UINT8       modifiers;
993         UINT8       errorCode;
994         UINT32      values[2];
995     } Type34_mst;
996     typedef const struct Type37_mst {
997         UINT8       marshalType;
998         UINT8       modifiers;
999         UINT8       errorCode;
1000        UINT32      values[2];
1001    } Type37_mst;
1002    typedef const struct Type40_mst {
1003        UINT8       marshalType;
1004        UINT8       modifiers;
1005        UINT8       errorCode;
1006        UINT32      values[2];
1007    } Type40_mst;
1008    typedef const struct Type41_mst {
1009        UINT8       marshalType;
1010        UINT8       modifiers;
1011        UINT8       errorCode;
1012        UINT32      values[2];
1013    } Type41_mst;
1014    typedef const struct Type43_mst {
1015        UINT8       marshalType;
1016        UINT8       modifiers;
1017        UINT8       errorCode;
```

 Copyright © TCG 2006-2020

```
1018        UINT32      values[2];
1019    } Type43_mst;
```

Defines for array lookup

```
1020    #define UINT8_ARRAY_MARSHAL_INDEX                  0   // 0x00
1021    #define TPM_CC_ARRAY_MARSHAL_INDEX                 1   // 0x01
1022    #define TPMA_CC_ARRAY_MARSHAL_INDEX                2   // 0x02
1023    #define TPM_ALG_ID_ARRAY_MARSHAL_INDEX             3   // 0x03
1024    #define TPM_HANDLE_ARRAY_MARSHAL_INDEX             4   // 0x04
1025    #define TPM2B_DIGEST_ARRAY_MARSHAL_INDEX           5   // 0x05
1026    #define TPMT_HA_ARRAY_MARSHAL_INDEX                6   // 0x06
1027    #define TPMS_PCR_SELECTION_ARRAY_MARSHAL_INDEX     7   // 0x07
1028    #define TPMS_ALG_PROPERTY_ARRAY_MARSHAL_INDEX      8   // 0x08
1029    #define TPMS_TAGGED_PROPERTY_ARRAY_MARSHAL_INDEX   9   // 0x09
1030    #define TPMS_TAGGED_PCR_SELECT_ARRAY_MARSHAL_INDEX 10  // 0x0A
1031    #define TPM_ECC_CURVE_ARRAY_MARSHAL_INDEX          11  // 0x0B
1032    #define TPMS_TAGGED_POLICY_ARRAY_MARSHAL_INDEX     12  // 0x0C
1033    #define TPMS_AC_OUTPUT_ARRAY_MARSHAL_INDEX         13  // 0x0D
```

The defines to connect a typename to an index in the MarshalLookupTable()

```
1034    #define UINT8_MARSHAL_INDEX                        0   // 0x00
1035    #define BYTE_MARSHAL_INDEX                         UINT8_MARSHAL_INDEX
1036    #define TPM_HT_MARSHAL_INDEX                       UINT8_MARSHAL_INDEX
1037    #define TPMA_LOCALITY_MARSHAL_INDEX                UINT8_MARSHAL_INDEX
1038    #define UINT16_MARSHAL_INDEX                       1   // 0x01
1039    #define TPM_KEY_SIZE_MARSHAL_INDEX                 UINT16_MARSHAL_INDEX
1040    #define TPM_KEY_BITS_MARSHAL_INDEX                 UINT16_MARSHAL_INDEX
1041    #define TPM_ALG_ID_MARSHAL_INDEX                   UINT16_MARSHAL_INDEX
1042    #define TPM_ST_MARSHAL_INDEX                       UINT16_MARSHAL_INDEX
1043    #define UINT32_MARSHAL_INDEX                       2   // 0x02
1044    #define TPM_ALGORITHM_ID_MARSHAL_INDEX             UINT32_MARSHAL_INDEX
1045    #define TPM_MODIFIER_INDICATOR_MARSHAL_INDEX       UINT32_MARSHAL_INDEX
1046    #define TPM_AUTHORIZATION_SIZE_MARSHAL_INDEX       UINT32_MARSHAL_INDEX
1047    #define TPM_PARAMETER_SIZE_MARSHAL_INDEX           UINT32_MARSHAL_INDEX
1048    #define TPM_SPEC_MARSHAL_INDEX                     UINT32_MARSHAL_INDEX
1049    #define TPM_GENERATED_MARSHAL_INDEX                UINT32_MARSHAL_INDEX
1050    #define TPM_CC_MARSHAL_INDEX                       UINT32_MARSHAL_INDEX
1051    #define TPM_RC_MARSHAL_INDEX                       UINT32_MARSHAL_INDEX
1052    #define TPM_PT_MARSHAL_INDEX                       UINT32_MARSHAL_INDEX
1053    #define TPM_PT_PCR_MARSHAL_INDEX                   UINT32_MARSHAL_INDEX
1054    #define TPM_PS_MARSHAL_INDEX                       UINT32_MARSHAL_INDEX
1055    #define TPM_HANDLE_MARSHAL_INDEX                   UINT32_MARSHAL_INDEX
1056    #define TPM_RH_MARSHAL_INDEX                       UINT32_MARSHAL_INDEX
1057    #define TPM_HC_MARSHAL_INDEX                       UINT32_MARSHAL_INDEX
1058    #define TPMA_PERMANENT_MARSHAL_INDEX               UINT32_MARSHAL_INDEX
1059    #define TPMA_STARTUP_CLEAR_MARSHAL_INDEX           UINT32_MARSHAL_INDEX
1060    #define TPMA_MEMORY_MARSHAL_INDEX                  UINT32_MARSHAL_INDEX
1061    #define TPMA_CC_MARSHAL_INDEX                      UINT32_MARSHAL_INDEX
1062    #define TPMA_MODES_MARSHAL_INDEX                   UINT32_MARSHAL_INDEX
1063    #define TPMA_X509_KEY_USAGE_MARSHAL_INDEX          UINT32_MARSHAL_INDEX
1064    #define TPM_NV_INDEX_MARSHAL_INDEX                 UINT32_MARSHAL_INDEX
1065    #define TPM_AE_MARSHAL_INDEX                       UINT32_MARSHAL_INDEX
1066    #define UINT64_MARSHAL_INDEX                       3   // 0x03
1067    #define INT8_MARSHAL_INDEX                         4   // 0x04
1068    #define INT16_MARSHAL_INDEX                        5   // 0x05
1069    #define INT32_MARSHAL_INDEX                        6   // 0x06
1070    #define INT64_MARSHAL_INDEX                        7   // 0x07
1071    #define UINT0_MARSHAL_INDEX                        8   // 0x08
1072    #define TPM_ECC_CURVE_MARSHAL_INDEX                9   // 0x09
1073    #define TPM_CLOCK_ADJUST_MARSHAL_INDEX             10  // 0x0A
1074    #define TPM_EO_MARSHAL_INDEX                       11  // 0x0B
1075    #define TPM_SU_MARSHAL_INDEX                       12  // 0x0C
1076    #define TPM_SE_MARSHAL_INDEX                       13  // 0x0D
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
1077  #define TPM_CAP_MARSHAL_INDEX                         14  // 0x0E
1078  #define TPMA_ALGORITHM_MARSHAL_INDEX                  15  // 0x0F
1079  #define TPMA_OBJECT_MARSHAL_INDEX                     16  // 0x10
1080  #define TPMA_SESSION_MARSHAL_INDEX                    17  // 0x11
1081  #define TPMI_YES_NO_MARSHAL_INDEX                     18  // 0x12
1082  #define TPMI_DH_OBJECT_MARSHAL_INDEX                  19  // 0x13
1083  #define TPMI_DH_PARENT_MARSHAL_INDEX                  20  // 0x14
1084  #define TPMI_DH_PERSISTENT_MARSHAL_INDEX              21  // 0x15
1085  #define TPMI_DH_ENTITY_MARSHAL_INDEX                  22  // 0x16
1086  #define TPMI_DH_PCR_MARSHAL_INDEX                     23  // 0x17
1087  #define TPMI_SH_AUTH_SESSION_MARSHAL_INDEX            24  // 0x18
1088  #define TPMI_SH_HMAC_MARSHAL_INDEX                    25  // 0x19
1089  #define TPMI_SH_POLICY_MARSHAL_INDEX                  26  // 0x1A
1090  #define TPMI_DH_CONTEXT_MARSHAL_INDEX                 27  // 0x1B
1091  #define TPMI_DH_SAVED_MARSHAL_INDEX                   28  // 0x1C
1092  #define TPMI_RH_HIERARCHY_MARSHAL_INDEX               29  // 0x1D
1093  #define TPMI_RH_ENABLES_MARSHAL_INDEX                 30  // 0x1E
1094  #define TPMI_RH_HIERARCHY_AUTH_MARSHAL_INDEX          31  // 0x1F
1095  #define TPMI_RH_PLATFORM_MARSHAL_INDEX                32  // 0x20
1096  #define TPMI_RH_OWNER_MARSHAL_INDEX                   33  // 0x21
1097  #define TPMI_RH_ENDORSEMENT_MARSHAL_INDEX             34  // 0x22
1098  #define TPMI_RH_PROVISION_MARSHAL_INDEX               35  // 0x23
1099  #define TPMI_RH_CLEAR_MARSHAL_INDEX                   36  // 0x24
1100  #define TPMI_RH_NV_AUTH_MARSHAL_INDEX                 37  // 0x25
1101  #define TPMI_RH_LOCKOUT_MARSHAL_INDEX                 38  // 0x26
1102  #define TPMI_RH_NV_INDEX_MARSHAL_INDEX                39  // 0x27
1103  #define TPMI_RH_AC_MARSHAL_INDEX                      40  // 0x28
1104  #define TPMI_ALG_HASH_MARSHAL_INDEX                   41  // 0x29
1105  #define TPMI_ALG_ASYM_MARSHAL_INDEX                   42  // 0x2A
1106  #define TPMI_ALG_SYM_MARSHAL_INDEX                    43  // 0x2B
1107  #define TPMI_ALG_SYM_OBJECT_MARSHAL_INDEX             44  // 0x2C
1108  #define TPMI_ALG_SYM_MODE_MARSHAL_INDEX               45  // 0x2D
1109  #define TPMI_ALG_KDF_MARSHAL_INDEX                    46  // 0x2E
1110  #define TPMI_ALG_SIG_SCHEME_MARSHAL_INDEX             47  // 0x2F
1111  #define TPMI_ECC_KEY_EXCHANGE_MARSHAL_INDEX           48  // 0x30
1112  #define TPMI_ST_COMMAND_TAG_MARSHAL_INDEX             49  // 0x31
1113  #define TPMI_ALG_MAC_SCHEME_MARSHAL_INDEX             50  // 0x32
1114  #define TPMI_ALG_CIPHER_MODE_MARSHAL_INDEX            51  // 0x33
1115  #define TPMS_EMPTY_MARSHAL_INDEX                      52  // 0x34
1116  #define TPMS_ENC_SCHEME_RSAES_MARSHAL_INDEX           TPMS_EMPTY_MARSHAL_INDEX
1117  #define TPMS_ALGORITHM_DESCRIPTION_MARSHAL_INDEX      53  // 0x35
1118  #define TPMU_HA_MARSHAL_INDEX                         54  // 0x36
1119  #define TPMT_HA_MARSHAL_INDEX                         55  // 0x37
1120  #define TPM2B_DIGEST_MARSHAL_INDEX                    56  // 0x38
1121  #define TPM2B_NONCE_MARSHAL_INDEX                     TPM2B_DIGEST_MARSHAL_INDEX
1122  #define TPM2B_AUTH_MARSHAL_INDEX                      TPM2B_DIGEST_MARSHAL_INDEX
1123  #define TPM2B_OPERAND_MARSHAL_INDEX                   TPM2B_DIGEST_MARSHAL_INDEX
1124  #define TPM2B_DATA_MARSHAL_INDEX                      57  // 0x39
1125  #define TPM2B_EVENT_MARSHAL_INDEX                     58  // 0x3A
1126  #define TPM2B_MAX_BUFFER_MARSHAL_INDEX                59  // 0x3B
1127  #define TPM2B_MAX_NV_BUFFER_MARSHAL_INDEX             60  // 0x3C
1128  #define TPM2B_TIMEOUT_MARSHAL_INDEX                   61  // 0x3D
1129  #define TPM2B_IV_MARSHAL_INDEX                        62  // 0x3E
1130  #define NULL_UNION_MARSHAL_INDEX                      63  // 0x3F
1131  #define TPMU_NAME_MARSHAL_INDEX                       NULL_UNION_MARSHAL_INDEX
1132  #define TPMU_SENSITIVE_CREATE_MARSHAL_INDEX           NULL_UNION_MARSHAL_INDEX
1133  #define TPM2B_NAME_MARSHAL_INDEX                      64  // 0x40
1134  #define TPMS_PCR_SELECT_MARSHAL_INDEX                 65  // 0x41
1135  #define TPMS_PCR_SELECTION_MARSHAL_INDEX              66  // 0x42
1136  #define TPMT_TK_CREATION_MARSHAL_INDEX                67  // 0x43
1137  #define TPMT_TK_VERIFIED_MARSHAL_INDEX                68  // 0x44
1138  #define TPMT_TK_AUTH_MARSHAL_INDEX                    69  // 0x45
1139  #define TPMT_TK_HASHCHECK_MARSHAL_INDEX               70  // 0x46
1140  #define TPMS_ALG_PROPERTY_MARSHAL_INDEX               71  // 0x47
1141  #define TPMS_TAGGED_PROPERTY_MARSHAL_INDEX            72  // 0x48
1142  #define TPMS_TAGGED_PCR_SELECT_MARSHAL_INDEX          73  // 0x49
```

```
1143 #define TPMS_TAGGED_POLICY_MARSHAL_INDEX          74 // 0x4A
1144 #define TPML_CC_MARSHAL_INDEX                     75 // 0x4B
1145 #define TPML_CCA_MARSHAL_INDEX                    76 // 0x4C
1146 #define TPML_ALG_MARSHAL_INDEX                    77 // 0x4D
1147 #define TPML_HANDLE_MARSHAL_INDEX                 78 // 0x4E
1148 #define TPML_DIGEST_MARSHAL_INDEX                 79 // 0x4F
1149 #define TPML_DIGEST_VALUES_MARSHAL_INDEX          80 // 0x50
1150 #define TPML_PCR_SELECTION_MARSHAL_INDEX          81 // 0x51
1151 #define TPML_ALG_PROPERTY_MARSHAL_INDEX           82 // 0x52
1152 #define TPML_TAGGED_TPM_PROPERTY_MARSHAL_INDEX    83 // 0x53
1153 #define TPML_TAGGED_PCR_PROPERTY_MARSHAL_INDEX    84 // 0x54
1154 #define TPML_ECC_CURVE_MARSHAL_INDEX              85 // 0x55
1155 #define TPML_TAGGED_POLICY_MARSHAL_INDEX          86 // 0x56
1156 #define TPMU_CAPABILITIES_MARSHAL_INDEX           87 // 0x57
1157 #define TPMS_CAPABILITY_DATA_MARSHAL_INDEX        88 // 0x58
1158 #define TPMS_CLOCK_INFO_MARSHAL_INDEX             89 // 0x59
1159 #define TPMS_TIME_INFO_MARSHAL_INDEX              90 // 0x5A
1160 #define TPMS_TIME_ATTEST_INFO_MARSHAL_INDEX       91 // 0x5B
1161 #define TPMS_CERTIFY_INFO_MARSHAL_INDEX           92 // 0x5C
1162 #define TPMS_QUOTE_INFO_MARSHAL_INDEX             93 // 0x5D
1163 #define TPMS_COMMAND_AUDIT_INFO_MARSHAL_INDEX     94 // 0x5E
1164 #define TPMS_SESSION_AUDIT_INFO_MARSHAL_INDEX     95 // 0x5F
1165 #define TPMS_CREATION_INFO_MARSHAL_INDEX          96 // 0x60
1166 #define TPMS_NV_CERTIFY_INFO_MARSHAL_INDEX        97 // 0x61
1167 #define TPMS_NV_DIGEST_CERTIFY_INFO_MARSHAL_INDEX 98 // 0x62
1168 #define TPMI_ST_ATTEST_MARSHAL_INDEX              99 // 0x63
1169 #define TPMU_ATTEST_MARSHAL_INDEX                 100 // 0x64
1170 #define TPMS_ATTEST_MARSHAL_INDEX                 101 // 0x65
1171 #define TPM2B_ATTEST_MARSHAL_INDEX                102 // 0x66
1172 #define TPMS_AUTH_COMMAND_MARSHAL_INDEX           103 // 0x67
1173 #define TPMS_AUTH_RESPONSE_MARSHAL_INDEX          104 // 0x68
1174 #define TPMI_TDES_KEY_BITS_MARSHAL_INDEX          105 // 0x69
1175 #define TPMI_AES_KEY_BITS_MARSHAL_INDEX           106 // 0x6A
1176 #define TPMI_SM4_KEY_BITS_MARSHAL_INDEX           107 // 0x6B
1177 #define TPMI_CAMELLIA_KEY_BITS_MARSHAL_INDEX      108 // 0x6C
1178 #define TPMU_SYM_KEY_BITS_MARSHAL_INDEX           109 // 0x6D
1179 #define TPMU_SYM_MODE_MARSHAL_INDEX               110 // 0x6E
1180 #define TPMT_SYM_DEF_MARSHAL_INDEX                111 // 0x6F
1181 #define TPMT_SYM_DEF_OBJECT_MARSHAL_INDEX         112 // 0x70
1182 #define TPM2B_SYM_KEY_MARSHAL_INDEX               113 // 0x71
1183 #define TPMS_SYMCIPHER_PARMS_MARSHAL_INDEX        114 // 0x72
1184 #define TPM2B_LABEL_MARSHAL_INDEX                 115 // 0x73
1185 #define TPMS_DERIVE_MARSHAL_INDEX                 116 // 0x74
1186 #define TPM2B_DERIVE_MARSHAL_INDEX                117 // 0x75
1187 #define TPM2B_SENSITIVE_DATA_MARSHAL_INDEX        118 // 0x76
1188 #define TPMS_SENSITIVE_CREATE_MARSHAL_INDEX       119 // 0x77
1189 #define TPM2B_SENSITIVE_CREATE_MARSHAL_INDEX      120 // 0x78
1190 #define TPMS_SCHEME_HASH_MARSHAL_INDEX            121 // 0x79
1191 #define TPMS_SCHEME_HMAC_MARSHAL_INDEX            TPMS_SCHEME_HASH_MARSHAL_INDEX
1192 #define TPMS_SIG_SCHEME_RSASSA_MARSHAL_INDEX      TPMS_SCHEME_HASH_MARSHAL_INDEX
1193 #define TPMS_SIG_SCHEME_RSAPSS_MARSHAL_INDEX      TPMS_SCHEME_HASH_MARSHAL_INDEX
1194 #define TPMS_SIG_SCHEME_ECDSA_MARSHAL_INDEX       TPMS_SCHEME_HASH_MARSHAL_INDEX
1195 #define TPMS_SIG_SCHEME_SM2_MARSHAL_INDEX         TPMS_SCHEME_HASH_MARSHAL_INDEX
1196 #define TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_INDEX   TPMS_SCHEME_HASH_MARSHAL_INDEX
1197 #define TPMS_ENC_SCHEME_OAEP_MARSHAL_INDEX        TPMS_SCHEME_HASH_MARSHAL_INDEX
1198 #define TPMS_KEY_SCHEME_ECDH_MARSHAL_INDEX        TPMS_SCHEME_HASH_MARSHAL_INDEX
1199 #define TPMS_KEY_SCHEME_ECMQV_MARSHAL_INDEX       TPMS_SCHEME_HASH_MARSHAL_INDEX
1200 #define TPMS_SCHEME_MGF1_MARSHAL_INDEX            TPMS_SCHEME_HASH_MARSHAL_INDEX
1201 #define TPMS_SCHEME_KDF1_SP800_56A_MARSHAL_INDEX  TPMS_SCHEME_HASH_MARSHAL_INDEX
1202 #define TPMS_SCHEME_KDF2_MARSHAL_INDEX            TPMS_SCHEME_HASH_MARSHAL_INDEX
1203 #define TPMS_SCHEME_KDF1_SP800_108_MARSHAL_INDEX  TPMS_SCHEME_HASH_MARSHAL_INDEX
1204 #define TPMS_SCHEME_ECDAA_MARSHAL_INDEX           122 // 0x7A
1205 #define TPMS_SIG_SCHEME_ECDAA_MARSHAL_INDEX       TPMS_SCHEME_ECDAA_MARSHAL_INDEX
1206 #define TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_INDEX   123 // 0x7B
1207 #define TPMS_SCHEME_XOR_MARSHAL_INDEX             124 // 0x7C
1208 #define TPMU_SCHEME_KEYEDHASH_MARSHAL_INDEX       125 // 0x7D
```

```
1209    #define TPMT_KEYEDHASH_SCHEME_MARSHAL_INDEX         126 // 0x7E
1210    #define TPMU_SIG_SCHEME_MARSHAL_INDEX               127 // 0x7F
1211    #define TPMT_SIG_SCHEME_MARSHAL_INDEX               128 // 0x80
1212    #define TPMU_KDF_SCHEME_MARSHAL_INDEX               129 // 0x81
1213    #define TPMT_KDF_SCHEME_MARSHAL_INDEX               130 // 0x82
1214    #define TPMI_ALG_ASYM_SCHEME_MARSHAL_INDEX          131 // 0x83
1215    #define TPMU_ASYM_SCHEME_MARSHAL_INDEX              132 // 0x84
1216    #define TPMT_ASYM_SCHEME_MARSHAL_INDEX              133 // 0x85
1217    #define TPMI_ALG_RSA_SCHEME_MARSHAL_INDEX           134 // 0x86
1218    #define TPMT_RSA_SCHEME_MARSHAL_INDEX               135 // 0x87
1219    #define TPMI_ALG_RSA_DECRYPT_MARSHAL_INDEX          136 // 0x88
1220    #define TPMT_RSA_DECRYPT_MARSHAL_INDEX              137 // 0x89
1221    #define TPM2B_PUBLIC_KEY_RSA_MARSHAL_INDEX          138 // 0x8A
1222    #define TPMI_RSA_KEY_BITS_MARSHAL_INDEX             139 // 0x8B
1223    #define TPM2B_PRIVATE_KEY_RSA_MARSHAL_INDEX         140 // 0x8C
1224    #define TPM2B_ECC_PARAMETER_MARSHAL_INDEX           141 // 0x8D
1225    #define TPMS_ECC_POINT_MARSHAL_INDEX                142 // 0x8E
1226    #define TPM2B_ECC_POINT_MARSHAL_INDEX               143 // 0x8F
1227    #define TPMI_ALG_ECC_SCHEME_MARSHAL_INDEX           144 // 0x90
1228    #define TPMI_ECC_CURVE_MARSHAL_INDEX                145 // 0x91
1229    #define TPMT_ECC_SCHEME_MARSHAL_INDEX               146 // 0x92
1230    #define TPMS_ALGORITHM_DETAIL_ECC_MARSHAL_INDEX     147 // 0x93
1231    #define TPMS_SIGNATURE_RSA_MARSHAL_INDEX            148 // 0x94
1232    #define TPMS_SIGNATURE_RSASSA_MARSHAL_INDEX         TPMS_SIGNATURE_RSA_MARSHAL_INDEX
1233    #define TPMS_SIGNATURE_RSAPSS_MARSHAL_INDEX         TPMS_SIGNATURE_RSA_MARSHAL_INDEX
1234    #define TPMS_SIGNATURE_ECC_MARSHAL_INDEX            149 // 0x95
1235    #define TPMS_SIGNATURE_ECDAA_MARSHAL_INDEX          TPMS_SIGNATURE_ECC_MARSHAL_INDEX
1236    #define TPMS_SIGNATURE_ECDSA_MARSHAL_INDEX          TPMS_SIGNATURE_ECC_MARSHAL_INDEX
1237    #define TPMS_SIGNATURE_SM2_MARSHAL_INDEX            TPMS_SIGNATURE_ECC_MARSHAL_INDEX
1238    #define TPMS_SIGNATURE_ECSCHNORR_MARSHAL_INDEX      TPMS_SIGNATURE_ECC_MARSHAL_INDEX
1239    #define TPMU_SIGNATURE_MARSHAL_INDEX                150 // 0x96
1240    #define TPMT_SIGNATURE_MARSHAL_INDEX                151 // 0x97
1241    #define TPMU_ENCRYPTED_SECRET_MARSHAL_INDEX         152 // 0x98
1242    #define TPM2B_ENCRYPTED_SECRET_MARSHAL_INDEX        153 // 0x99
1243    #define TPMI_ALG_PUBLIC_MARSHAL_INDEX               154 // 0x9A
1244    #define TPMU_PUBLIC_ID_MARSHAL_INDEX                155 // 0x9B
1245    #define TPMS_KEYEDHASH_PARMS_MARSHAL_INDEX          156 // 0x9C
1246    #define TPMS_ASYM_PARMS_MARSHAL_INDEX               157 // 0x9D
1247    #define TPMS_RSA_PARMS_MARSHAL_INDEX                158 // 0x9E
1248    #define TPMS_ECC_PARMS_MARSHAL_INDEX                159 // 0x9F
1249    #define TPMU_PUBLIC_PARMS_MARSHAL_INDEX             160 // 0xA0
1250    #define TPMT_PUBLIC_PARMS_MARSHAL_INDEX             161 // 0xA1
1251    #define TPMT_PUBLIC_MARSHAL_INDEX                   162 // 0xA2
1252    #define TPM2B_PUBLIC_MARSHAL_INDEX                  163 // 0xA3
1253    #define TPM2B_TEMPLATE_MARSHAL_INDEX                164 // 0xA4
1254    #define TPM2B_PRIVATE_VENDOR_SPECIFIC_MARSHAL_INDEX 165 // 0xA5
1255    #define TPMU_SENSITIVE_COMPOSITE_MARSHAL_INDEX      166 // 0xA6
1256    #define TPMT_SENSITIVE_MARSHAL_INDEX                167 // 0xA7
1257    #define TPM2B_SENSITIVE_MARSHAL_INDEX               168 // 0xA8
1258    #define _PRIVATE_MARSHAL_INDEX                      169 // 0xA9
1259    #define TPM2B_PRIVATE_MARSHAL_INDEX                 170 // 0xAA
1260    #define TPMS_ID_OBJECT_MARSHAL_INDEX                171 // 0xAB
1261    #define TPM2B_ID_OBJECT_MARSHAL_INDEX               172 // 0xAC
1262    #define TPMS_NV_PIN_COUNTER_PARAMETERS_MARSHAL_INDEX    173 // 0xAD
1263    #define TPMA_NV_MARSHAL_INDEX                       174 // 0xAE
1264    #define TPMS_NV_PUBLIC_MARSHAL_INDEX                175 // 0xAF
1265    #define TPM2B_NV_PUBLIC_MARSHAL_INDEX               176 // 0xB0
1266    #define TPM2B_CONTEXT_SENSITIVE_MARSHAL_INDEX       177 // 0xB1
1267    #define TPMS_CONTEXT_DATA_MARSHAL_INDEX             178 // 0xB2
1268    #define TPM2B_CONTEXT_DATA_MARSHAL_INDEX            179 // 0xB3
1269    #define TPMS_CONTEXT_MARSHAL_INDEX                  180 // 0xB4
1270    #define TPMS_CREATION_DATA_MARSHAL_INDEX            181 // 0xB5
1271    #define TPM2B_CREATION_DATA_MARSHAL_INDEX           182 // 0xB6
1272    #define TPM_AT_MARSHAL_INDEX                        183 // 0xB7
1273    #define TPMS_AC_OUTPUT_MARSHAL_INDEX                184 // 0xB8
1274    #define TPML_AC_CAPABILITIES_MARSHAL_INDEX          185 // 0xB9
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1275 #define Type00_MARSHAL_INDEX                        186 // 0xBA
1276 #define Type01_MARSHAL_INDEX                        187 // 0xBB
1277 #define Type02_MARSHAL_INDEX                        188 // 0xBC
1278 #define Type03_MARSHAL_INDEX                        189 // 0xBD
1279 #define Type04_MARSHAL_INDEX                        190 // 0xBE
1280 #define Type05_MARSHAL_INDEX                        191 // 0xBF
1281 #define Type06_MARSHAL_INDEX                        192 // 0xC0
1282 #define Type07_MARSHAL_INDEX                        193 // 0xC1
1283 #define Type08_MARSHAL_INDEX                        194 // 0xC2
1284 #define Type09_MARSHAL_INDEX                        Type08_MARSHAL_INDEX
1285 #define Type14_MARSHAL_INDEX                        Type08_MARSHAL_INDEX
1286 #define Type10_MARSHAL_INDEX                        195 // 0xC3
1287 #define Type11_MARSHAL_INDEX                        196 // 0xC4
1288 #define Type12_MARSHAL_INDEX                        197 // 0xC5
1289 #define Type13_MARSHAL_INDEX                        198 // 0xC6
1290 #define Type15_MARSHAL_INDEX                        199 // 0xC7
1291 #define Type16_MARSHAL_INDEX                        Type15_MARSHAL_INDEX
1292 #define Type17_MARSHAL_INDEX                        200 // 0xC8
1293 #define Type18_MARSHAL_INDEX                        201 // 0xC9
1294 #define Type19_MARSHAL_INDEX                        202 // 0xCA
1295 #define Type20_MARSHAL_INDEX                        203 // 0xCB
1296 #define Type21_MARSHAL_INDEX                        Type20_MARSHAL_INDEX
1297 #define Type22_MARSHAL_INDEX                        204 // 0xCC
1298 #define Type23_MARSHAL_INDEX                        205 // 0xCD
1299 #define Type24_MARSHAL_INDEX                        206 // 0xCE
1300 #define Type25_MARSHAL_INDEX                        207 // 0xCF
1301 #define Type26_MARSHAL_INDEX                        208 // 0xD0
1302 #define Type27_MARSHAL_INDEX                        209 // 0xD1
1303 #define Type28_MARSHAL_INDEX                        210 // 0xD2
1304 #define Type29_MARSHAL_INDEX                        211 // 0xD3
1305 #define Type30_MARSHAL_INDEX                        212 // 0xD4
1306 #define Type31_MARSHAL_INDEX                        213 // 0xD5
1307 #define Type32_MARSHAL_INDEX                        214 // 0xD6
1308 #define Type33_MARSHAL_INDEX                        215 // 0xD7
1309 #define Type34_MARSHAL_INDEX                        216 // 0xD8
1310 #define Type35_MARSHAL_INDEX                        217 // 0xD9
1311 #define Type36_MARSHAL_INDEX                        218 // 0xDA
1312 #define Type37_MARSHAL_INDEX                        219 // 0xDB
1313 #define Type38_MARSHAL_INDEX                        220 // 0xDC
1314 #define Type39_MARSHAL_INDEX                        221 // 0xDD
1315 #define Type40_MARSHAL_INDEX                        222 // 0xDE
1316 #define Type41_MARSHAL_INDEX                        223 // 0xDF
1317 #define Type42_MARSHAL_INDEX                        224 // 0xE0
1318 #define Type43_MARSHAL_INDEX                        225 // 0xE1
1319 //#defines to change calling sequence for code using marshaling
1320 #define UINT8_Unmarshal(target, buffer, size)                               \
1321     Unmarshal(UINT8_MARSHAL_INDEX, (target), (buffer), (size))
1322 #define UINT8_Marshal(source, buffer, size)                                 \
1323     Marshal(UINT8_MARSHAL_INDEX, (source), (buffer), (size))
1324 #define BYTE_Unmarshal(target, buffer, size)                                \
1325     Unmarshal(UINT8_MARSHAL_INDEX, (target), (buffer), (size))
1326 #define BYTE_Marshal(source, buffer, size)                                  \
1327     Marshal(UINT8_MARSHAL_INDEX, (source), (buffer), (size))
1328 #define INT8_Unmarshal(target, buffer, size)                                \
1329     Unmarshal(INT8_MARSHAL_INDEX, (target), (buffer), (size))
1330 #define INT8_Marshal(source, buffer, size)                                  \
1331     Marshal(INT8_MARSHAL_INDEX, (source), (buffer), (size))
1332 #define UINT16_Unmarshal(target, buffer, size)                              \
1333     Unmarshal(UINT16_MARSHAL_INDEX, (target), (buffer), (size))
1334 #define UINT16_Marshal(source, buffer, size)                                \
1335     Marshal(UINT16_MARSHAL_INDEX, (source), (buffer), (size))
1336 #define INT16_Unmarshal(target, buffer, size)                               \
1337     Unmarshal(INT16_MARSHAL_INDEX, (target), (buffer), (size))
1338 #define INT16_Marshal(source, buffer, size)                                 \
1339     Marshal(INT16_MARSHAL_INDEX, (source), (buffer), (size))
1340 #define UINT32_Unmarshal(target, buffer, size)                              \
```

```
1341        Unmarshal(UINT32_MARSHAL_INDEX, (target), (buffer), (size))
1342    #define UINT32_Marshal(source, buffer, size)                                   \
1343        Marshal(UINT32_MARSHAL_INDEX, (source), (buffer), (size))
1344    #define INT32_Unmarshal(target, buffer, size)                                  \
1345        Unmarshal(INT32_MARSHAL_INDEX, (target), (buffer), (size))
1346    #define INT32_Marshal(source, buffer, size)                                    \
1347        Marshal(INT32_MARSHAL_INDEX, (source), (buffer), (size))
1348    #define UINT64_Unmarshal(target, buffer, size)                                 \
1349        Unmarshal(UINT64_MARSHAL_INDEX, (target), (buffer), (size))
1350    #define UINT64_Marshal(source, buffer, size)                                   \
1351        Marshal(UINT64_MARSHAL_INDEX, (source), (buffer), (size))
1352    #define INT64_Unmarshal(target, buffer, size)                                  \
1353        Unmarshal(INT64_MARSHAL_INDEX, (target), (buffer), (size))
1354    #define INT64_Marshal(source, buffer, size)                                    \
1355        Marshal(INT64_MARSHAL_INDEX, (source), (buffer), (size))
1356    #define TPM_ALGORITHM_ID_Unmarshal(target, buffer, size)                       \
1357        Unmarshal(TPM_ALGORITHM_ID_MARSHAL_INDEX, (target), (buffer), (size))
1358    #define TPM_ALGORITHM_ID_Marshal(source, buffer, size)                         \
1359        Marshal(TPM_ALGORITHM_ID_MARSHAL_INDEX, (source), (buffer), (size))
1360    #define TPM_MODIFIER_INDICATOR_Unmarshal(target, buffer, size)                 \
1361        Unmarshal(TPM_MODIFIER_INDICATOR_MARSHAL_INDEX, (target), (buffer), (size))
1362    #define TPM_MODIFIER_INDICATOR_Marshal(source, buffer, size)                   \
1363        Marshal(TPM_MODIFIER_INDICATOR_MARSHAL_INDEX, (source), (buffer), (size))
1364    #define TPM_AUTHORIZATION_SIZE_Unmarshal(target, buffer, size)                 \
1365        Unmarshal(TPM_AUTHORIZATION_SIZE_MARSHAL_INDEX, (target), (buffer), (size))
1366    #define TPM_AUTHORIZATION_SIZE_Marshal(source, buffer, size)                   \
1367        Marshal(TPM_AUTHORIZATION_SIZE_MARSHAL_INDEX, (source), (buffer), (size))
1368    #define TPM_PARAMETER_SIZE_Unmarshal(target, buffer, size)                     \
1369        Unmarshal(TPM_PARAMETER_SIZE_MARSHAL_INDEX, (target), (buffer), (size))
1370    #define TPM_PARAMETER_SIZE_Marshal(source, buffer, size)                       \
1371        Marshal(TPM_PARAMETER_SIZE_MARSHAL_INDEX, (source), (buffer), (size))
1372    #define TPM_KEY_SIZE_Unmarshal(target, buffer, size)                           \
1373        Unmarshal(TPM_KEY_SIZE_MARSHAL_INDEX, (target), (buffer), (size))
1374    #define TPM_KEY_SIZE_Marshal(source, buffer, size)                             \
1375        Marshal(TPM_KEY_SIZE_MARSHAL_INDEX, (source), (buffer), (size))
1376    #define TPM_KEY_BITS_Unmarshal(target, buffer, size)                           \
1377        Unmarshal(TPM_KEY_BITS_MARSHAL_INDEX, (target), (buffer), (size))
1378    #define TPM_KEY_BITS_Marshal(source, buffer, size)                             \
1379        Marshal(TPM_KEY_BITS_MARSHAL_INDEX, (source), (buffer), (size))
1380    #define TPM_GENERATED_Marshal(source, buffer, size)                            \
1381        Marshal(TPM_GENERATED_MARSHAL_INDEX, (source), (buffer), (size))
1382    #define TPM_ALG_ID_Unmarshal(target, buffer, size)                             \
1383        Unmarshal(TPM_ALG_ID_MARSHAL_INDEX, (target), (buffer), (size))
1384    #define TPM_ALG_ID_Marshal(source, buffer, size)                               \
1385        Marshal(TPM_ALG_ID_MARSHAL_INDEX, (source), (buffer), (size))
1386    #define TPM_ECC_CURVE_Unmarshal(target, buffer, size)                          \
1387        Unmarshal(TPM_ECC_CURVE_MARSHAL_INDEX, (target), (buffer), (size))
1388    #define TPM_ECC_CURVE_Marshal(source, buffer, size)                            \
1389        Marshal(TPM_ECC_CURVE_MARSHAL_INDEX, (source), (buffer), (size))
1390    #define TPM_CC_Unmarshal(target, buffer, size)                                 \
1391        Unmarshal(TPM_CC_MARSHAL_INDEX, (target), (buffer), (size))
1392    #define TPM_CC_Marshal(source, buffer, size)                                   \
1393        Marshal(TPM_CC_MARSHAL_INDEX, (source), (buffer), (size))
1394    #define TPM_RC_Marshal(source, buffer, size)                                   \
1395        Marshal(TPM_RC_MARSHAL_INDEX, (source), (buffer), (size))
1396    #define TPM_CLOCK_ADJUST_Unmarshal(target, buffer, size)                       \
1397        Unmarshal(TPM_CLOCK_ADJUST_MARSHAL_INDEX, (target), (buffer), (size))
1398    #define TPM_EO_Unmarshal(target, buffer, size)                                 \
1399        Unmarshal(TPM_EO_MARSHAL_INDEX, (target), (buffer), (size))
1400    #define TPM_EO_Marshal(source, buffer, size)                                   \
1401        Marshal(TPM_EO_MARSHAL_INDEX, (source), (buffer), (size))
1402    #define TPM_ST_Unmarshal(target, buffer, size)                                 \
1403        Unmarshal(TPM_ST_MARSHAL_INDEX, (target), (buffer), (size))
1404    #define TPM_ST_Marshal(source, buffer, size)                                   \
1405        Marshal(TPM_ST_MARSHAL_INDEX, (source), (buffer), (size))
1406    #define TPM_SU_Unmarshal(target, buffer, size)                                 \
```

```
1407     Unmarshal(TPM_SU_MARSHAL_INDEX, (target), (buffer), (size))
1408 #define TPM_SE_Unmarshal(target, buffer, size)                                   \
1409     Unmarshal(TPM_SE_MARSHAL_INDEX, (target), (buffer), (size))
1410 #define TPM_CAP_Unmarshal(target, buffer, size)                                  \
1411     Unmarshal(TPM_CAP_MARSHAL_INDEX, (target), (buffer), (size))
1412 #define TPM_CAP_Marshal(source, buffer, size)                                    \
1413     Marshal(TPM_CAP_MARSHAL_INDEX, (source), (buffer), (size))
1414 #define TPM_PT_Unmarshal(target, buffer, size)                                   \
1415     Unmarshal(TPM_PT_MARSHAL_INDEX, (target), (buffer), (size))
1416 #define TPM_PT_Marshal(source, buffer, size)                                     \
1417     Marshal(TPM_PT_MARSHAL_INDEX, (source), (buffer), (size))
1418 #define TPM_PT_PCR_Unmarshal(target, buffer, size)                               \
1419     Unmarshal(TPM_PT_PCR_MARSHAL_INDEX, (target), (buffer), (size))
1420 #define TPM_PT_PCR_Marshal(source, buffer, size)                                 \
1421     Marshal(TPM_PT_PCR_MARSHAL_INDEX, (source), (buffer), (size))
1422 #define TPM_PS_Marshal(source, buffer, size)                                     \
1423     Marshal(TPM_PS_MARSHAL_INDEX, (source), (buffer), (size))
1424 #define TPM_HANDLE_Unmarshal(target, buffer, size)                               \
1425     Unmarshal(TPM_HANDLE_MARSHAL_INDEX, (target), (buffer), (size))
1426 #define TPM_HANDLE_Marshal(source, buffer, size)                                 \
1427     Marshal(TPM_HANDLE_MARSHAL_INDEX, (source), (buffer), (size))
1428 #define TPM_HT_Unmarshal(target, buffer, size)                                   \
1429     Unmarshal(TPM_HT_MARSHAL_INDEX, (target), (buffer), (size))
1430 #define TPM_HT_Marshal(source, buffer, size)                                     \
1431     Marshal(TPM_HT_MARSHAL_INDEX, (source), (buffer), (size))
1432 #define TPM_RH_Unmarshal(target, buffer, size)                                   \
1433     Unmarshal(TPM_RH_MARSHAL_INDEX, (target), (buffer), (size))
1434 #define TPM_RH_Marshal(source, buffer, size)                                     \
1435     Marshal(TPM_RH_MARSHAL_INDEX, (source), (buffer), (size))
1436 #define TPM_HC_Unmarshal(target, buffer, size)                                   \
1437     Unmarshal(TPM_HC_MARSHAL_INDEX, (target), (buffer), (size))
1438 #define TPM_HC_Marshal(source, buffer, size)                                     \
1439     Marshal(TPM_HC_MARSHAL_INDEX, (source), (buffer), (size))
1440 #define TPMA_ALGORITHM_Unmarshal(target, buffer, size)                           \
1441     Unmarshal(TPMA_ALGORITHM_MARSHAL_INDEX, (target), (buffer), (size))
1442 #define TPMA_ALGORITHM_Marshal(source, buffer, size)                             \
1443     Marshal(TPMA_ALGORITHM_MARSHAL_INDEX, (source), (buffer), (size))
1444 #define TPMA_OBJECT_Unmarshal(target, buffer, size)                              \
1445     Unmarshal(TPMA_OBJECT_MARSHAL_INDEX, (target), (buffer), (size))
1446 #define TPMA_OBJECT_Marshal(source, buffer, size)                                \
1447     Marshal(TPMA_OBJECT_MARSHAL_INDEX, (source), (buffer), (size))
1448 #define TPMA_SESSION_Unmarshal(target, buffer, size)                             \
1449     Unmarshal(TPMA_SESSION_MARSHAL_INDEX, (target), (buffer), (size))
1450 #define TPMA_SESSION_Marshal(source, buffer, size)                               \
1451     Marshal(TPMA_SESSION_MARSHAL_INDEX, (source), (buffer), (size))
1452 #define TPMA_LOCALITY_Unmarshal(target, buffer, size)                            \
1453     Unmarshal(TPMA_LOCALITY_MARSHAL_INDEX, (target), (buffer), (size))
1454 #define TPMA_LOCALITY_Marshal(source, buffer, size)                              \
1455     Marshal(TPMA_LOCALITY_MARSHAL_INDEX, (source), (buffer), (size))
1456 #define TPMA_PERMANENT_Marshal(source, buffer, size)                             \
1457     Marshal(TPMA_PERMANENT_MARSHAL_INDEX, (source), (buffer), (size))
1458 #define TPMA_STARTUP_CLEAR_Marshal(source, buffer, size)                         \
1459     Marshal(TPMA_STARTUP_CLEAR_MARSHAL_INDEX, (source), (buffer), (size))
1460 #define TPMA_MEMORY_Marshal(source, buffer, size)                                \
1461     Marshal(TPMA_MEMORY_MARSHAL_INDEX, (source), (buffer), (size))
1462 #define TPMA_CC_Marshal(source, buffer, size)                                    \
1463     Marshal(TPMA_CC_MARSHAL_INDEX, (source), (buffer), (size))
1464 #define TPMA_MODES_Marshal(source, buffer, size)                                 \
1465     Marshal(TPMA_MODES_MARSHAL_INDEX, (source), (buffer), (size))
1466 #define TPMA_X509_KEY_USAGE_Marshal(source, buffer, size)                        \
1467     Marshal(TPMA_X509_KEY_USAGE_MARSHAL_INDEX, (source), (buffer), (size))
1468 #define TPMI_YES_NO_Unmarshal(target, buffer, size)                              \
1469     Unmarshal(TPMI_YES_NO_MARSHAL_INDEX, (target), (buffer), (size))
1470 #define TPMI_YES_NO_Marshal(source, buffer, size)                                \
1471     Marshal(TPMI_YES_NO_MARSHAL_INDEX, (source), (buffer), (size))
1472 #define TPMI_DH_OBJECT_Unmarshal(target, buffer, size, flag)                     \
```

 Copyright © TCG 2006-2020

```
1473        Unmarshal(TPMI_DH_OBJECT_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),      \
1474          (buffer), (size))
1475    #define TPMI_DH_OBJECT_Marshal(source, buffer, size)                                \
1476        Marshal(TPMI_DH_OBJECT_MARSHAL_INDEX, (source), (buffer), (size))
1477    #define TPMI_DH_PARENT_Unmarshal(target, buffer, size, flag)                        \
1478        Unmarshal(TPMI_DH_PARENT_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),      \
1479          (buffer), (size))
1480    #define TPMI_DH_PARENT_Marshal(source, buffer, size)                                \
1481        Marshal(TPMI_DH_PARENT_MARSHAL_INDEX, (source), (buffer), (size))
1482    #define TPMI_DH_PERSISTENT_Unmarshal(target, buffer, size)                          \
1483        Unmarshal(TPMI_DH_PERSISTENT_MARSHAL_INDEX, (target), (buffer), (size))
1484    #define TPMI_DH_PERSISTENT_Marshal(source, buffer, size)                            \
1485        Marshal(TPMI_DH_PERSISTENT_MARSHAL_INDEX, (source), (buffer), (size))
1486    #define TPMI_DH_ENTITY_Unmarshal(target, buffer, size, flag)                        \
1487        Unmarshal(TPMI_DH_ENTITY_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),      \
1488          (buffer), (size))
1489    #define TPMI_DH_PCR_Unmarshal(target, buffer, size, flag)                           \
1490        Unmarshal(TPMI_DH_PCR_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),         \
1491          (buffer), (size))
1492    #define TPMI_SH_AUTH_SESSION_Unmarshal(target, buffer, size, flag)                  \
1493        Unmarshal(TPMI_SH_AUTH_SESSION_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),\
1494          (buffer), (size))
1495    #define TPMI_SH_AUTH_SESSION_Marshal(source, buffer, size)                          \
1496        Marshal(TPMI_SH_AUTH_SESSION_MARSHAL_INDEX, (source), (buffer), (size))
1497    #define TPMI_SH_HMAC_Unmarshal(target, buffer, size)                                \
1498        Unmarshal(TPMI_SH_HMAC_MARSHAL_INDEX, (target), (buffer), (size))
1499    #define TPMI_SH_HMAC_Marshal(source, buffer, size)                                  \
1500        Marshal(TPMI_SH_HMAC_MARSHAL_INDEX, (source), (buffer), (size))
1501    #define TPMI_SH_POLICY_Unmarshal(target, buffer, size)                              \
1502        Unmarshal(TPMI_SH_POLICY_MARSHAL_INDEX, (target), (buffer), (size))
1503    #define TPMI_SH_POLICY_Marshal(source, buffer, size)                                \
1504        Marshal(TPMI_SH_POLICY_MARSHAL_INDEX, (source), (buffer), (size))
1505    #define TPMI_DH_CONTEXT_Unmarshal(target, buffer, size)                             \
1506        Unmarshal(TPMI_DH_CONTEXT_MARSHAL_INDEX, (target), (buffer), (size))
1507    #define TPMI_DH_CONTEXT_Marshal(source, buffer, size)                               \
1508        Marshal(TPMI_DH_CONTEXT_MARSHAL_INDEX, (source), (buffer), (size))
1509    #define TPMI_DH_SAVED_Unmarshal(target, buffer, size)                               \
1510        Unmarshal(TPMI_DH_SAVED_MARSHAL_INDEX, (target), (buffer), (size))
1511    #define TPMI_DH_SAVED_Marshal(source, buffer, size)                                 \
1512        Marshal(TPMI_DH_SAVED_MARSHAL_INDEX, (source), (buffer), (size))
1513    #define TPMI_RH_HIERARCHY_Unmarshal(target, buffer, size, flag)                     \
1514        Unmarshal(TPMI_RH_HIERARCHY_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),   \
1515          (buffer), (size))
1516    #define TPMI_RH_HIERARCHY_Marshal(source, buffer, size)                             \
1517        Marshal(TPMI_RH_HIERARCHY_MARSHAL_INDEX, (source), (buffer), (size))
1518    #define TPMI_RH_ENABLES_Unmarshal(target, buffer, size, flag)                       \
1519        Unmarshal(TPMI_RH_ENABLES_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),     \
1520          (buffer), (size))
1521    #define TPMI_RH_ENABLES_Marshal(source, buffer, size)                               \
1522        Marshal(TPMI_RH_ENABLES_MARSHAL_INDEX, (source), (buffer), (size))
1523    #define TPMI_RH_HIERARCHY_AUTH_Unmarshal(target, buffer, size)                      \
1524        Unmarshal(TPMI_RH_HIERARCHY_AUTH_MARSHAL_INDEX, (target), (buffer), (size))
1525    #define TPMI_RH_PLATFORM_Unmarshal(target, buffer, size)                            \
1526        Unmarshal(TPMI_RH_PLATFORM_MARSHAL_INDEX, (target), (buffer), (size))
1527    #define TPMI_RH_OWNER_Unmarshal(target, buffer, size, flag)                         \
1528        Unmarshal(TPMI_RH_OWNER_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),       \
1529          (buffer), (size))
1530    #define TPMI_RH_ENDORSEMENT_Unmarshal(target, buffer, size, flag)                   \
1531        Unmarshal(TPMI_RH_ENDORSEMENT_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target), \
1532          (buffer), (size))
1533    #define TPMI_RH_PROVISION_Unmarshal(target, buffer, size)                           \
1534        Unmarshal(TPMI_RH_PROVISION_MARSHAL_INDEX, (target), (buffer), (size))
1535    #define TPMI_RH_CLEAR_Unmarshal(target, buffer, size)                               \
1536        Unmarshal(TPMI_RH_CLEAR_MARSHAL_INDEX, (target), (buffer), (size))
1537    #define TPMI_RH_NV_AUTH_Unmarshal(target, buffer, size)                             \
1538        Unmarshal(TPMI_RH_NV_AUTH_MARSHAL_INDEX, (target), (buffer), (size))
```

```
1539    #define TPMI_RH_LOCKOUT_Unmarshal(target, buffer, size)                    \
1540        Unmarshal(TPMI_RH_LOCKOUT_MARSHAL_INDEX, (target), (buffer), (size))
1541    #define TPMI_RH_NV_INDEX_Unmarshal(target, buffer, size)                   \
1542        Unmarshal(TPMI_RH_NV_INDEX_MARSHAL_INDEX, (target), (buffer), (size))
1543    #define TPMI_RH_NV_INDEX_Marshal(source, buffer, size)                     \
1544        Marshal(TPMI_RH_NV_INDEX_MARSHAL_INDEX, (source), (buffer), (size))
1545    #define TPMI_RH_AC_Unmarshal(target, buffer, size)                         \
1546        Unmarshal(TPMI_RH_AC_MARSHAL_INDEX, (target), (buffer), (size))
1547    #define TPMI_ALG_HASH_Unmarshal(target, buffer, size, flag)                \
1548        Unmarshal(TPMI_ALG_HASH_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),      \
1549          (buffer), (size))
1550    #define TPMI_ALG_HASH_Marshal(source, buffer, size)                        \
1551        Marshal(TPMI_ALG_HASH_MARSHAL_INDEX, (source), (buffer), (size))
1552    #define TPMI_ALG_ASYM_Unmarshal(target, buffer, size, flag)                \
1553        Unmarshal(TPMI_ALG_ASYM_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),      \
1554          (buffer), (size))
1555    #define TPMI_ALG_ASYM_Marshal(source, buffer, size)                        \
1556        Marshal(TPMI_ALG_ASYM_MARSHAL_INDEX, (source), (buffer), (size))
1557    #define TPMI_ALG_SYM_Unmarshal(target, buffer, size, flag)                 \
1558        Unmarshal(TPMI_ALG_SYM_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),       \
1559          (buffer), (size))
1560    #define TPMI_ALG_SYM_Marshal(source, buffer, size)                         \
1561        Marshal(TPMI_ALG_SYM_MARSHAL_INDEX, (source), (buffer), (size))
1562    #define TPMI_ALG_SYM_OBJECT_Unmarshal(target, buffer, size, flag)          \
1563        Unmarshal(TPMI_ALG_SYM_OBJECT_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target), \
1564          (buffer), (size))
1565    #define TPMI_ALG_SYM_OBJECT_Marshal(source, buffer, size)                  \
1566        Marshal(TPMI_ALG_SYM_OBJECT_MARSHAL_INDEX, (source), (buffer), (size))
1567    #define TPMI_ALG_SYM_MODE_Unmarshal(target, buffer, size, flag)            \
1568        Unmarshal(TPMI_ALG_SYM_MODE_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),   \
1569          (buffer), (size))
1570    #define TPMI_ALG_SYM_MODE_Marshal(source, buffer, size)                    \
1571        Marshal(TPMI_ALG_SYM_MODE_MARSHAL_INDEX, (source), (buffer), (size))
1572    #define TPMI_ALG_KDF_Unmarshal(target, buffer, size, flag)                 \
1573        Unmarshal(TPMI_ALG_KDF_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),       \
1574          (buffer), (size))
1575    #define TPMI_ALG_KDF_Marshal(source, buffer, size)                         \
1576        Marshal(TPMI_ALG_KDF_MARSHAL_INDEX, (source), (buffer), (size))
1577    #define TPMI_ALG_SIG_SCHEME_Unmarshal(target, buffer, size, flag)          \
1578        Unmarshal(TPMI_ALG_SIG_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target), \
1579          (buffer), (size))
1580    #define TPMI_ALG_SIG_SCHEME_Marshal(source, buffer, size)                  \
1581        Marshal(TPMI_ALG_SIG_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
1582    #define TPMI_ECC_KEY_EXCHANGE_Unmarshal(target, buffer, size, flag)        \
1583        Unmarshal(TPMI_ECC_KEY_EXCHANGE_MARSHAL_INDEX | (flag ? NULL_FLAG : 0),        \
1584          (target), (buffer), (size))
1585    #define TPMI_ECC_KEY_EXCHANGE_Marshal(source, buffer, size)                \
1586        Marshal(TPMI_ECC_KEY_EXCHANGE_MARSHAL_INDEX, (source), (buffer), (size))
1587    #define TPMI_ST_COMMAND_TAG_Unmarshal(target, buffer, size)                \
1588        Unmarshal(TPMI_ST_COMMAND_TAG_MARSHAL_INDEX, (target), (buffer), (size))
1589    #define TPMI_ST_COMMAND_TAG_Marshal(source, buffer, size)                  \
1590        Marshal(TPMI_ST_COMMAND_TAG_MARSHAL_INDEX, (source), (buffer), (size))
1591    #define TPMI_ALG_MAC_SCHEME_Unmarshal(target, buffer, size, flag)          \
1592        Unmarshal(TPMI_ALG_MAC_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target), \
1593          (buffer), (size))
1594    #define TPMI_ALG_MAC_SCHEME_Marshal(source, buffer, size)                  \
1595        Marshal(TPMI_ALG_MAC_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
1596    #define TPMI_ALG_CIPHER_MODE_Unmarshal(target, buffer, size, flag)         \
1597        Unmarshal(TPMI_ALG_CIPHER_MODE_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),\
1598          (buffer), (size))
1599    #define TPMI_ALG_CIPHER_MODE_Marshal(source, buffer, size)                 \
1600        Marshal(TPMI_ALG_CIPHER_MODE_MARSHAL_INDEX, (source), (buffer), (size))
1601    #define TPMS_EMPTY_Unmarshal(target, buffer, size)                         \
1602        Unmarshal(TPMS_EMPTY_MARSHAL_INDEX, (target), (buffer), (size))
1603    #define TPMS_EMPTY_Marshal(source, buffer, size)                           \
1604        Marshal(TPMS_EMPTY_MARSHAL_INDEX, (source), (buffer), (size))
```

 Copyright © TCG 2006-2020

```
1605 #define TPMS_ALGORITHM_DESCRIPTION_Marshal(source, buffer, size)                \
1606     Marshal(TPMS_ALGORITHM_DESCRIPTION_MARSHAL_INDEX, (source), (buffer), (size))
1607 #define TPMU_HA_Unmarshal(target, buffer, size, selector)                       \
1608     UnmarshalUnion(TPMU_HA_MARSHAL_INDEX, (target), (buffer), (size), (selector))
1609 #define TPMU_HA_Marshal(source, buffer, size, selector)                         \
1610     MarshalUnion(TPMU_HA_MARSHAL_INDEX, (target), (buffer), (size), (selector))
1611 #define TPMT_HA_Unmarshal(target, buffer, size, flag)                           \
1612     Unmarshal(TPMT_HA_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target), (buffer), \
1613       (size))
1614 #define TPMT_HA_Marshal(source, buffer, size)                                   \
1615     Marshal(TPMT_HA_MARSHAL_INDEX, (source), (buffer), (size))
1616 #define TPM2B_DIGEST_Unmarshal(target, buffer, size)                            \
1617     Unmarshal(TPM2B_DIGEST_MARSHAL_INDEX, (target), (buffer), (size))
1618 #define TPM2B_DIGEST_Marshal(source, buffer, size)                              \
1619     Marshal(TPM2B_DIGEST_MARSHAL_INDEX, (source), (buffer), (size))
1620 #define TPM2B_DATA_Unmarshal(target, buffer, size)                              \
1621     Unmarshal(TPM2B_DATA_MARSHAL_INDEX, (target), (buffer), (size))
1622 #define TPM2B_DATA_Marshal(source, buffer, size)                                \
1623     Marshal(TPM2B_DATA_MARSHAL_INDEX, (source), (buffer), (size))
1624 #define TPM2B_NONCE_Unmarshal(target, buffer, size)                             \
1625     Unmarshal(TPM2B_NONCE_MARSHAL_INDEX, (target), (buffer), (size))
1626 #define TPM2B_NONCE_Marshal(source, buffer, size)                               \
1627     Marshal(TPM2B_NONCE_MARSHAL_INDEX, (source), (buffer), (size))
1628 #define TPM2B_AUTH_Unmarshal(target, buffer, size)                              \
1629     Unmarshal(TPM2B_AUTH_MARSHAL_INDEX, (target), (buffer), (size))
1630 #define TPM2B_AUTH_Marshal(source, buffer, size)                                \
1631     Marshal(TPM2B_AUTH_MARSHAL_INDEX, (source), (buffer), (size))
1632 #define TPM2B_OPERAND_Unmarshal(target, buffer, size)                           \
1633     Unmarshal(TPM2B_OPERAND_MARSHAL_INDEX, (target), (buffer), (size))
1634 #define TPM2B_OPERAND_Marshal(source, buffer, size)                             \
1635     Marshal(TPM2B_OPERAND_MARSHAL_INDEX, (source), (buffer), (size))
1636 #define TPM2B_EVENT_Unmarshal(target, buffer, size)                             \
1637     Unmarshal(TPM2B_EVENT_MARSHAL_INDEX, (target), (buffer), (size))
1638 #define TPM2B_EVENT_Marshal(source, buffer, size)                               \
1639     Marshal(TPM2B_EVENT_MARSHAL_INDEX, (source), (buffer), (size))
1640 #define TPM2B_MAX_BUFFER_Unmarshal(target, buffer, size)                        \
1641     Unmarshal(TPM2B_MAX_BUFFER_MARSHAL_INDEX, (target), (buffer), (size))
1642 #define TPM2B_MAX_BUFFER_Marshal(source, buffer, size)                          \
1643     Marshal(TPM2B_MAX_BUFFER_MARSHAL_INDEX, (source), (buffer), (size))
1644 #define TPM2B_MAX_NV_BUFFER_Unmarshal(target, buffer, size)                     \
1645     Unmarshal(TPM2B_MAX_NV_BUFFER_MARSHAL_INDEX, (target), (buffer), (size))
1646 #define TPM2B_MAX_NV_BUFFER_Marshal(source, buffer, size)                       \
1647     Marshal(TPM2B_MAX_NV_BUFFER_MARSHAL_INDEX, (source), (buffer), (size))
1648 #define TPM2B_TIMEOUT_Unmarshal(target, buffer, size)                           \
1649     Unmarshal(TPM2B_TIMEOUT_MARSHAL_INDEX, (target), (buffer), (size))
1650 #define TPM2B_TIMEOUT_Marshal(source, buffer, size)                             \
1651     Marshal(TPM2B_TIMEOUT_MARSHAL_INDEX, (source), (buffer), (size))
1652 #define TPM2B_IV_Unmarshal(target, buffer, size)                                \
1653     Unmarshal(TPM2B_IV_MARSHAL_INDEX, (target), (buffer), (size))
1654 #define TPM2B_IV_Marshal(source, buffer, size)                                  \
1655     Marshal(TPM2B_IV_MARSHAL_INDEX, (source), (buffer), (size))
1656 #define TPM2B_NAME_Unmarshal(target, buffer, size)                              \
1657     Unmarshal(TPM2B_NAME_MARSHAL_INDEX, (target), (buffer), (size))
1658 #define TPM2B_NAME_Marshal(source, buffer, size)                                \
1659     Marshal(TPM2B_NAME_MARSHAL_INDEX, (source), (buffer), (size))
1660 #define TPMS_PCR_SELECT_Unmarshal(target, buffer, size)                         \
1661     Unmarshal(TPMS_PCR_SELECT_MARSHAL_INDEX, (target), (buffer), (size))
1662 #define TPMS_PCR_SELECT_Marshal(source, buffer, size)                           \
1663     Marshal(TPMS_PCR_SELECT_MARSHAL_INDEX, (source), (buffer), (size))
1664 #define TPMS_PCR_SELECTION_Unmarshal(target, buffer, size)                      \
1665     Unmarshal(TPMS_PCR_SELECTION_MARSHAL_INDEX, (target), (buffer), (size))
1666 #define TPMS_PCR_SELECTION_Marshal(source, buffer, size)                        \
1667     Marshal(TPMS_PCR_SELECTION_MARSHAL_INDEX, (source), (buffer), (size))
1668 #define TPMT_TK_CREATION_Unmarshal(target, buffer, size)                        \
1669     Unmarshal(TPMT_TK_CREATION_MARSHAL_INDEX, (target), (buffer), (size))
1670 #define TPMT_TK_CREATION_Marshal(source, buffer, size)                          \
```

```
1671        Marshal(TPMT_TK_CREATION_MARSHAL_INDEX, (source), (buffer), (size))
1672    #define TPMT_TK_VERIFIED_Unmarshal(target, buffer, size)                       \
1673        Unmarshal(TPMT_TK_VERIFIED_MARSHAL_INDEX, (target), (buffer), (size))
1674    #define TPMT_TK_VERIFIED_Marshal(source, buffer, size)                         \
1675        Marshal(TPMT_TK_VERIFIED_MARSHAL_INDEX, (source), (buffer), (size))
1676    #define TPMT_TK_AUTH_Unmarshal(target, buffer, size)                           \
1677        Unmarshal(TPMT_TK_AUTH_MARSHAL_INDEX, (target), (buffer), (size))
1678    #define TPMT_TK_AUTH_Marshal(source, buffer, size)                             \
1679        Marshal(TPMT_TK_AUTH_MARSHAL_INDEX, (source), (buffer), (size))
1680    #define TPMT_TK_HASHCHECK_Unmarshal(target, buffer, size)                      \
1681        Unmarshal(TPMT_TK_HASHCHECK_MARSHAL_INDEX, (target), (buffer), (size))
1682    #define TPMT_TK_HASHCHECK_Marshal(source, buffer, size)                        \
1683        Marshal(TPMT_TK_HASHCHECK_MARSHAL_INDEX, (source), (buffer), (size))
1684    #define TPMS_ALG_PROPERTY_Marshal(source, buffer, size)                        \
1685        Marshal(TPMS_ALG_PROPERTY_MARSHAL_INDEX, (source), (buffer), (size))
1686    #define TPMS_TAGGED_PROPERTY_Marshal(source, buffer, size)                     \
1687        Marshal(TPMS_TAGGED_PROPERTY_MARSHAL_INDEX, (source), (buffer), (size))
1688    #define TPMS_TAGGED_PCR_SELECT_Marshal(source, buffer, size)                   \
1689        Marshal(TPMS_TAGGED_PCR_SELECT_MARSHAL_INDEX, (source), (buffer), (size))
1690    #define TPMS_TAGGED_POLICY_Marshal(source, buffer, size)                       \
1691        Marshal(TPMS_TAGGED_POLICY_MARSHAL_INDEX, (source), (buffer), (size))
1692    #define TPML_CC_Unmarshal(target, buffer, size)                                \
1693        Unmarshal(TPML_CC_MARSHAL_INDEX, (target), (buffer), (size))
1694    #define TPML_CC_Marshal(source, buffer, size)                                  \
1695        Marshal(TPML_CC_MARSHAL_INDEX, (source), (buffer), (size))
1696    #define TPML_CCA_Marshal(source, buffer, size)                                 \
1697        Marshal(TPML_CCA_MARSHAL_INDEX, (source), (buffer), (size))
1698    #define TPML_ALG_Unmarshal(target, buffer, size)                               \
1699        Unmarshal(TPML_ALG_MARSHAL_INDEX, (target), (buffer), (size))
1700    #define TPML_ALG_Marshal(source, buffer, size)                                 \
1701        Marshal(TPML_ALG_MARSHAL_INDEX, (source), (buffer), (size))
1702    #define TPML_HANDLE_Marshal(source, buffer, size)                              \
1703        Marshal(TPML_HANDLE_MARSHAL_INDEX, (source), (buffer), (size))
1704    #define TPML_DIGEST_Unmarshal(target, buffer, size)                            \
1705        Unmarshal(TPML_DIGEST_MARSHAL_INDEX, (target), (buffer), (size))
1706    #define TPML_DIGEST_Marshal(source, buffer, size)                              \
1707        Marshal(TPML_DIGEST_MARSHAL_INDEX, (source), (buffer), (size))
1708    #define TPML_DIGEST_VALUES_Unmarshal(target, buffer, size)                     \
1709        Unmarshal(TPML_DIGEST_VALUES_MARSHAL_INDEX, (target), (buffer), (size))
1710    #define TPML_DIGEST_VALUES_Marshal(source, buffer, size)                       \
1711        Marshal(TPML_DIGEST_VALUES_MARSHAL_INDEX, (source), (buffer), (size))
1712    #define TPML_PCR_SELECTION_Unmarshal(target, buffer, size)                     \
1713        Unmarshal(TPML_PCR_SELECTION_MARSHAL_INDEX, (target), (buffer), (size))
1714    #define TPML_PCR_SELECTION_Marshal(source, buffer, size)                       \
1715        Marshal(TPML_PCR_SELECTION_MARSHAL_INDEX, (source), (buffer), (size))
1716    #define TPML_ALG_PROPERTY_Marshal(source, buffer, size)                        \
1717        Marshal(TPML_ALG_PROPERTY_MARSHAL_INDEX, (source), (buffer), (size))
1718    #define TPML_TAGGED_TPM_PROPERTY_Marshal(source, buffer, size)                 \
1719        Marshal(TPML_TAGGED_TPM_PROPERTY_MARSHAL_INDEX, (source), (buffer), (size))
1720    #define TPML_TAGGED_PCR_PROPERTY_Marshal(source, buffer, size)                 \
1721        Marshal(TPML_TAGGED_PCR_PROPERTY_MARSHAL_INDEX, (source), (buffer), (size))
1722    #define TPML_ECC_CURVE_Marshal(source, buffer, size)                           \
1723        Marshal(TPML_ECC_CURVE_MARSHAL_INDEX, (source), (buffer), (size))
1724    #define TPML_TAGGED_POLICY_Marshal(source, buffer, size)                       \
1725        Marshal(TPML_TAGGED_POLICY_MARSHAL_INDEX, (source), (buffer), (size))
1726    #define TPMU_CAPABILITIES_Marshal(source, buffer, size, selector)              \
1727        MarshalUnion(TPMU_CAPABILITIES_MARSHAL_INDEX, (target), (buffer), (size),  \
1728          (selector))
1729    #define TPMS_CAPABILITY_DATA_Marshal(source, buffer, size)                     \
1730        Marshal(TPMS_CAPABILITY_DATA_MARSHAL_INDEX, (source), (buffer), (size))
1731    #define TPMS_CLOCK_INFO_Unmarshal(target, buffer, size)                        \
1732        Unmarshal(TPMS_CLOCK_INFO_MARSHAL_INDEX, (target), (buffer), (size))
1733    #define TPMS_CLOCK_INFO_Marshal(source, buffer, size)                          \
1734        Marshal(TPMS_CLOCK_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1735    #define TPMS_TIME_INFO_Unmarshal(target, buffer, size)                         \
1736        Unmarshal(TPMS_TIME_INFO_MARSHAL_INDEX, (target), (buffer), (size))
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
1737 #define TPMS_TIME_INFO_Marshal(source, buffer, size)                          \
1738     Marshal(TPMS_TIME_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1739 #define TPMS_TIME_ATTEST_INFO_Marshal(source, buffer, size)                   \
1740     Marshal(TPMS_TIME_ATTEST_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1741 #define TPMS_CERTIFY_INFO_Marshal(source, buffer, size)                       \
1742     Marshal(TPMS_CERTIFY_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1743 #define TPMS_QUOTE_INFO_Marshal(source, buffer, size)                         \
1744     Marshal(TPMS_QUOTE_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1745 #define TPMS_COMMAND_AUDIT_INFO_Marshal(source, buffer, size)                 \
1746     Marshal(TPMS_COMMAND_AUDIT_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1747 #define TPMS_SESSION_AUDIT_INFO_Marshal(source, buffer, size)                 \
1748     Marshal(TPMS_SESSION_AUDIT_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1749 #define TPMS_CREATION_INFO_Marshal(source, buffer, size)                      \
1750     Marshal(TPMS_CREATION_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1751 #define TPMS_NV_CERTIFY_INFO_Marshal(source, buffer, size)                    \
1752     Marshal(TPMS_NV_CERTIFY_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1753 #define TPMS_NV_DIGEST_CERTIFY_INFO_Marshal(source, buffer, size)             \
1754     Marshal(TPMS_NV_DIGEST_CERTIFY_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1755 #define TPMI_ST_ATTEST_Marshal(source, buffer, size)                          \
1756     Marshal(TPMI_ST_ATTEST_MARSHAL_INDEX, (source), (buffer), (size))
1757 #define TPMU_ATTEST_Marshal(source, buffer, size, selector)                   \
1758     MarshalUnion(TPMU_ATTEST_MARSHAL_INDEX, (target), (buffer), (size), (selector))
1759 #define TPMS_ATTEST_Marshal(source, buffer, size)                             \
1760     Marshal(TPMS_ATTEST_MARSHAL_INDEX, (source), (buffer), (size))
1761 #define TPM2B_ATTEST_Marshal(source, buffer, size)                            \
1762     Marshal(TPM2B_ATTEST_MARSHAL_INDEX, (source), (buffer), (size))
1763 #define TPMS_AUTH_COMMAND_Unmarshal(target, buffer, size)                     \
1764     Unmarshal(TPMS_AUTH_COMMAND_MARSHAL_INDEX, (target), (buffer), (size))
1765 #define TPMS_AUTH_RESPONSE_Marshal(source, buffer, size)                      \
1766     Marshal(TPMS_AUTH_RESPONSE_MARSHAL_INDEX, (source), (buffer), (size))
1767 #define TPMI_TDES_KEY_BITS_Unmarshal(target, buffer, size)                    \
1768     Unmarshal(TPMI_TDES_KEY_BITS_MARSHAL_INDEX, (target), (buffer), (size))
1769 #define TPMI_TDES_KEY_BITS_Marshal(source, buffer, size)                      \
1770     Marshal(TPMI_TDES_KEY_BITS_MARSHAL_INDEX, (source), (buffer), (size))
1771 #define TPMI_AES_KEY_BITS_Unmarshal(target, buffer, size)                     \
1772     Unmarshal(TPMI_AES_KEY_BITS_MARSHAL_INDEX, (target), (buffer), (size))
1773 #define TPMI_AES_KEY_BITS_Marshal(source, buffer, size)                       \
1774     Marshal(TPMI_AES_KEY_BITS_MARSHAL_INDEX, (source), (buffer), (size))
1775 #define TPMI_SM4_KEY_BITS_Unmarshal(target, buffer, size)                     \
1776     Unmarshal(TPMI_SM4_KEY_BITS_MARSHAL_INDEX, (target), (buffer), (size))
1777 #define TPMI_SM4_KEY_BITS_Marshal(source, buffer, size)                       \
1778     Marshal(TPMI_SM4_KEY_BITS_MARSHAL_INDEX, (source), (buffer), (size))
1779 #define TPMI_CAMELLIA_KEY_BITS_Unmarshal(target, buffer, size)                \
1780     Unmarshal(TPMI_CAMELLIA_KEY_BITS_MARSHAL_INDEX, (target), (buffer), (size))
1781 #define TPMI_CAMELLIA_KEY_BITS_Marshal(source, buffer, size)                  \
1782     Marshal(TPMI_CAMELLIA_KEY_BITS_MARSHAL_INDEX, (source), (buffer), (size))
1783 #define TPMU_SYM_KEY_BITS_Unmarshal(target, buffer, size, selector)           \
1784     UnmarshalUnion(TPMU_SYM_KEY_BITS_MARSHAL_INDEX, (target), (buffer), (size), \
1785       (selector))
1786 #define TPMU_SYM_KEY_BITS_Marshal(source, buffer, size, selector)             \
1787     MarshalUnion(TPMU_SYM_KEY_BITS_MARSHAL_INDEX, (target), (buffer), (size), \
1788       (selector))
1789 #define TPMU_SYM_MODE_Unmarshal(target, buffer, size, selector)               \
1790     UnmarshalUnion(TPMU_SYM_MODE_MARSHAL_INDEX, (target), (buffer), (size),  \
1791       (selector))
1792 #define TPMU_SYM_MODE_Marshal(source, buffer, size, selector)                 \
1793     MarshalUnion(TPMU_SYM_MODE_MARSHAL_INDEX, (target), (buffer), (size),    \
1794       (selector))
1795 #define TPMT_SYM_DEF_Unmarshal(target, buffer, size, flag)                    \
1796     Unmarshal(TPMT_SYM_DEF_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),  \
1797       (buffer), (size))
1798 #define TPMT_SYM_DEF_Marshal(source, buffer, size)                            \
1799     Marshal(TPMT_SYM_DEF_MARSHAL_INDEX, (source), (buffer), (size))
1800 #define TPMT_SYM_DEF_OBJECT_Unmarshal(target, buffer, size, flag)             \
1801     Unmarshal(TPMT_SYM_DEF_OBJECT_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target), \
1802       (buffer), (size))
```

 Copyright © TCG 2006-2020

```
1803 #define TPMT_SYM_DEF_OBJECT_Marshal(source, buffer, size)                     \
1804     Marshal(TPMT_SYM_DEF_OBJECT_MARSHAL_INDEX, (source), (buffer), (size))
1805 #define TPM2B_SYM_KEY_Unmarshal(target, buffer, size)                         \
1806     Unmarshal(TPM2B_SYM_KEY_MARSHAL_INDEX, (target), (buffer), (size))
1807 #define TPM2B_SYM_KEY_Marshal(source, buffer, size)                           \
1808     Marshal(TPM2B_SYM_KEY_MARSHAL_INDEX, (source), (buffer), (size))
1809 #define TPMS_SYMCIPHER_PARMS_Unmarshal(target, buffer, size)                  \
1810     Unmarshal(TPMS_SYMCIPHER_PARMS_MARSHAL_INDEX, (target), (buffer), (size))
1811 #define TPMS_SYMCIPHER_PARMS_Marshal(source, buffer, size)                    \
1812     Marshal(TPMS_SYMCIPHER_PARMS_MARSHAL_INDEX, (source), (buffer), (size))
1813 #define TPM2B_LABEL_Unmarshal(target, buffer, size)                           \
1814     Unmarshal(TPM2B_LABEL_MARSHAL_INDEX, (target), (buffer), (size))
1815 #define TPM2B_LABEL_Marshal(source, buffer, size)                             \
1816     Marshal(TPM2B_LABEL_MARSHAL_INDEX, (source), (buffer), (size))
1817 #define TPMS_DERIVE_Unmarshal(target, buffer, size)                           \
1818     Unmarshal(TPMS_DERIVE_MARSHAL_INDEX, (target), (buffer), (size))
1819 #define TPMS_DERIVE_Marshal(source, buffer, size)                             \
1820     Marshal(TPMS_DERIVE_MARSHAL_INDEX, (source), (buffer), (size))
1821 #define TPM2B_DERIVE_Unmarshal(target, buffer, size)                          \
1822     Unmarshal(TPM2B_DERIVE_MARSHAL_INDEX, (target), (buffer), (size))
1823 #define TPM2B_DERIVE_Marshal(source, buffer, size)                            \
1824     Marshal(TPM2B_DERIVE_MARSHAL_INDEX, (source), (buffer), (size))
1825 #define TPM2B_SENSITIVE_DATA_Unmarshal(target, buffer, size)                  \
1826     Unmarshal(TPM2B_SENSITIVE_DATA_MARSHAL_INDEX, (target), (buffer), (size))
1827 #define TPM2B_SENSITIVE_DATA_Marshal(source, buffer, size)                    \
1828     Marshal(TPM2B_SENSITIVE_DATA_MARSHAL_INDEX, (source), (buffer), (size))
1829 #define TPMS_SENSITIVE_CREATE_Unmarshal(target, buffer, size)                 \
1830     Unmarshal(TPMS_SENSITIVE_CREATE_MARSHAL_INDEX, (target), (buffer), (size))
1831 #define TPM2B_SENSITIVE_CREATE_Unmarshal(target, buffer, size)                \
1832     Unmarshal(TPM2B_SENSITIVE_CREATE_MARSHAL_INDEX, (target), (buffer), (size))
1833 #define TPMS_SCHEME_HASH_Unmarshal(target, buffer, size)                      \
1834     Unmarshal(TPMS_SCHEME_HASH_MARSHAL_INDEX, (target), (buffer), (size))
1835 #define TPMS_SCHEME_HASH_Marshal(source, buffer, size)                        \
1836     Marshal(TPMS_SCHEME_HASH_MARSHAL_INDEX, (source), (buffer), (size))
1837 #define TPMS_SCHEME_ECDAA_Unmarshal(target, buffer, size)                     \
1838     Unmarshal(TPMS_SCHEME_ECDAA_MARSHAL_INDEX, (target), (buffer), (size))
1839 #define TPMS_SCHEME_ECDAA_Marshal(source, buffer, size)                       \
1840     Marshal(TPMS_SCHEME_ECDAA_MARSHAL_INDEX, (source), (buffer), (size))
1841 #define TPMI_ALG_KEYEDHASH_SCHEME_Unmarshal(target, buffer, size, flag)       \
1842     Unmarshal(TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), \
1843       (target), (buffer), (size))
1844 #define TPMI_ALG_KEYEDHASH_SCHEME_Marshal(source, buffer, size)               \
1845     Marshal(TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
1846 #define TPMS_SCHEME_HMAC_Unmarshal(target, buffer, size)                      \
1847     Unmarshal(TPMS_SCHEME_HMAC_MARSHAL_INDEX, (target), (buffer), (size))
1848 #define TPMS_SCHEME_HMAC_Marshal(source, buffer, size)                        \
1849     Marshal(TPMS_SCHEME_HMAC_MARSHAL_INDEX, (source), (buffer), (size))
1850 #define TPMS_SCHEME_XOR_Unmarshal(target, buffer, size)                       \
1851     Unmarshal(TPMS_SCHEME_XOR_MARSHAL_INDEX, (target), (buffer), (size))
1852 #define TPMS_SCHEME_XOR_Marshal(source, buffer, size)                         \
1853     Marshal(TPMS_SCHEME_XOR_MARSHAL_INDEX, (source), (buffer), (size))
1854 #define TPMU_SCHEME_KEYEDHASH_Unmarshal(target, buffer, size, selector)       \
1855     UnmarshalUnion(TPMU_SCHEME_KEYEDHASH_MARSHAL_INDEX, (target), (buffer), (size), \
1856       (selector))
1857 #define TPMU_SCHEME_KEYEDHASH_Marshal(source, buffer, size, selector)         \
1858     MarshalUnion(TPMU_SCHEME_KEYEDHASH_MARSHAL_INDEX, (target), (buffer), (size), \
1859       (selector))
1860 #define TPMT_KEYEDHASH_SCHEME_Unmarshal(target, buffer, size, flag)           \
1861     Unmarshal(TPMT_KEYEDHASH_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0),   \
1862       (target), (buffer), (size))
1863 #define TPMT_KEYEDHASH_SCHEME_Marshal(source, buffer, size)                   \
1864     Marshal(TPMT_KEYEDHASH_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
1865 #define TPMS_SIG_SCHEME_RSASSA_Unmarshal(target, buffer, size)                \
1866     Unmarshal(TPMS_SIG_SCHEME_RSASSA_MARSHAL_INDEX, (target), (buffer), (size))
1867 #define TPMS_SIG_SCHEME_RSASSA_Marshal(source, buffer, size)                  \
1868     Marshal(TPMS_SIG_SCHEME_RSASSA_MARSHAL_INDEX, (source), (buffer), (size))
```

```
1869    #define TPMS_SIG_SCHEME_RSAPSS_Unmarshal(target, buffer, size)                \
1870        Unmarshal(TPMS_SIG_SCHEME_RSAPSS_MARSHAL_INDEX, (target), (buffer), (size))
1871    #define TPMS_SIG_SCHEME_RSAPSS_Marshal(source, buffer, size)                  \
1872        Marshal(TPMS_SIG_SCHEME_RSAPSS_MARSHAL_INDEX, (source), (buffer), (size))
1873    #define TPMS_SIG_SCHEME_ECDSA_Unmarshal(target, buffer, size)                 \
1874        Unmarshal(TPMS_SIG_SCHEME_ECDSA_MARSHAL_INDEX, (target), (buffer), (size))
1875    #define TPMS_SIG_SCHEME_ECDSA_Marshal(source, buffer, size)                   \
1876        Marshal(TPMS_SIG_SCHEME_ECDSA_MARSHAL_INDEX, (source), (buffer), (size))
1877    #define TPMS_SIG_SCHEME_SM2_Unmarshal(target, buffer, size)                   \
1878        Unmarshal(TPMS_SIG_SCHEME_SM2_MARSHAL_INDEX, (target), (buffer), (size))
1879    #define TPMS_SIG_SCHEME_SM2_Marshal(source, buffer, size)                     \
1880        Marshal(TPMS_SIG_SCHEME_SM2_MARSHAL_INDEX, (source), (buffer), (size))
1881    #define TPMS_SIG_SCHEME_ECSCHNORR_Unmarshal(target, buffer, size)             \
1882        Unmarshal(TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_INDEX, (target), (buffer), (size))
1883    #define TPMS_SIG_SCHEME_ECSCHNORR_Marshal(source, buffer, size)               \
1884        Marshal(TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_INDEX, (source), (buffer), (size))
1885    #define TPMS_SIG_SCHEME_ECDAA_Unmarshal(target, buffer, size)                 \
1886        Unmarshal(TPMS_SIG_SCHEME_ECDAA_MARSHAL_INDEX, (target), (buffer), (size))
1887    #define TPMS_SIG_SCHEME_ECDAA_Marshal(source, buffer, size)                   \
1888        Marshal(TPMS_SIG_SCHEME_ECDAA_MARSHAL_INDEX, (source), (buffer), (size))
1889    #define TPMU_SIG_SCHEME_Unmarshal(target, buffer, size, selector)             \
1890        UnmarshalUnion(TPMU_SIG_SCHEME_MARSHAL_INDEX, (target), (buffer), (size), \
1891          (selector))
1892    #define TPMU_SIG_SCHEME_Marshal(source, buffer, size, selector)               \
1893        MarshalUnion(TPMU_SIG_SCHEME_MARSHAL_INDEX, (target), (buffer), (size),   \
1894          (selector))
1895    #define TPMT_SIG_SCHEME_Unmarshal(target, buffer, size, flag)                 \
1896        Unmarshal(TPMT_SIG_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target), \
1897          (buffer), (size))
1898    #define TPMT_SIG_SCHEME_Marshal(source, buffer, size)                         \
1899        Marshal(TPMT_SIG_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
1900    #define TPMS_ENC_SCHEME_OAEP_Unmarshal(target, buffer, size)                  \
1901        Unmarshal(TPMS_ENC_SCHEME_OAEP_MARSHAL_INDEX, (target), (buffer), (size))
1902    #define TPMS_ENC_SCHEME_OAEP_Marshal(source, buffer, size)                    \
1903        Marshal(TPMS_ENC_SCHEME_OAEP_MARSHAL_INDEX, (source), (buffer), (size))
1904    #define TPMS_ENC_SCHEME_RSAES_Unmarshal(target, buffer, size)                 \
1905        Unmarshal(TPMS_ENC_SCHEME_RSAES_MARSHAL_INDEX, (target), (buffer), (size))
1906    #define TPMS_ENC_SCHEME_RSAES_Marshal(source, buffer, size)                   \
1907        Marshal(TPMS_ENC_SCHEME_RSAES_MARSHAL_INDEX, (source), (buffer), (size))
1908    #define TPMS_KEY_SCHEME_ECDH_Unmarshal(target, buffer, size)                  \
1909        Unmarshal(TPMS_KEY_SCHEME_ECDH_MARSHAL_INDEX, (target), (buffer), (size))
1910    #define TPMS_KEY_SCHEME_ECDH_Marshal(source, buffer, size)                    \
1911        Marshal(TPMS_KEY_SCHEME_ECDH_MARSHAL_INDEX, (source), (buffer), (size))
1912    #define TPMS_KEY_SCHEME_ECMQV_Unmarshal(target, buffer, size)                 \
1913        Unmarshal(TPMS_KEY_SCHEME_ECMQV_MARSHAL_INDEX, (target), (buffer), (size))
1914    #define TPMS_KEY_SCHEME_ECMQV_Marshal(source, buffer, size)                   \
1915        Marshal(TPMS_KEY_SCHEME_ECMQV_MARSHAL_INDEX, (source), (buffer), (size))
1916    #define TPMS_SCHEME_MGF1_Unmarshal(target, buffer, size)                      \
1917        Unmarshal(TPMS_SCHEME_MGF1_MARSHAL_INDEX, (target), (buffer), (size))
1918    #define TPMS_SCHEME_MGF1_Marshal(source, buffer, size)                        \
1919        Marshal(TPMS_SCHEME_MGF1_MARSHAL_INDEX, (source), (buffer), (size))
1920    #define TPMS_SCHEME_KDF1_SP800_56A_Unmarshal(target, buffer, size)            \
1921        Unmarshal(TPMS_SCHEME_KDF1_SP800_56A_MARSHAL_INDEX, (target), (buffer), (size))
1922    #define TPMS_SCHEME_KDF1_SP800_56A_Marshal(source, buffer, size)              \
1923        Marshal(TPMS_SCHEME_KDF1_SP800_56A_MARSHAL_INDEX, (source), (buffer), (size))
1924    #define TPMS_SCHEME_KDF2_Unmarshal(target, buffer, size)                      \
1925        Unmarshal(TPMS_SCHEME_KDF2_MARSHAL_INDEX, (target), (buffer), (size))
1926    #define TPMS_SCHEME_KDF2_Marshal(source, buffer, size)                        \
1927        Marshal(TPMS_SCHEME_KDF2_MARSHAL_INDEX, (source), (buffer), (size))
1928    #define TPMS_SCHEME_KDF1_SP800_108_Unmarshal(target, buffer, size)            \
1929        Unmarshal(TPMS_SCHEME_KDF1_SP800_108_MARSHAL_INDEX, (target), (buffer), (size))
1930    #define TPMS_SCHEME_KDF1_SP800_108_Marshal(source, buffer, size)              \
1931        Marshal(TPMS_SCHEME_KDF1_SP800_108_MARSHAL_INDEX, (source), (buffer), (size))
1932    #define TPMU_KDF_SCHEME_Unmarshal(target, buffer, size, selector)             \
1933        UnmarshalUnion(TPMU_KDF_SCHEME_MARSHAL_INDEX, (target), (buffer), (size), \
1934          (selector))
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1935 #define TPMU_KDF_SCHEME_Marshal(source, buffer, size, selector)              \
1936     MarshalUnion(TPMU_KDF_SCHEME_MARSHAL_INDEX, (target), (buffer), (size),  \
1937       (selector))
1938 #define TPMT_KDF_SCHEME_Unmarshal(target, buffer, size, flag)                \
1939     Unmarshal(TPMT_KDF_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),    \
1940       (buffer), (size))
1941 #define TPMT_KDF_SCHEME_Marshal(source, buffer, size)                        \
1942     Marshal(TPMT_KDF_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
1943 #define TPMI_ALG_ASYM_SCHEME_Unmarshal(target, buffer, size, flag)           \
1944     Unmarshal(TPMI_ALG_ASYM_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),\
1945       (buffer), (size))
1946 #define TPMI_ALG_ASYM_SCHEME_Marshal(source, buffer, size)                   \
1947     Marshal(TPMI_ALG_ASYM_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
1948 #define TPMU_ASYM_SCHEME_Unmarshal(target, buffer, size, selector)           \
1949     UnmarshalUnion(TPMU_ASYM_SCHEME_MARSHAL_INDEX, (target), (buffer), (size),     \
1950       (selector))
1951 #define TPMU_ASYM_SCHEME_Marshal(source, buffer, size, selector)             \
1952     MarshalUnion(TPMU_ASYM_SCHEME_MARSHAL_INDEX, (target), (buffer), (size),       \
1953       (selector))
1954 #define TPMI_ALG_RSA_SCHEME_Unmarshal(target, buffer, size, flag)            \
1955     Unmarshal(TPMI_ALG_RSA_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target), \
1956       (buffer), (size))
1957 #define TPMI_ALG_RSA_SCHEME_Marshal(source, buffer, size)                    \
1958     Marshal(TPMI_ALG_RSA_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
1959 #define TPMT_RSA_SCHEME_Unmarshal(target, buffer, size, flag)                \
1960     Unmarshal(TPMT_RSA_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),    \
1961       (buffer), (size))
1962 #define TPMT_RSA_SCHEME_Marshal(source, buffer, size)                        \
1963     Marshal(TPMT_RSA_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
1964 #define TPMI_ALG_RSA_DECRYPT_Unmarshal(target, buffer, size, flag)           \
1965     Unmarshal(TPMI_ALG_RSA_DECRYPT_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),\
1966       (buffer), (size))
1967 #define TPMI_ALG_RSA_DECRYPT_Marshal(source, buffer, size)                   \
1968     Marshal(TPMI_ALG_RSA_DECRYPT_MARSHAL_INDEX, (source), (buffer), (size))
1969 #define TPMT_RSA_DECRYPT_Unmarshal(target, buffer, size, flag)               \
1970     Unmarshal(TPMT_RSA_DECRYPT_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),   \
1971       (buffer), (size))
1972 #define TPMT_RSA_DECRYPT_Marshal(source, buffer, size)                       \
1973     Marshal(TPMT_RSA_DECRYPT_MARSHAL_INDEX, (source), (buffer), (size))
1974 #define TPM2B_PUBLIC_KEY_RSA_Unmarshal(target, buffer, size)                 \
1975     Unmarshal(TPM2B_PUBLIC_KEY_RSA_MARSHAL_INDEX, (target), (buffer), (size))
1976 #define TPM2B_PUBLIC_KEY_RSA_Marshal(source, buffer, size)                   \
1977     Marshal(TPM2B_PUBLIC_KEY_RSA_MARSHAL_INDEX, (source), (buffer), (size))
1978 #define TPMI_RSA_KEY_BITS_Unmarshal(target, buffer, size)                    \
1979     Unmarshal(TPMI_RSA_KEY_BITS_MARSHAL_INDEX, (target), (buffer), (size))
1980 #define TPMI_RSA_KEY_BITS_Marshal(source, buffer, size)                      \
1981     Marshal(TPMI_RSA_KEY_BITS_MARSHAL_INDEX, (source), (buffer), (size))
1982 #define TPM2B_PRIVATE_KEY_RSA_Unmarshal(target, buffer, size)                \
1983     Unmarshal(TPM2B_PRIVATE_KEY_RSA_MARSHAL_INDEX, (target), (buffer), (size))
1984 #define TPM2B_PRIVATE_KEY_RSA_Marshal(source, buffer, size)                  \
1985     Marshal(TPM2B_PRIVATE_KEY_RSA_MARSHAL_INDEX, (source), (buffer), (size))
1986 #define TPM2B_ECC_PARAMETER_Unmarshal(target, buffer, size)                  \
1987     Unmarshal(TPM2B_ECC_PARAMETER_MARSHAL_INDEX, (target), (buffer), (size))
1988 #define TPM2B_ECC_PARAMETER_Marshal(source, buffer, size)                    \
1989     Marshal(TPM2B_ECC_PARAMETER_MARSHAL_INDEX, (source), (buffer), (size))
1990 #define TPMS_ECC_POINT_Unmarshal(target, buffer, size)                       \
1991     Unmarshal(TPMS_ECC_POINT_MARSHAL_INDEX, (target), (buffer), (size))
1992 #define TPMS_ECC_POINT_Marshal(source, buffer, size)                         \
1993     Marshal(TPMS_ECC_POINT_MARSHAL_INDEX, (source), (buffer), (size))
1994 #define TPM2B_ECC_POINT_Unmarshal(target, buffer, size)                      \
1995     Unmarshal(TPM2B_ECC_POINT_MARSHAL_INDEX, (target), (buffer), (size))
1996 #define TPM2B_ECC_POINT_Marshal(source, buffer, size)                        \
1997     Marshal(TPM2B_ECC_POINT_MARSHAL_INDEX, (source), (buffer), (size))
1998 #define TPMI_ALG_ECC_SCHEME_Unmarshal(target, buffer, size, flag)            \
1999     Unmarshal(TPMI_ALG_ECC_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target), \
2000       (buffer), (size))
```

```
2001    #define TPMI_ALG_ECC_SCHEME_Marshal(source, buffer, size)                        \
2002        Marshal(TPMI_ALG_ECC_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
2003    #define TPMI_ECC_CURVE_Unmarshal(target, buffer, size)                           \
2004        Unmarshal(TPMI_ECC_CURVE_MARSHAL_INDEX, (target), (buffer), (size))
2005    #define TPMI_ECC_CURVE_Marshal(source, buffer, size)                             \
2006        Marshal(TPMI_ECC_CURVE_MARSHAL_INDEX, (source), (buffer), (size))
2007    #define TPMT_ECC_SCHEME_Unmarshal(target, buffer, size, flag)                    \
2008        Unmarshal(TPMT_ECC_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),  \
2009          (buffer), (size))
2010    #define TPMT_ECC_SCHEME_Marshal(source, buffer, size)                            \
2011        Marshal(TPMT_ECC_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
2012    #define TPMS_ALGORITHM_DETAIL_ECC_Marshal(source, buffer, size)                  \
2013        Marshal(TPMS_ALGORITHM_DETAIL_ECC_MARSHAL_INDEX, (source), (buffer), (size))
2014    #define TPMS_SIGNATURE_RSA_Unmarshal(target, buffer, size)                       \
2015        Unmarshal(TPMS_SIGNATURE_RSA_MARSHAL_INDEX, (target), (buffer), (size))
2016    #define TPMS_SIGNATURE_RSA_Marshal(source, buffer, size)                         \
2017        Marshal(TPMS_SIGNATURE_RSA_MARSHAL_INDEX, (source), (buffer), (size))
2018    #define TPMS_SIGNATURE_RSASSA_Unmarshal(target, buffer, size)                    \
2019        Unmarshal(TPMS_SIGNATURE_RSASSA_MARSHAL_INDEX, (target), (buffer), (size))
2020    #define TPMS_SIGNATURE_RSASSA_Marshal(source, buffer, size)                      \
2021        Marshal(TPMS_SIGNATURE_RSASSA_MARSHAL_INDEX, (source), (buffer), (size))
2022    #define TPMS_SIGNATURE_RSAPSS_Unmarshal(target, buffer, size)                    \
2023        Unmarshal(TPMS_SIGNATURE_RSAPSS_MARSHAL_INDEX, (target), (buffer), (size))
2024    #define TPMS_SIGNATURE_RSAPSS_Marshal(source, buffer, size)                      \
2025        Marshal(TPMS_SIGNATURE_RSAPSS_MARSHAL_INDEX, (source), (buffer), (size))
2026    #define TPMS_SIGNATURE_ECC_Unmarshal(target, buffer, size)                       \
2027        Unmarshal(TPMS_SIGNATURE_ECC_MARSHAL_INDEX, (target), (buffer), (size))
2028    #define TPMS_SIGNATURE_ECC_Marshal(source, buffer, size)                         \
2029        Marshal(TPMS_SIGNATURE_ECC_MARSHAL_INDEX, (source), (buffer), (size))
2030    #define TPMS_SIGNATURE_ECDAA_Unmarshal(target, buffer, size)                     \
2031        Unmarshal(TPMS_SIGNATURE_ECDAA_MARSHAL_INDEX, (target), (buffer), (size))
2032    #define TPMS_SIGNATURE_ECDAA_Marshal(source, buffer, size)                       \
2033        Marshal(TPMS_SIGNATURE_ECDAA_MARSHAL_INDEX, (source), (buffer), (size))
2034    #define TPMS_SIGNATURE_ECDSA_Unmarshal(target, buffer, size)                     \
2035        Unmarshal(TPMS_SIGNATURE_ECDSA_MARSHAL_INDEX, (target), (buffer), (size))
2036    #define TPMS_SIGNATURE_ECDSA_Marshal(source, buffer, size)                       \
2037        Marshal(TPMS_SIGNATURE_ECDSA_MARSHAL_INDEX, (source), (buffer), (size))
2038    #define TPMS_SIGNATURE_SM2_Unmarshal(target, buffer, size)                       \
2039        Unmarshal(TPMS_SIGNATURE_SM2_MARSHAL_INDEX, (target), (buffer), (size))
2040    #define TPMS_SIGNATURE_SM2_Marshal(source, buffer, size)                         \
2041        Marshal(TPMS_SIGNATURE_SM2_MARSHAL_INDEX, (source), (buffer), (size))
2042    #define TPMS_SIGNATURE_ECSCHNORR_Unmarshal(target, buffer, size)                 \
2043        Unmarshal(TPMS_SIGNATURE_ECSCHNORR_MARSHAL_INDEX, (target), (buffer), (size))
2044    #define TPMS_SIGNATURE_ECSCHNORR_Marshal(source, buffer, size)                   \
2045        Marshal(TPMS_SIGNATURE_ECSCHNORR_MARSHAL_INDEX, (source), (buffer), (size))
2046    #define TPMU_SIGNATURE_Unmarshal(target, buffer, size, selector)                 \
2047        UnmarshalUnion(TPMU_SIGNATURE_MARSHAL_INDEX, (target), (buffer), (size),     \
2048          (selector))
2049    #define TPMU_SIGNATURE_Marshal(source, buffer, size, selector)                   \
2050        MarshalUnion(TPMU_SIGNATURE_MARSHAL_INDEX, (target), (buffer), (size),       \
2051          (selector))
2052    #define TPMT_SIGNATURE_Unmarshal(target, buffer, size, flag)                     \
2053        Unmarshal(TPMT_SIGNATURE_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),   \
2054          (buffer), (size))
2055    #define TPMT_SIGNATURE_Marshal(source, buffer, size)                             \
2056        Marshal(TPMT_SIGNATURE_MARSHAL_INDEX, (source), (buffer), (size))
2057    #define TPMU_ENCRYPTED_SECRET_Unmarshal(target, buffer, size, selector)          \
2058        UnmarshalUnion(TPMU_ENCRYPTED_SECRET_MARSHAL_INDEX, (target), (buffer), (size), \
2059          (selector))
2060    #define TPMU_ENCRYPTED_SECRET_Marshal(source, buffer, size, selector)            \
2061        MarshalUnion(TPMU_ENCRYPTED_SECRET_MARSHAL_INDEX, (target), (buffer), (size), \
2062          (selector))
2063    #define TPM2B_ENCRYPTED_SECRET_Unmarshal(target, buffer, size)                   \
2064        Unmarshal(TPM2B_ENCRYPTED_SECRET_MARSHAL_INDEX, (target), (buffer), (size))
2065    #define TPM2B_ENCRYPTED_SECRET_Marshal(source, buffer, size)                     \
2066        Marshal(TPM2B_ENCRYPTED_SECRET_MARSHAL_INDEX, (source), (buffer), (size))
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
2067 #define TPMI_ALG_PUBLIC_Unmarshal(target, buffer, size)                        \
2068     Unmarshal(TPMI_ALG_PUBLIC_MARSHAL_INDEX, (target), (buffer), (size))
2069 #define TPMI_ALG_PUBLIC_Marshal(source, buffer, size)                          \
2070     Marshal(TPMI_ALG_PUBLIC_MARSHAL_INDEX, (source), (buffer), (size))
2071 #define TPMU_PUBLIC_ID_Unmarshal(target, buffer, size, selector)               \
2072     UnmarshalUnion(TPMU_PUBLIC_ID_MARSHAL_INDEX, (target), (buffer), (size),   \
2073       (selector))
2074 #define TPMU_PUBLIC_ID_Marshal(source, buffer, size, selector)                 \
2075     MarshalUnion(TPMU_PUBLIC_ID_MARSHAL_INDEX, (target), (buffer), (size),     \
2076       (selector))
2077 #define TPMS_KEYEDHASH_PARMS_Unmarshal(target, buffer, size)                   \
2078     Unmarshal(TPMS_KEYEDHASH_PARMS_MARSHAL_INDEX, (target), (buffer), (size))
2079 #define TPMS_KEYEDHASH_PARMS_Marshal(source, buffer, size)                     \
2080     Marshal(TPMS_KEYEDHASH_PARMS_MARSHAL_INDEX, (source), (buffer), (size))
2081 #define TPMS_RSA_PARMS_Unmarshal(target, buffer, size)                         \
2082     Unmarshal(TPMS_RSA_PARMS_MARSHAL_INDEX, (target), (buffer), (size))
2083 #define TPMS_RSA_PARMS_Marshal(source, buffer, size)                           \
2084     Marshal(TPMS_RSA_PARMS_MARSHAL_INDEX, (source), (buffer), (size))
2085 #define TPMS_ECC_PARMS_Unmarshal(target, buffer, size)                         \
2086     Unmarshal(TPMS_ECC_PARMS_MARSHAL_INDEX, (target), (buffer), (size))
2087 #define TPMS_ECC_PARMS_Marshal(source, buffer, size)                           \
2088     Marshal(TPMS_ECC_PARMS_MARSHAL_INDEX, (source), (buffer), (size))
2089 #define TPMU_PUBLIC_PARMS_Unmarshal(target, buffer, size, selector)            \
2090     UnmarshalUnion(TPMU_PUBLIC_PARMS_MARSHAL_INDEX, (target), (buffer), (size),\
2091       (selector))
2092 #define TPMU_PUBLIC_PARMS_Marshal(source, buffer, size, selector)              \
2093     MarshalUnion(TPMU_PUBLIC_PARMS_MARSHAL_INDEX, (target), (buffer), (size),  \
2094       (selector))
2095 #define TPMT_PUBLIC_PARMS_Unmarshal(target, buffer, size)                      \
2096     Unmarshal(TPMT_PUBLIC_PARMS_MARSHAL_INDEX, (target), (buffer), (size))
2097 #define TPMT_PUBLIC_PARMS_Marshal(source, buffer, size)                        \
2098     Marshal(TPMT_PUBLIC_PARMS_MARSHAL_INDEX, (source), (buffer), (size))
2099 #define TPMT_PUBLIC_Unmarshal(target, buffer, size, flag)                      \
2100     Unmarshal(TPMT_PUBLIC_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),    \
2101       (buffer), (size))
2102 #define TPMT_PUBLIC_Marshal(source, buffer, size)                              \
2103     Marshal(TPMT_PUBLIC_MARSHAL_INDEX, (source), (buffer), (size))
2104 #define TPM2B_PUBLIC_Unmarshal(target, buffer, size, flag)                     \
2105     Unmarshal(TPM2B_PUBLIC_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),   \
2106       (buffer), (size))
2107 #define TPM2B_PUBLIC_Marshal(source, buffer, size)                             \
2108     Marshal(TPM2B_PUBLIC_MARSHAL_INDEX, (source), (buffer), (size))
2109 #define TPM2B_TEMPLATE_Unmarshal(target, buffer, size)                         \
2110     Unmarshal(TPM2B_TEMPLATE_MARSHAL_INDEX, (target), (buffer), (size))
2111 #define TPM2B_TEMPLATE_Marshal(source, buffer, size)                           \
2112     Marshal(TPM2B_TEMPLATE_MARSHAL_INDEX, (source), (buffer), (size))
2113 #define TPM2B_PRIVATE_VENDOR_SPECIFIC_Unmarshal(target, buffer, size)          \
2114     Unmarshal(TPM2B_PRIVATE_VENDOR_SPECIFIC_MARSHAL_INDEX, (target), (buffer), \
2115       (size))
2116 #define TPM2B_PRIVATE_VENDOR_SPECIFIC_Marshal(source, buffer, size)            \
2117     Marshal(TPM2B_PRIVATE_VENDOR_SPECIFIC_MARSHAL_INDEX, (source), (buffer), (size))
2118 #define TPMU_SENSITIVE_COMPOSITE_Unmarshal(target, buffer, size, selector)     \
2119     UnmarshalUnion(TPMU_SENSITIVE_COMPOSITE_MARSHAL_INDEX, (target), (buffer), \
2120       (size), (selector))
2121 #define TPMU_SENSITIVE_COMPOSITE_Marshal(source, buffer, size, selector)       \
2122     MarshalUnion(TPMU_SENSITIVE_COMPOSITE_MARSHAL_INDEX, (target), (buffer), (size),\
2123       (selector))
2124 #define TPMT_SENSITIVE_Unmarshal(target, buffer, size)                         \
2125     Unmarshal(TPMT_SENSITIVE_MARSHAL_INDEX, (target), (buffer), (size))
2126 #define TPMT_SENSITIVE_Marshal(source, buffer, size)                           \
2127     Marshal(TPMT_SENSITIVE_MARSHAL_INDEX, (source), (buffer), (size))
2128 #define TPM2B_SENSITIVE_Unmarshal(target, buffer, size)                        \
2129     Unmarshal(TPM2B_SENSITIVE_MARSHAL_INDEX, (target), (buffer), (size))
2130 #define TPM2B_SENSITIVE_Marshal(source, buffer, size)                          \
2131     Marshal(TPM2B_SENSITIVE_MARSHAL_INDEX, (source), (buffer), (size))
2132 #define TPM2B_PRIVATE_Unmarshal(target, buffer, size)                          \
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
2133        Unmarshal(TPM2B_PRIVATE_MARSHAL_INDEX, (target), (buffer), (size))
2134    #define TPM2B_PRIVATE_Marshal(source, buffer, size)                             \
2135        Marshal(TPM2B_PRIVATE_MARSHAL_INDEX, (source), (buffer), (size))
2136    #define TPM2B_ID_OBJECT_Unmarshal(target, buffer, size)                         \
2137        Unmarshal(TPM2B_ID_OBJECT_MARSHAL_INDEX, (target), (buffer), (size))
2138    #define TPM2B_ID_OBJECT_Marshal(source, buffer, size)                           \
2139        Marshal(TPM2B_ID_OBJECT_MARSHAL_INDEX, (source), (buffer), (size))
2140    #define TPM_NV_INDEX_Marshal(source, buffer, size)                              \
2141        Marshal(TPM_NV_INDEX_MARSHAL_INDEX, (source), (buffer), (size))
2142    #define TPMS_NV_PIN_COUNTER_PARAMETERS_Unmarshal(target, buffer, size)          \
2143        Unmarshal(TPMS_NV_PIN_COUNTER_PARAMETERS_MARSHAL_INDEX, (target), (buffer),  \
2144          (size))
2145    #define TPMS_NV_PIN_COUNTER_PARAMETERS_Marshal(source, buffer, size)            \
2146        Marshal(TPMS_NV_PIN_COUNTER_PARAMETERS_MARSHAL_INDEX, (source), (buffer),    \
2147          (size))
2148    #define TPMA_NV_Unmarshal(target, buffer, size)                                 \
2149        Unmarshal(TPMA_NV_MARSHAL_INDEX, (target), (buffer), (size))
2150    #define TPMA_NV_Marshal(source, buffer, size)                                   \
2151        Marshal(TPMA_NV_MARSHAL_INDEX, (source), (buffer), (size))
2152    #define TPMS_NV_PUBLIC_Unmarshal(target, buffer, size)                          \
2153        Unmarshal(TPMS_NV_PUBLIC_MARSHAL_INDEX, (target), (buffer), (size))
2154    #define TPMS_NV_PUBLIC_Marshal(source, buffer, size)                            \
2155        Marshal(TPMS_NV_PUBLIC_MARSHAL_INDEX, (source), (buffer), (size))
2156    #define TPM2B_NV_PUBLIC_Unmarshal(target, buffer, size)                         \
2157        Unmarshal(TPM2B_NV_PUBLIC_MARSHAL_INDEX, (target), (buffer), (size))
2158    #define TPM2B_NV_PUBLIC_Marshal(source, buffer, size)                           \
2159        Marshal(TPM2B_NV_PUBLIC_MARSHAL_INDEX, (source), (buffer), (size))
2160    #define TPM2B_CONTEXT_SENSITIVE_Unmarshal(target, buffer, size)                 \
2161        Unmarshal(TPM2B_CONTEXT_SENSITIVE_MARSHAL_INDEX, (target), (buffer), (size))
2162    #define TPM2B_CONTEXT_SENSITIVE_Marshal(source, buffer, size)                   \
2163        Marshal(TPM2B_CONTEXT_SENSITIVE_MARSHAL_INDEX, (source), (buffer), (size))
2164    #define TPMS_CONTEXT_DATA_Unmarshal(target, buffer, size)                       \
2165        Unmarshal(TPMS_CONTEXT_DATA_MARSHAL_INDEX, (target), (buffer), (size))
2166    #define TPMS_CONTEXT_DATA_Marshal(source, buffer, size)                         \
2167        Marshal(TPMS_CONTEXT_DATA_MARSHAL_INDEX, (source), (buffer), (size))
2168    #define TPM2B_CONTEXT_DATA_Unmarshal(target, buffer, size)                      \
2169        Unmarshal(TPM2B_CONTEXT_DATA_MARSHAL_INDEX, (target), (buffer), (size))
2170    #define TPM2B_CONTEXT_DATA_Marshal(source, buffer, size)                        \
2171        Marshal(TPM2B_CONTEXT_DATA_MARSHAL_INDEX, (source), (buffer), (size))
2172    #define TPMS_CONTEXT_Unmarshal(target, buffer, size)                            \
2173        Unmarshal(TPMS_CONTEXT_MARSHAL_INDEX, (target), (buffer), (size))
2174    #define TPMS_CONTEXT_Marshal(source, buffer, size)                              \
2175        Marshal(TPMS_CONTEXT_MARSHAL_INDEX, (source), (buffer), (size))
2176    #define TPMS_CREATION_DATA_Marshal(source, buffer, size)                        \
2177        Marshal(TPMS_CREATION_DATA_MARSHAL_INDEX, (source), (buffer), (size))
2178    #define TPM2B_CREATION_DATA_Marshal(source, buffer, size)                       \
2179        Marshal(TPM2B_CREATION_DATA_MARSHAL_INDEX, (source), (buffer), (size))
2180    #define TPM_AT_Unmarshal(target, buffer, size)                                  \
2181        Unmarshal(TPM_AT_MARSHAL_INDEX, (target), (buffer), (size))
2182    #define TPM_AT_Marshal(source, buffer, size)                                    \
2183        Marshal(TPM_AT_MARSHAL_INDEX, (source), (buffer), (size))
2184    #define TPM_AE_Marshal(source, buffer, size)                                    \
2185        Marshal(TPM_AE_MARSHAL_INDEX, (source), (buffer), (size))
2186    #define TPMS_AC_OUTPUT_Marshal(source, buffer, size)                            \
2187        Marshal(TPMS_AC_OUTPUT_MARSHAL_INDEX, (source), (buffer), (size))
2188    #define TPML_AC_CAPABILITIES_Marshal(source, buffer, size)                      \
2189        Marshal(TPML_AC_CAPABILITIES_MARSHAL_INDEX, (source), (buffer), (size))
2190    #endif // _Table_Marshal_Data_
```

9.10.7.3 TableMarshalDefines.h

```
1    #ifndef _TABLE_MARSHAL_DEFINES_H_
2    #define _TABLE_MARSHAL_DEFINES_H_
3    #define NULL_SHIFT  15
4    #define NULL_FLAG   (1 << NULL_SHIFT)
```

The range macro processes a min, max value and produces a values that is used in the computation to see if something is within a range. The max value is (max-min). This lets the check for something (*val*) within a range become: if((val - min) <= max) // passes if in range if((val - min) > max) // passes if not in range This works because all values are converted to UINT32 values before the compare. For (val - min), all values greater than or equal to val will become positive values with a value equal to *min* being zero. This means that in an unsigned compare against 'max,' any value that is outside the range will appear to be a number greater than max. The benefit of this operation is that this will work even if the input value is a signed number as long as the input is sign extended.

```
5    #define RANGE(_min_, _max_, _base_)                                           \
6        (UINT32)_min_, (UINT32)((_base_)(_max_ - _min_))
```

This macro is like the offsetof macro but, instead of computing the offset of a structure element, it computes the stride between elements that are in a structure array. This is used instead of sizeof() because the sizeof() operator on a structure can return an implementation dependent value.

```
7    #define STRIDE(s)    ((UINT16)(size_t)&(((s *)0)[1]))
8    #define MARSHAL_REF(TYPE)    ((UINT16)(offsetof(MARSHAL_DATA, TYPE)))
```

This macro creates the entry in the array lookup table

```
9    #define ARRAY_MARSHAL_ENTRY(TYPE)                                            \
10           {(marshalIndex_t)TYPE##_MARSHAL_REF, (UINT16)STRIDE(TYPE)}
```

Defines for array lookup

```
11   #define UINT8_ARRAY_MARSHAL_INDEX                        0   // 0x00
12   #define TPM_CC_ARRAY_MARSHAL_INDEX                       1   // 0x01
13   #define TPMA_CC_ARRAY_MARSHAL_INDEX                      2   // 0x02
14   #define TPM_ALG_ID_ARRAY_MARSHAL_INDEX                   3   // 0x03
15   #define TPM_HANDLE_ARRAY_MARSHAL_INDEX                   4   // 0x04
16   #define TPM2B_DIGEST_ARRAY_MARSHAL_INDEX                 5   // 0x05
17   #define TPMT_HA_ARRAY_MARSHAL_INDEX                      6   // 0x06
18   #define TPMS_PCR_SELECTION_ARRAY_MARSHAL_INDEX           7   // 0x07
19   #define TPMS_ALG_PROPERTY_ARRAY_MARSHAL_INDEX            8   // 0x08
20   #define TPMS_TAGGED_PROPERTY_ARRAY_MARSHAL_INDEX         9   // 0x09
21   #define TPMS_TAGGED_PCR_SELECT_ARRAY_MARSHAL_INDEX       10  // 0x0A
22   #define TPM_ECC_CURVE_ARRAY_MARSHAL_INDEX                11  // 0x0B
23   #define TPMS_TAGGED_POLICY_ARRAY_MARSHAL_INDEX           12  // 0x0C
24   #define TPMS_ACT_DATA_ARRAY_MARSHAL_INDEX                13  // 0x0D
25   #define TPMS_AC_OUTPUT_ARRAY_MARSHAL_INDEX               14  // 0x0E
```

Defines for referencing a type by offset

```
26   #define UINT8_MARSHAL_REF                                \
27               ((UINT16)(offsetof(MarshalData_st, UINT8_DATA)))
28   #define BYTE_MARSHAL_REF                                 UINT8_MARSHAL_REF
29   #define TPM_HT_MARSHAL_REF                               UINT8_MARSHAL_REF
30   #define TPMA_LOCALITY_MARSHAL_REF                        UINT8_MARSHAL_REF
31   #define UINT16_MARSHAL_REF                               \
32               ((UINT16)(offsetof(MarshalData_st, UINT16_DATA)))
33   #define TPM_KEY_SIZE_MARSHAL_REF                         UINT16_MARSHAL_REF
34   #define TPM_KEY_BITS_MARSHAL_REF                         UINT16_MARSHAL_REF
35   #define TPM_ALG_ID_MARSHAL_REF                           UINT16_MARSHAL_REF
36   #define TPM_ST_MARSHAL_REF                               UINT16_MARSHAL_REF
37   #define UINT32_MARSHAL_REF                               \
38               ((UINT16)(offsetof(MarshalData_st, UINT32_DATA)))
39   #define TPM_ALGORITHM_ID_MARSHAL_REF                     UINT32_MARSHAL_REF
40   #define TPM_MODIFIER_INDICATOR_MARSHAL_REF               UINT32_MARSHAL_REF
41   #define TPM_AUTHORIZATION_SIZE_MARSHAL_REF               UINT32_MARSHAL_REF
42   #define TPM_PARAMETER_SIZE_MARSHAL_REF                   UINT32_MARSHAL_REF
43   #define TPM_SPEC_MARSHAL_REF                             UINT32_MARSHAL_REF
```

 Copyright © TCG 2006-2020

```
44 #define TPM_GENERATED_MARSHAL_REF                  UINT32_MARSHAL_REF
45 #define TPM_CC_MARSHAL_REF                         UINT32_MARSHAL_REF
46 #define TPM_RC_MARSHAL_REF                         UINT32_MARSHAL_REF
47 #define TPM_PT_MARSHAL_REF                         UINT32_MARSHAL_REF
48 #define TPM_PT_PCR_MARSHAL_REF                     UINT32_MARSHAL_REF
49 #define TPM_PS_MARSHAL_REF                         UINT32_MARSHAL_REF
50 #define TPM_HANDLE_MARSHAL_REF                     UINT32_MARSHAL_REF
51 #define TPM_RH_MARSHAL_REF                         UINT32_MARSHAL_REF
52 #define TPM_HC_MARSHAL_REF                         UINT32_MARSHAL_REF
53 #define TPMA_PERMANENT_MARSHAL_REF                 UINT32_MARSHAL_REF
54 #define TPMA_STARTUP_CLEAR_MARSHAL_REF             UINT32_MARSHAL_REF
55 #define TPMA_MEMORY_MARSHAL_REF                    UINT32_MARSHAL_REF
56 #define TPMA_CC_MARSHAL_REF                        UINT32_MARSHAL_REF
57 #define TPMA_MODES_MARSHAL_REF                     UINT32_MARSHAL_REF
58 #define TPMA_X509_KEY_USAGE_MARSHAL_REF            UINT32_MARSHAL_REF
59 #define TPM_NV_INDEX_MARSHAL_REF                   UINT32_MARSHAL_REF
60 #define TPM_AE_MARSHAL_REF                         UINT32_MARSHAL_REF
61 #define UINT64_MARSHAL_REF                         \
62             ((UINT16)(offsetof(MarshalData_st, UINT64_DATA)))
63 #define INT8_MARSHAL_REF                           \
64             ((UINT16)(offsetof(MarshalData_st, INT8_DATA)))
65 #define INT16_MARSHAL_REF                          \
66             ((UINT16)(offsetof(MarshalData_st, INT16_DATA)))
67 #define INT32_MARSHAL_REF                          \
68             ((UINT16)(offsetof(MarshalData_st, INT32_DATA)))
69 #define INT64_MARSHAL_REF                          \
70             ((UINT16)(offsetof(MarshalData_st, INT64_DATA)))
71 #define UINT0_MARSHAL_REF                          \
72             ((UINT16)(offsetof(MarshalData_st, UINT0_DATA)))
73 #define TPM_ECC_CURVE_MARSHAL_REF                  \
74             ((UINT16)(offsetof(MarshalData_st, TPM_ECC_CURVE_DATA)))
75 #define TPM_CLOCK_ADJUST_MARSHAL_REF               \
76             ((UINT16)(offsetof(MarshalData_st, TPM_CLOCK_ADJUST_DATA)))
77 #define TPM_EO_MARSHAL_REF                         \
78             ((UINT16)(offsetof(MarshalData_st, TPM_EO_DATA)))
79 #define TPM_SU_MARSHAL_REF                         \
80             ((UINT16)(offsetof(MarshalData_st, TPM_SU_DATA)))
81 #define TPM_SE_MARSHAL_REF                         \
82             ((UINT16)(offsetof(MarshalData_st, TPM_SE_DATA)))
83 #define TPM_CAP_MARSHAL_REF                        \
84             ((UINT16)(offsetof(MarshalData_st, TPM_CAP_DATA)))
85 #define TPMA_ALGORITHM_MARSHAL_REF                 \
86             ((UINT16)(offsetof(MarshalData_st, TPMA_ALGORITHM_DATA)))
87 #define TPMA_OBJECT_MARSHAL_REF                    \
88             ((UINT16)(offsetof(MarshalData_st, TPMA_OBJECT_DATA)))
89 #define TPMA_SESSION_MARSHAL_REF                   \
90             ((UINT16)(offsetof(MarshalData_st, TPMA_SESSION_DATA)))
91 #define TPMA_ACT_MARSHAL_REF                       \
92             ((UINT16)(offsetof(MarshalData_st, TPMA_ACT_DATA)))
93 #define TPMI_YES_NO_MARSHAL_REF                    \
94             ((UINT16)(offsetof(MarshalData_st, TPMI_YES_NO_DATA)))
95 #define TPMI_DH_OBJECT_MARSHAL_REF                 \
96             ((UINT16)(offsetof(MarshalData_st, TPMI_DH_OBJECT_DATA)))
97 #define TPMI_DH_PARENT_MARSHAL_REF                 \
98             ((UINT16)(offsetof(MarshalData_st, TPMI_DH_PARENT_DATA)))
99 #define TPMI_DH_PERSISTENT_MARSHAL_REF             \
100            ((UINT16)(offsetof(MarshalData_st, TPMI_DH_PERSISTENT_DATA)))
101 #define TPMI_DH_ENTITY_MARSHAL_REF                 \
102            ((UINT16)(offsetof(MarshalData_st, TPMI_DH_ENTITY_DATA)))
103 #define TPMI_DH_PCR_MARSHAL_REF                    \
104            ((UINT16)(offsetof(MarshalData_st, TPMI_DH_PCR_DATA)))
105 #define TPMI_SH_AUTH_SESSION_MARSHAL_REF           \
106            ((UINT16)(offsetof(MarshalData_st, TPMI_SH_AUTH_SESSION_DATA)))
107 #define TPMI_SH_HMAC_MARSHAL_REF                   \
108            ((UINT16)(offsetof(MarshalData_st, TPMI_SH_HMAC_DATA)))
109 #define TPMI_SH_POLICY_MARSHAL_REF                 \
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
110                 ((UINT16)(offsetof(MarshalData_st, TPMI_SH_POLICY_DATA)))
111     #define TPMI_DH_CONTEXT_MARSHAL_REF                   \
112                 ((UINT16)(offsetof(MarshalData_st, TPMI_DH_CONTEXT_DATA)))
113     #define TPMI_DH_SAVED_MARSHAL_REF                     \
114                 ((UINT16)(offsetof(MarshalData_st, TPMI_DH_SAVED_DATA)))
115     #define TPMI_RH_HIERARCHY_MARSHAL_REF                 \
116                 ((UINT16)(offsetof(MarshalData_st, TPMI_RH_HIERARCHY_DATA)))
117     #define TPMI_RH_ENABLES_MARSHAL_REF                   \
118                 ((UINT16)(offsetof(MarshalData_st, TPMI_RH_ENABLES_DATA)))
119     #define TPMI_RH_HIERARCHY_AUTH_MARSHAL_REF            \
120                 ((UINT16)(offsetof(MarshalData_st, TPMI_RH_HIERARCHY_AUTH_DATA)))
121     #define TPMI_RH_HIERARCHY_POLICY_MARSHAL_REF          \
122                 ((UINT16)(offsetof(MarshalData_st, TPMI_RH_HIERARCHY_POLICY_DATA)))
123     #define TPMI_RH_PLATFORM_MARSHAL_REF                  \
124                 ((UINT16)(offsetof(MarshalData_st, TPMI_RH_PLATFORM_DATA)))
125     #define TPMI_RH_OWNER_MARSHAL_REF                     \
126                 ((UINT16)(offsetof(MarshalData_st, TPMI_RH_OWNER_DATA)))
127     #define TPMI_RH_ENDORSEMENT_MARSHAL_REF               \
128                 ((UINT16)(offsetof(MarshalData_st, TPMI_RH_ENDORSEMENT_DATA)))
129     #define TPMI_RH_PROVISION_MARSHAL_REF                 \
130                 ((UINT16)(offsetof(MarshalData_st, TPMI_RH_PROVISION_DATA)))
131     #define TPMI_RH_CLEAR_MARSHAL_REF                     \
132                 ((UINT16)(offsetof(MarshalData_st, TPMI_RH_CLEAR_DATA)))
133     #define TPMI_RH_NV_AUTH_MARSHAL_REF                   \
134                 ((UINT16)(offsetof(MarshalData_st, TPMI_RH_NV_AUTH_DATA)))
135     #define TPMI_RH_LOCKOUT_MARSHAL_REF                   \
136                 ((UINT16)(offsetof(MarshalData_st, TPMI_RH_LOCKOUT_DATA)))
137     #define TPMI_RH_NV_INDEX_MARSHAL_REF                  \
138                 ((UINT16)(offsetof(MarshalData_st, TPMI_RH_NV_INDEX_DATA)))
139     #define TPMI_RH_AC_MARSHAL_REF                        \
140                 ((UINT16)(offsetof(MarshalData_st, TPMI_RH_AC_DATA)))
141     #define TPMI_RH_ACT_MARSHAL_REF                       \
142                 ((UINT16)(offsetof(MarshalData_st, TPMI_RH_ACT_DATA)))
143     #define TPMI_ALG_HASH_MARSHAL_REF                     \
144                 ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_HASH_DATA)))
145     #define TPMI_ALG_ASYM_MARSHAL_REF                     \
146                 ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_ASYM_DATA)))
147     #define TPMI_ALG_SYM_MARSHAL_REF                      \
148                 ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_SYM_DATA)))
149     #define TPMI_ALG_SYM_OBJECT_MARSHAL_REF               \
150                 ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_SYM_OBJECT_DATA)))
151     #define TPMI_ALG_SYM_MODE_MARSHAL_REF                 \
152                 ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_SYM_MODE_DATA)))
153     #define TPMI_ALG_KDF_MARSHAL_REF                      \
154                 ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_KDF_DATA)))
155     #define TPMI_ALG_SIG_SCHEME_MARSHAL_REF               \
156                 ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_SIG_SCHEME_DATA)))
157     #define TPMI_ECC_KEY_EXCHANGE_MARSHAL_REF             \
158                 ((UINT16)(offsetof(MarshalData_st, TPMI_ECC_KEY_EXCHANGE_DATA)))
159     #define TPMI_ST_COMMAND_TAG_MARSHAL_REF               \
160                 ((UINT16)(offsetof(MarshalData_st, TPMI_ST_COMMAND_TAG_DATA)))
161     #define TPMI_ALG_MAC_SCHEME_MARSHAL_REF               \
162                 ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_MAC_SCHEME_DATA)))
163     #define TPMI_ALG_CIPHER_MODE_MARSHAL_REF              \
164                 ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_CIPHER_MODE_DATA)))
165     #define TPMS_EMPTY_MARSHAL_REF                        \
166                 ((UINT16)(offsetof(MarshalData_st, TPMS_EMPTY_DATA)))
167     #define TPMS_ENC_SCHEME_RSAES_MARSHAL_REF             TPMS_EMPTY_MARSHAL_REF
168     #define TPMS_ALGORITHM_DESCRIPTION_MARSHAL_REF        \
169                 ((UINT16)(offsetof(MarshalData_st, TPMS_ALGORITHM_DESCRIPTION_DATA)))
170     #define TPMU_HA_MARSHAL_REF                           \
171                 ((UINT16)(offsetof(MarshalData_st, TPMU_HA_DATA)))
172     #define TPMT_HA_MARSHAL_REF                           \
173                 ((UINT16)(offsetof(MarshalData_st, TPMT_HA_DATA)))
174     #define TPM2B_DIGEST_MARSHAL_REF                      \
175                 ((UINT16)(offsetof(MarshalData_st, TPM2B_DIGEST_DATA)))
```

 Copyright © TCG 2006-2020

```
176   #define TPM2B_NONCE_MARSHAL_REF                       TPM2B_DIGEST_MARSHAL_REF
177   #define TPM2B_AUTH_MARSHAL_REF                        TPM2B_DIGEST_MARSHAL_REF
178   #define TPM2B_OPERAND_MARSHAL_REF                     TPM2B_DIGEST_MARSHAL_REF
179   #define TPM2B_DATA_MARSHAL_REF                        \
180           ((UINT16)(offsetof(MarshalData_st, TPM2B_DATA_DATA)))
181   #define TPM2B_EVENT_MARSHAL_REF                       \
182           ((UINT16)(offsetof(MarshalData_st, TPM2B_EVENT_DATA)))
183   #define TPM2B_MAX_BUFFER_MARSHAL_REF                  \
184           ((UINT16)(offsetof(MarshalData_st, TPM2B_MAX_BUFFER_DATA)))
185   #define TPM2B_MAX_NV_BUFFER_MARSHAL_REF               \
186           ((UINT16)(offsetof(MarshalData_st, TPM2B_MAX_NV_BUFFER_DATA)))
187   #define TPM2B_TIMEOUT_MARSHAL_REF                     \
188           ((UINT16)(offsetof(MarshalData_st, TPM2B_TIMEOUT_DATA)))
189   #define TPM2B_IV_MARSHAL_REF                          \
190           ((UINT16)(offsetof(MarshalData_st, TPM2B_IV_DATA)))
191   #define NULL_UNION_MARSHAL_REF                        \
192           ((UINT16)(offsetof(MarshalData_st, NULL_UNION_DATA)))
193   #define TPMU_NAME_MARSHAL_REF                         NULL_UNION_MARSHAL_REF
194   #define TPMU_SENSITIVE_CREATE_MARSHAL_REF             NULL_UNION_MARSHAL_REF
195   #define TPM2B_NAME_MARSHAL_REF                        \
196           ((UINT16)(offsetof(MarshalData_st, TPM2B_NAME_DATA)))
197   #define TPMS_PCR_SELECT_MARSHAL_REF                   \
198           ((UINT16)(offsetof(MarshalData_st, TPMS_PCR_SELECT_DATA)))
199   #define TPMS_PCR_SELECTION_MARSHAL_REF                \
200           ((UINT16)(offsetof(MarshalData_st, TPMS_PCR_SELECTION_DATA)))
201   #define TPMT_TK_CREATION_MARSHAL_REF                  \
202           ((UINT16)(offsetof(MarshalData_st, TPMT_TK_CREATION_DATA)))
203   #define TPMT_TK_VERIFIED_MARSHAL_REF                  \
204           ((UINT16)(offsetof(MarshalData_st, TPMT_TK_VERIFIED_DATA)))
205   #define TPMT_TK_AUTH_MARSHAL_REF                      \
206           ((UINT16)(offsetof(MarshalData_st, TPMT_TK_AUTH_DATA)))
207   #define TPMT_TK_HASHCHECK_MARSHAL_REF                 \
208           ((UINT16)(offsetof(MarshalData_st, TPMT_TK_HASHCHECK_DATA)))
209   #define TPMS_ALG_PROPERTY_MARSHAL_REF                 \
210           ((UINT16)(offsetof(MarshalData_st, TPMS_ALG_PROPERTY_DATA)))
211   #define TPMS_TAGGED_PROPERTY_MARSHAL_REF              \
212           ((UINT16)(offsetof(MarshalData_st, TPMS_TAGGED_PROPERTY_DATA)))
213   #define TPMS_TAGGED_PCR_SELECT_MARSHAL_REF            \
214           ((UINT16)(offsetof(MarshalData_st, TPMS_TAGGED_PCR_SELECT_DATA)))
215   #define TPMS_TAGGED_POLICY_MARSHAL_REF                \
216           ((UINT16)(offsetof(MarshalData_st, TPMS_TAGGED_POLICY_DATA)))
217   #define TPMS_ACT_DATA_MARSHAL_REF                     \
218           ((UINT16)(offsetof(MarshalData_st, TPMS_ACT_DATA_DATA)))
219   #define TPML_CC_MARSHAL_REF                           \
220           ((UINT16)(offsetof(MarshalData_st, TPML_CC_DATA)))
221   #define TPML_CCA_MARSHAL_REF                          \
222           ((UINT16)(offsetof(MarshalData_st, TPML_CCA_DATA)))
223   #define TPML_ALG_MARSHAL_REF                          \
224           ((UINT16)(offsetof(MarshalData_st, TPML_ALG_DATA)))
225   #define TPML_HANDLE_MARSHAL_REF                       \
226           ((UINT16)(offsetof(MarshalData_st, TPML_HANDLE_DATA)))
227   #define TPML_DIGEST_MARSHAL_REF                       \
228           ((UINT16)(offsetof(MarshalData_st, TPML_DIGEST_DATA)))
229   #define TPML_DIGEST_VALUES_MARSHAL_REF                \
230           ((UINT16)(offsetof(MarshalData_st, TPML_DIGEST_VALUES_DATA)))
231   #define TPML_PCR_SELECTION_MARSHAL_REF                \
232           ((UINT16)(offsetof(MarshalData_st, TPML_PCR_SELECTION_DATA)))
233   #define TPML_ALG_PROPERTY_MARSHAL_REF                 \
234           ((UINT16)(offsetof(MarshalData_st, TPML_ALG_PROPERTY_DATA)))
235   #define TPML_TAGGED_TPM_PROPERTY_MARSHAL_REF          \
236           ((UINT16)(offsetof(MarshalData_st, TPML_TAGGED_TPM_PROPERTY_DATA)))
237   #define TPML_TAGGED_PCR_PROPERTY_MARSHAL_REF          \
238           ((UINT16)(offsetof(MarshalData_st, TPML_TAGGED_PCR_PROPERTY_DATA)))
239   #define TPML_ECC_CURVE_MARSHAL_REF                    \
240           ((UINT16)(offsetof(MarshalData_st, TPML_ECC_CURVE_DATA)))
241   #define TPML_TAGGED_POLICY_MARSHAL_REF                \
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
242                 ((UINT16)(offsetof(MarshalData_st, TPML_TAGGED_POLICY_DATA)))
243     #define TPML_ACT_DATA_MARSHAL_REF                           \
244                 ((UINT16)(offsetof(MarshalData_st, TPML_ACT_DATA_DATA)))
245     #define TPMU_CAPABILITIES_MARSHAL_REF                       \
246                 ((UINT16)(offsetof(MarshalData_st, TPMU_CAPABILITIES_DATA)))
247     #define TPMS_CAPABILITY_DATA_MARSHAL_REF                    \
248                 ((UINT16)(offsetof(MarshalData_st, TPMS_CAPABILITY_DATA_DATA)))
249     #define TPMS_CLOCK_INFO_MARSHAL_REF                         \
250                 ((UINT16)(offsetof(MarshalData_st, TPMS_CLOCK_INFO_DATA)))
251     #define TPMS_TIME_INFO_MARSHAL_REF                          \
252                 ((UINT16)(offsetof(MarshalData_st, TPMS_TIME_INFO_DATA)))
253     #define TPMS_TIME_ATTEST_INFO_MARSHAL_REF                   \
254                 ((UINT16)(offsetof(MarshalData_st, TPMS_TIME_ATTEST_INFO_DATA)))
255     #define TPMS_CERTIFY_INFO_MARSHAL_REF                       \
256                 ((UINT16)(offsetof(MarshalData_st, TPMS_CERTIFY_INFO_DATA)))
257     #define TPMS_QUOTE_INFO_MARSHAL_REF                         \
258                 ((UINT16)(offsetof(MarshalData_st, TPMS_QUOTE_INFO_DATA)))
259     #define TPMS_COMMAND_AUDIT_INFO_MARSHAL_REF                 \
260                 ((UINT16)(offsetof(MarshalData_st, TPMS_COMMAND_AUDIT_INFO_DATA)))
261     #define TPMS_SESSION_AUDIT_INFO_MARSHAL_REF                 \
262                 ((UINT16)(offsetof(MarshalData_st, TPMS_SESSION_AUDIT_INFO_DATA)))
263     #define TPMS_CREATION_INFO_MARSHAL_REF                      \
264                 ((UINT16)(offsetof(MarshalData_st, TPMS_CREATION_INFO_DATA)))
265     #define TPMS_NV_CERTIFY_INFO_MARSHAL_REF                    \
266                 ((UINT16)(offsetof(MarshalData_st, TPMS_NV_CERTIFY_INFO_DATA)))
267     #define TPMS_NV_DIGEST_CERTIFY_INFO_MARSHAL_REF             \
268                 ((UINT16)(offsetof(MarshalData_st, TPMS_NV_DIGEST_CERTIFY_INFO_DATA)))
269     #define TPMI_ST_ATTEST_MARSHAL_REF                          \
270                 ((UINT16)(offsetof(MarshalData_st, TPMI_ST_ATTEST_DATA)))
271     #define TPMU_ATTEST_MARSHAL_REF                             \
272                 ((UINT16)(offsetof(MarshalData_st, TPMU_ATTEST_DATA)))
273     #define TPMS_ATTEST_MARSHAL_REF                             \
274                 ((UINT16)(offsetof(MarshalData_st, TPMS_ATTEST_DATA)))
275     #define TPM2B_ATTEST_MARSHAL_REF                            \
276                 ((UINT16)(offsetof(MarshalData_st, TPM2B_ATTEST_DATA)))
277     #define TPMS_AUTH_COMMAND_MARSHAL_REF                       \
278                 ((UINT16)(offsetof(MarshalData_st, TPMS_AUTH_COMMAND_DATA)))
279     #define TPMS_AUTH_RESPONSE_MARSHAL_REF                      \
280                 ((UINT16)(offsetof(MarshalData_st, TPMS_AUTH_RESPONSE_DATA)))
281     #define TPMI_TDES_KEY_BITS_MARSHAL_REF                      \
282                 ((UINT16)(offsetof(MarshalData_st, TPMI_TDES_KEY_BITS_DATA)))
283     #define TPMI_AES_KEY_BITS_MARSHAL_REF                       \
284                 ((UINT16)(offsetof(MarshalData_st, TPMI_AES_KEY_BITS_DATA)))
285     #define TPMI_SM4_KEY_BITS_MARSHAL_REF                       \
286                 ((UINT16)(offsetof(MarshalData_st, TPMI_SM4_KEY_BITS_DATA)))
287     #define TPMI_CAMELLIA_KEY_BITS_MARSHAL_REF                  \
288                 ((UINT16)(offsetof(MarshalData_st, TPMI_CAMELLIA_KEY_BITS_DATA)))
289     #define TPMU_SYM_KEY_BITS_MARSHAL_REF                       \
290                 ((UINT16)(offsetof(MarshalData_st, TPMU_SYM_KEY_BITS_DATA)))
291     #define TPMU_SYM_MODE_MARSHAL_REF                           \
292                 ((UINT16)(offsetof(MarshalData_st, TPMU_SYM_MODE_DATA)))
293     #define TPMT_SYM_DEF_MARSHAL_REF                            \
294                 ((UINT16)(offsetof(MarshalData_st, TPMT_SYM_DEF_DATA)))
295     #define TPMT_SYM_DEF_OBJECT_MARSHAL_REF                     \
296                 ((UINT16)(offsetof(MarshalData_st, TPMT_SYM_DEF_OBJECT_DATA)))
297     #define TPM2B_SYM_KEY_MARSHAL_REF                           \
298                 ((UINT16)(offsetof(MarshalData_st, TPM2B_SYM_KEY_DATA)))
299     #define TPMS_SYMCIPHER_PARMS_MARSHAL_REF                    \
300                 ((UINT16)(offsetof(MarshalData_st, TPMS_SYMCIPHER_PARMS_DATA)))
301     #define TPM2B_LABEL_MARSHAL_REF                             \
302                 ((UINT16)(offsetof(MarshalData_st, TPM2B_LABEL_DATA)))
303     #define TPMS_DERIVE_MARSHAL_REF                             \
304                 ((UINT16)(offsetof(MarshalData_st, TPMS_DERIVE_DATA)))
305     #define TPM2B_DERIVE_MARSHAL_REF                            \
306                 ((UINT16)(offsetof(MarshalData_st, TPM2B_DERIVE_DATA)))
307     #define TPM2B_SENSITIVE_DATA_MARSHAL_REF                    \
```

```
308                 ((UINT16)(offsetof(MarshalData_st, TPM2B_SENSITIVE_DATA_DATA)))
309     #define TPMS_SENSITIVE_CREATE_MARSHAL_REF                \
310                 ((UINT16)(offsetof(MarshalData_st, TPMS_SENSITIVE_CREATE_DATA)))
311     #define TPM2B_SENSITIVE_CREATE_MARSHAL_REF               \
312                 ((UINT16)(offsetof(MarshalData_st, TPM2B_SENSITIVE_CREATE_DATA)))
313     #define TPMS_SCHEME_HASH_MARSHAL_REF                     \
314                 ((UINT16)(offsetof(MarshalData_st, TPMS_SCHEME_HASH_DATA)))
315     #define TPMS_SCHEME_HMAC_MARSHAL_REF                     TPMS_SCHEME_HASH_MARSHAL_REF
316     #define TPMS_SIG_SCHEME_RSASSA_MARSHAL_REF               TPMS_SCHEME_HASH_MARSHAL_REF
317     #define TPMS_SIG_SCHEME_RSAPSS_MARSHAL_REF               TPMS_SCHEME_HASH_MARSHAL_REF
318     #define TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF                TPMS_SCHEME_HASH_MARSHAL_REF
319     #define TPMS_SIG_SCHEME_SM2_MARSHAL_REF                  TPMS_SCHEME_HASH_MARSHAL_REF
320     #define TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_REF            TPMS_SCHEME_HASH_MARSHAL_REF
321     #define TPMS_ENC_SCHEME_OAEP_MARSHAL_REF                 TPMS_SCHEME_HASH_MARSHAL_REF
322     #define TPMS_KEY_SCHEME_ECDH_MARSHAL_REF                 TPMS_SCHEME_HASH_MARSHAL_REF
323     #define TPMS_KEY_SCHEME_ECMQV_MARSHAL_REF                TPMS_SCHEME_HASH_MARSHAL_REF
324     #define TPMS_SCHEME_MGF1_MARSHAL_REF                     TPMS_SCHEME_HASH_MARSHAL_REF
325     #define TPMS_SCHEME_KDF1_SP800_56A_MARSHAL_REF           TPMS_SCHEME_HASH_MARSHAL_REF
326     #define TPMS_SCHEME_KDF2_MARSHAL_REF                     TPMS_SCHEME_HASH_MARSHAL_REF
327     #define TPMS_SCHEME_KDF1_SP800_108_MARSHAL_REF           TPMS_SCHEME_HASH_MARSHAL_REF
328     #define TPMS_SCHEME_ECDAA_MARSHAL_REF                    \
329                 ((UINT16)(offsetof(MarshalData_st, TPMS_SCHEME_ECDAA_DATA)))
330     #define TPMS_SIG_SCHEME_ECDAA_MARSHAL_REF                TPMS_SCHEME_ECDAA_MARSHAL_REF
331     #define TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_REF            \
332                 ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_KEYEDHASH_SCHEME_DATA)))
333     #define TPMS_SCHEME_XOR_MARSHAL_REF                      \
334                 ((UINT16)(offsetof(MarshalData_st, TPMS_SCHEME_XOR_DATA)))
335     #define TPMU_SCHEME_KEYEDHASH_MARSHAL_REF                \
336                 ((UINT16)(offsetof(MarshalData_st, TPMU_SCHEME_KEYEDHASH_DATA)))
337     #define TPMT_KEYEDHASH_SCHEME_MARSHAL_REF                \
338                 ((UINT16)(offsetof(MarshalData_st, TPMT_KEYEDHASH_SCHEME_DATA)))
339     #define TPMU_SIG_SCHEME_MARSHAL_REF                      \
340                 ((UINT16)(offsetof(MarshalData_st, TPMU_SIG_SCHEME_DATA)))
341     #define TPMT_SIG_SCHEME_MARSHAL_REF                      \
342                 ((UINT16)(offsetof(MarshalData_st, TPMT_SIG_SCHEME_DATA)))
343     #define TPMU_KDF_SCHEME_MARSHAL_REF                      \
344                 ((UINT16)(offsetof(MarshalData_st, TPMU_KDF_SCHEME_DATA)))
345     #define TPMT_KDF_SCHEME_MARSHAL_REF                      \
346                 ((UINT16)(offsetof(MarshalData_st, TPMT_KDF_SCHEME_DATA)))
347     #define TPMI_ALG_ASYM_SCHEME_MARSHAL_REF                 \
348                 ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_ASYM_SCHEME_DATA)))
349     #define TPMU_ASYM_SCHEME_MARSHAL_REF                     \
350                 ((UINT16)(offsetof(MarshalData_st, TPMU_ASYM_SCHEME_DATA)))
351     #define TPMI_ALG_RSA_SCHEME_MARSHAL_REF                  \
352                 ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_RSA_SCHEME_DATA)))
353     #define TPMT_RSA_SCHEME_MARSHAL_REF                      \
354                 ((UINT16)(offsetof(MarshalData_st, TPMT_RSA_SCHEME_DATA)))
355     #define TPMI_ALG_RSA_DECRYPT_MARSHAL_REF                 \
356                 ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_RSA_DECRYPT_DATA)))
357     #define TPMT_RSA_DECRYPT_MARSHAL_REF                     \
358                 ((UINT16)(offsetof(MarshalData_st, TPMT_RSA_DECRYPT_DATA)))
359     #define TPM2B_PUBLIC_KEY_RSA_MARSHAL_REF                 \
360                 ((UINT16)(offsetof(MarshalData_st, TPM2B_PUBLIC_KEY_RSA_DATA)))
361     #define TPMI_RSA_KEY_BITS_MARSHAL_REF                    \
362                 ((UINT16)(offsetof(MarshalData_st, TPMI_RSA_KEY_BITS_DATA)))
363     #define TPM2B_PRIVATE_KEY_RSA_MARSHAL_REF                \
364                 ((UINT16)(offsetof(MarshalData_st, TPM2B_PRIVATE_KEY_RSA_DATA)))
365     #define TPM2B_ECC_PARAMETER_MARSHAL_REF                  \
366                 ((UINT16)(offsetof(MarshalData_st, TPM2B_ECC_PARAMETER_DATA)))
367     #define TPMS_ECC_POINT_MARSHAL_REF                       \
368                 ((UINT16)(offsetof(MarshalData_st, TPMS_ECC_POINT_DATA)))
369     #define TPM2B_ECC_POINT_MARSHAL_REF                      \
370                 ((UINT16)(offsetof(MarshalData_st, TPM2B_ECC_POINT_DATA)))
371     #define TPMI_ALG_ECC_SCHEME_MARSHAL_REF                  \
372                 ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_ECC_SCHEME_DATA)))
373     #define TPMI_ECC_CURVE_MARSHAL_REF                       \
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
374                 ((UINT16)(offsetof(MarshalData_st, TPMI_ECC_CURVE_DATA)))
375     #define TPMT_ECC_SCHEME_MARSHAL_REF                        \
376                 ((UINT16)(offsetof(MarshalData_st, TPMT_ECC_SCHEME_DATA)))
377     #define TPMS_ALGORITHM_DETAIL_ECC_MARSHAL_REF              \
378                 ((UINT16)(offsetof(MarshalData_st, TPMS_ALGORITHM_DETAIL_ECC_DATA)))
379     #define TPMS_SIGNATURE_RSA_MARSHAL_REF                     \
380                 ((UINT16)(offsetof(MarshalData_st, TPMS_SIGNATURE_RSA_DATA)))
381     #define TPMS_SIGNATURE_RSASSA_MARSHAL_REF                  TPMS_SIGNATURE_RSA_MARSHAL_REF
382     #define TPMS_SIGNATURE_RSAPSS_MARSHAL_REF                  TPMS_SIGNATURE_RSA_MARSHAL_REF
383     #define TPMS_SIGNATURE_ECC_MARSHAL_REF                     \
384                 ((UINT16)(offsetof(MarshalData_st, TPMS_SIGNATURE_ECC_DATA)))
385     #define TPMS_SIGNATURE_ECDAA_MARSHAL_REF                   TPMS_SIGNATURE_ECC_MARSHAL_REF
386     #define TPMS_SIGNATURE_ECDSA_MARSHAL_REF                   TPMS_SIGNATURE_ECC_MARSHAL_REF
387     #define TPMS_SIGNATURE_SM2_MARSHAL_REF                     TPMS_SIGNATURE_ECC_MARSHAL_REF
388     #define TPMS_SIGNATURE_ECSCHNORR_MARSHAL_REF               TPMS_SIGNATURE_ECC_MARSHAL_REF
389     #define TPMU_SIGNATURE_MARSHAL_REF                         \
390                 ((UINT16)(offsetof(MarshalData_st, TPMU_SIGNATURE_DATA)))
391     #define TPMT_SIGNATURE_MARSHAL_REF                         \
392                 ((UINT16)(offsetof(MarshalData_st, TPMT_SIGNATURE_DATA)))
393     #define TPMU_ENCRYPTED_SECRET_MARSHAL_REF                  \
394                 ((UINT16)(offsetof(MarshalData_st, TPMU_ENCRYPTED_SECRET_DATA)))
395     #define TPM2B_ENCRYPTED_SECRET_MARSHAL_REF                 \
396                 ((UINT16)(offsetof(MarshalData_st, TPM2B_ENCRYPTED_SECRET_DATA)))
397     #define TPMI_ALG_PUBLIC_MARSHAL_REF                        \
398                 ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_PUBLIC_DATA)))
399     #define TPMU_PUBLIC_ID_MARSHAL_REF                         \
400                 ((UINT16)(offsetof(MarshalData_st, TPMU_PUBLIC_ID_DATA)))
401     #define TPMS_KEYEDHASH_PARMS_MARSHAL_REF                   \
402                 ((UINT16)(offsetof(MarshalData_st, TPMS_KEYEDHASH_PARMS_DATA)))
403     #define TPMS_RSA_PARMS_MARSHAL_REF                         \
404                 ((UINT16)(offsetof(MarshalData_st, TPMS_RSA_PARMS_DATA)))
405     #define TPMS_ECC_PARMS_MARSHAL_REF                         \
406                 ((UINT16)(offsetof(MarshalData_st, TPMS_ECC_PARMS_DATA)))
407     #define TPMU_PUBLIC_PARMS_MARSHAL_REF                      \
408                 ((UINT16)(offsetof(MarshalData_st, TPMU_PUBLIC_PARMS_DATA)))
409     #define TPMT_PUBLIC_PARMS_MARSHAL_REF                      \
410                 ((UINT16)(offsetof(MarshalData_st, TPMT_PUBLIC_PARMS_DATA)))
411     #define TPMT_PUBLIC_MARSHAL_REF                            \
412                 ((UINT16)(offsetof(MarshalData_st, TPMT_PUBLIC_DATA)))
413     #define TPM2B_PUBLIC_MARSHAL_REF                           \
414                 ((UINT16)(offsetof(MarshalData_st, TPM2B_PUBLIC_DATA)))
415     #define TPM2B_TEMPLATE_MARSHAL_REF                         \
416                 ((UINT16)(offsetof(MarshalData_st, TPM2B_TEMPLATE_DATA)))
417     #define TPM2B_PRIVATE_VENDOR_SPECIFIC_MARSHAL_REF          \
418                 ((UINT16)(offsetof(MarshalData_st, TPM2B_PRIVATE_VENDOR_SPECIFIC_DATA)))
419     #define TPMU_SENSITIVE_COMPOSITE_MARSHAL_REF               \
420                 ((UINT16)(offsetof(MarshalData_st, TPMU_SENSITIVE_COMPOSITE_DATA)))
421     #define TPMT_SENSITIVE_MARSHAL_REF                         \
422                 ((UINT16)(offsetof(MarshalData_st, TPMT_SENSITIVE_DATA)))
423     #define TPM2B_SENSITIVE_MARSHAL_REF                        \
424                 ((UINT16)(offsetof(MarshalData_st, TPM2B_SENSITIVE_DATA)))
425     #define TPM2B_PRIVATE_MARSHAL_REF                          \
426                 ((UINT16)(offsetof(MarshalData_st, TPM2B_PRIVATE_DATA)))
427     #define TPM2B_ID_OBJECT_MARSHAL_REF                        \
428                 ((UINT16)(offsetof(MarshalData_st, TPM2B_ID_OBJECT_DATA)))
429     #define TPMS_NV_PIN_COUNTER_PARAMETERS_MARSHAL_REF         \
430                 ((UINT16)(offsetof(MarshalData_st, TPMS_NV_PIN_COUNTER_PARAMETERS_DATA)))
431     #define TPMA_NV_MARSHAL_REF                                \
432                 ((UINT16)(offsetof(MarshalData_st, TPMA_NV_DATA)))
433     #define TPMS_NV_PUBLIC_MARSHAL_REF                         \
434                 ((UINT16)(offsetof(MarshalData_st, TPMS_NV_PUBLIC_DATA)))
435     #define TPM2B_NV_PUBLIC_MARSHAL_REF                        \
436                 ((UINT16)(offsetof(MarshalData_st, TPM2B_NV_PUBLIC_DATA)))
437     #define TPM2B_CONTEXT_SENSITIVE_MARSHAL_REF                \
438                 ((UINT16)(offsetof(MarshalData_st, TPM2B_CONTEXT_SENSITIVE_DATA)))
439     #define TPMS_CONTEXT_DATA_MARSHAL_REF                      \
```

```
440                 ((UINT16)(offsetof(MarshalData_st, TPMS_CONTEXT_DATA_DATA)))
441     #define TPM2B_CONTEXT_DATA_MARSHAL_REF                       \
442                 ((UINT16)(offsetof(MarshalData_st, TPM2B_CONTEXT_DATA_DATA)))
443     #define TPMS_CONTEXT_MARSHAL_REF                             \
444                 ((UINT16)(offsetof(MarshalData_st, TPMS_CONTEXT_DATA)))
445     #define TPMS_CREATION_DATA_MARSHAL_REF                       \
446                 ((UINT16)(offsetof(MarshalData_st, TPMS_CREATION_DATA_DATA)))
447     #define TPM2B_CREATION_DATA_MARSHAL_REF                      \
448                 ((UINT16)(offsetof(MarshalData_st, TPM2B_CREATION_DATA_DATA)))
449     #define TPM_AT_MARSHAL_REF                                   \
450                 ((UINT16)(offsetof(MarshalData_st, TPM_AT_DATA)))
451     #define TPMS_AC_OUTPUT_MARSHAL_REF                           \
452                 ((UINT16)(offsetof(MarshalData_st, TPMS_AC_OUTPUT_DATA)))
453     #define TPML_AC_CAPABILITIES_MARSHAL_REF                     \
454                 ((UINT16)(offsetof(MarshalData_st, TPML_AC_CAPABILITIES_DATA)))
455     #define Type00_MARSHAL_REF                                   \
456                 ((UINT16)(offsetof(MarshalData_st, Type00_DATA)))
457     #define Type01_MARSHAL_REF                                   \
458                 ((UINT16)(offsetof(MarshalData_st, Type01_DATA)))
459     #define Type02_MARSHAL_REF                                   \
460                 ((UINT16)(offsetof(MarshalData_st, Type02_DATA)))
461     #define Type03_MARSHAL_REF                                   \
462                 ((UINT16)(offsetof(MarshalData_st, Type03_DATA)))
463     #define Type04_MARSHAL_REF                                   \
464                 ((UINT16)(offsetof(MarshalData_st, Type04_DATA)))
465     #define Type05_MARSHAL_REF                                   \
466                 ((UINT16)(offsetof(MarshalData_st, Type05_DATA)))
467     #define Type06_MARSHAL_REF                                   \
468                 ((UINT16)(offsetof(MarshalData_st, Type06_DATA)))
469     #define Type07_MARSHAL_REF                                   \
470                 ((UINT16)(offsetof(MarshalData_st, Type07_DATA)))
471     #define Type08_MARSHAL_REF                                   \
472                 ((UINT16)(offsetof(MarshalData_st, Type08_DATA)))
473     #define Type09_MARSHAL_REF                                   Type08_MARSHAL_REF
474     #define Type14_MARSHAL_REF                                   Type08_MARSHAL_REF
475     #define Type10_MARSHAL_REF                                   \
476                 ((UINT16)(offsetof(MarshalData_st, Type10_DATA)))
477     #define Type11_MARSHAL_REF                                   \
478                 ((UINT16)(offsetof(MarshalData_st, Type11_DATA)))
479     #define Type12_MARSHAL_REF                                   \
480                 ((UINT16)(offsetof(MarshalData_st, Type12_DATA)))
481     #define Type13_MARSHAL_REF                                   \
482                 ((UINT16)(offsetof(MarshalData_st, Type13_DATA)))
483     #define Type15_MARSHAL_REF                                   \
484                 ((UINT16)(offsetof(MarshalData_st, Type15_DATA)))
485     #define Type16_MARSHAL_REF                                   Type15_MARSHAL_REF
486     #define Type17_MARSHAL_REF                                   \
487                 ((UINT16)(offsetof(MarshalData_st, Type17_DATA)))
488     #define Type18_MARSHAL_REF                                   \
489                 ((UINT16)(offsetof(MarshalData_st, Type18_DATA)))
490     #define Type19_MARSHAL_REF                                   \
491                 ((UINT16)(offsetof(MarshalData_st, Type19_DATA)))
492     #define Type20_MARSHAL_REF                                   \
493                 ((UINT16)(offsetof(MarshalData_st, Type20_DATA)))
494     #define Type21_MARSHAL_REF                                   Type20_MARSHAL_REF
495     #define Type22_MARSHAL_REF                                   \
496                 ((UINT16)(offsetof(MarshalData_st, Type22_DATA)))
497     #define Type23_MARSHAL_REF                                   \
498                 ((UINT16)(offsetof(MarshalData_st, Type23_DATA)))
499     #define Type24_MARSHAL_REF                                   \
500                 ((UINT16)(offsetof(MarshalData_st, Type24_DATA)))
501     #define Type25_MARSHAL_REF                                   \
502                 ((UINT16)(offsetof(MarshalData_st, Type25_DATA)))
503     #define Type26_MARSHAL_REF                                   \
504                 ((UINT16)(offsetof(MarshalData_st, Type26_DATA)))
505     #define Type27_MARSHAL_REF                                   \
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
506             ((UINT16)(offsetof(MarshalData_st, Type27_DATA)))
507     #define Type28_MARSHAL_REF                              \
508             ((UINT16)(offsetof(MarshalData_st, Type28_DATA)))
509     #define Type29_MARSHAL_REF                              \
510             ((UINT16)(offsetof(MarshalData_st, Type29_DATA)))
511     #define Type30_MARSHAL_REF                              \
512             ((UINT16)(offsetof(MarshalData_st, Type30_DATA)))
513     #define Type31_MARSHAL_REF                              \
514             ((UINT16)(offsetof(MarshalData_st, Type31_DATA)))
515     #define Type32_MARSHAL_REF                              \
516             ((UINT16)(offsetof(MarshalData_st, Type32_DATA)))
517     #define Type33_MARSHAL_REF                              \
518             ((UINT16)(offsetof(MarshalData_st, Type33_DATA)))
519     #define Type34_MARSHAL_REF                              \
520             ((UINT16)(offsetof(MarshalData_st, Type34_DATA)))
521     #define Type35_MARSHAL_REF                              \
522             ((UINT16)(offsetof(MarshalData_st, Type35_DATA)))
523     #define Type36_MARSHAL_REF                              \
524             ((UINT16)(offsetof(MarshalData_st, Type36_DATA)))
525     #define Type37_MARSHAL_REF                              \
526             ((UINT16)(offsetof(MarshalData_st, Type37_DATA)))
527     #define Type38_MARSHAL_REF                              \
528             ((UINT16)(offsetof(MarshalData_st, Type38_DATA)))
529     #define Type39_MARSHAL_REF                              \
530             ((UINT16)(offsetof(MarshalData_st, Type39_DATA)))
531     #define Type40_MARSHAL_REF                              \
532             ((UINT16)(offsetof(MarshalData_st, Type40_DATA)))
533     #define Type41_MARSHAL_REF                              \
534             ((UINT16)(offsetof(MarshalData_st, Type41_DATA)))
535     #define Type42_MARSHAL_REF                              \
536             ((UINT16)(offsetof(MarshalData_st, Type42_DATA)))
537     #define Type43_MARSHAL_REF                              \
538             ((UINT16)(offsetof(MarshalData_st, Type43_DATA)))
539     #define Type44_MARSHAL_REF                              \
540             ((UINT16)(offsetof(MarshalData_st, Type44_DATA)))
541     //#defines to change calling sequence for code using marshaling
542     #define UINT8_Unmarshal(target, buffer, size)                               \
543         Unmarshal(UINT8_MARSHAL_REF, (target), (buffer), (size))
544     #define UINT8_Marshal(source, buffer, size)                                 \
545         Marshal(UINT8_MARSHAL_REF, (source), (buffer), (size))
546     #define BYTE_Unmarshal(target, buffer, size)                                \
547         Unmarshal(UINT8_MARSHAL_REF, (target), (buffer), (size))
548     #define BYTE_Marshal(source, buffer, size)                                  \
549         Marshal(UINT8_MARSHAL_REF, (source), (buffer), (size))
550     #define INT8_Unmarshal(target, buffer, size)                                \
551         Unmarshal(INT8_MARSHAL_REF, (target), (buffer), (size))
552     #define INT8_Marshal(source, buffer, size)                                  \
553         Marshal(INT8_MARSHAL_REF, (source), (buffer), (size))
554     #define UINT16_Unmarshal(target, buffer, size)                              \
555         Unmarshal(UINT16_MARSHAL_REF, (target), (buffer), (size))
556     #define UINT16_Marshal(source, buffer, size)                                \
557         Marshal(UINT16_MARSHAL_REF, (source), (buffer), (size))
558     #define INT16_Unmarshal(target, buffer, size)                               \
559         Unmarshal(INT16_MARSHAL_REF, (target), (buffer), (size))
560     #define INT16_Marshal(source, buffer, size)                                 \
561         Marshal(INT16_MARSHAL_REF, (source), (buffer), (size))
562     #define UINT32_Unmarshal(target, buffer, size)                              \
563         Unmarshal(UINT32_MARSHAL_REF, (target), (buffer), (size))
564     #define UINT32_Marshal(source, buffer, size)                                \
565         Marshal(UINT32_MARSHAL_REF, (source), (buffer), (size))
566     #define INT32_Unmarshal(target, buffer, size)                               \
567         Unmarshal(INT32_MARSHAL_REF, (target), (buffer), (size))
568     #define INT32_Marshal(source, buffer, size)                                 \
569         Marshal(INT32_MARSHAL_REF, (source), (buffer), (size))
570     #define UINT64_Unmarshal(target, buffer, size)                              \
571         Unmarshal(UINT64_MARSHAL_REF, (target), (buffer), (size))
```

```
572 #define UINT64_Marshal(source, buffer, size)                                      \
573     Marshal(UINT64_MARSHAL_REF, (source), (buffer), (size))
574 #define INT64_Unmarshal(target, buffer, size)                                     \
575     Unmarshal(INT64_MARSHAL_REF, (target), (buffer), (size))
576 #define INT64_Marshal(source, buffer, size)                                       \
577     Marshal(INT64_MARSHAL_REF, (source), (buffer), (size))
578 #define TPM_ALGORITHM_ID_Unmarshal(target, buffer, size)                          \
579     Unmarshal(TPM_ALGORITHM_ID_MARSHAL_REF, (target), (buffer), (size))
580 #define TPM_ALGORITHM_ID_Marshal(source, buffer, size)                            \
581     Marshal(TPM_ALGORITHM_ID_MARSHAL_REF, (source), (buffer), (size))
582 #define TPM_MODIFIER_INDICATOR_Unmarshal(target, buffer, size)                    \
583     Unmarshal(TPM_MODIFIER_INDICATOR_MARSHAL_REF, (target), (buffer), (size))
584 #define TPM_MODIFIER_INDICATOR_Marshal(source, buffer, size)                      \
585     Marshal(TPM_MODIFIER_INDICATOR_MARSHAL_REF, (source), (buffer), (size))
586 #define TPM_AUTHORIZATION_SIZE_Unmarshal(target, buffer, size)                    \
587     Unmarshal(TPM_AUTHORIZATION_SIZE_MARSHAL_REF, (target), (buffer), (size))
588 #define TPM_AUTHORIZATION_SIZE_Marshal(source, buffer, size)                      \
589     Marshal(TPM_AUTHORIZATION_SIZE_MARSHAL_REF, (source), (buffer), (size))
590 #define TPM_PARAMETER_SIZE_Unmarshal(target, buffer, size)                        \
591     Unmarshal(TPM_PARAMETER_SIZE_MARSHAL_REF, (target), (buffer), (size))
592 #define TPM_PARAMETER_SIZE_Marshal(source, buffer, size)                          \
593     Marshal(TPM_PARAMETER_SIZE_MARSHAL_REF, (source), (buffer), (size))
594 #define TPM_KEY_SIZE_Unmarshal(target, buffer, size)                              \
595     Unmarshal(TPM_KEY_SIZE_MARSHAL_REF, (target), (buffer), (size))
596 #define TPM_KEY_SIZE_Marshal(source, buffer, size)                                \
597     Marshal(TPM_KEY_SIZE_MARSHAL_REF, (source), (buffer), (size))
598 #define TPM_KEY_BITS_Unmarshal(target, buffer, size)                              \
599     Unmarshal(TPM_KEY_BITS_MARSHAL_REF, (target), (buffer), (size))
600 #define TPM_KEY_BITS_Marshal(source, buffer, size)                                \
601     Marshal(TPM_KEY_BITS_MARSHAL_REF, (source), (buffer), (size))
602 #define TPM_GENERATED_Marshal(source, buffer, size)                               \
603     Marshal(TPM_GENERATED_MARSHAL_REF, (source), (buffer), (size))
604 #define TPM_ALG_ID_Unmarshal(target, buffer, size)                                \
605     Unmarshal(TPM_ALG_ID_MARSHAL_REF, (target), (buffer), (size))
606 #define TPM_ALG_ID_Marshal(source, buffer, size)                                  \
607     Marshal(TPM_ALG_ID_MARSHAL_REF, (source), (buffer), (size))
608 #define TPM_ECC_CURVE_Unmarshal(target, buffer, size)                             \
609     Unmarshal(TPM_ECC_CURVE_MARSHAL_REF, (target), (buffer), (size))
610 #define TPM_ECC_CURVE_Marshal(source, buffer, size)                               \
611     Marshal(TPM_ECC_CURVE_MARSHAL_REF, (source), (buffer), (size))
612 #define TPM_CC_Unmarshal(target, buffer, size)                                    \
613     Unmarshal(TPM_CC_MARSHAL_REF, (target), (buffer), (size))
614 #define TPM_CC_Marshal(source, buffer, size)                                      \
615     Marshal(TPM_CC_MARSHAL_REF, (source), (buffer), (size))
616 #define TPM_RC_Marshal(source, buffer, size)                                      \
617     Marshal(TPM_RC_MARSHAL_REF, (source), (buffer), (size))
618 #define TPM_CLOCK_ADJUST_Unmarshal(target, buffer, size)                          \
619     Unmarshal(TPM_CLOCK_ADJUST_MARSHAL_REF, (target), (buffer), (size))
620 #define TPM_EO_Unmarshal(target, buffer, size)                                    \
621     Unmarshal(TPM_EO_MARSHAL_REF, (target), (buffer), (size))
622 #define TPM_EO_Marshal(source, buffer, size)                                      \
623     Marshal(TPM_EO_MARSHAL_REF, (source), (buffer), (size))
624 #define TPM_ST_Unmarshal(target, buffer, size)                                    \
625     Unmarshal(TPM_ST_MARSHAL_REF, (target), (buffer), (size))
626 #define TPM_ST_Marshal(source, buffer, size)                                      \
627     Marshal(TPM_ST_MARSHAL_REF, (source), (buffer), (size))
628 #define TPM_SU_Unmarshal(target, buffer, size)                                    \
629     Unmarshal(TPM_SU_MARSHAL_REF, (target), (buffer), (size))
630 #define TPM_SE_Unmarshal(target, buffer, size)                                    \
631     Unmarshal(TPM_SE_MARSHAL_REF, (target), (buffer), (size))
632 #define TPM_CAP_Unmarshal(target, buffer, size)                                   \
633     Unmarshal(TPM_CAP_MARSHAL_REF, (target), (buffer), (size))
634 #define TPM_CAP_Marshal(source, buffer, size)                                     \
635     Marshal(TPM_CAP_MARSHAL_REF, (source), (buffer), (size))
636 #define TPM_PT_Unmarshal(target, buffer, size)                                    \
637     Unmarshal(TPM_PT_MARSHAL_REF, (target), (buffer), (size))
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
638 #define TPM_PT_Marshal(source, buffer, size)                                     \
639     Marshal(TPM_PT_MARSHAL_REF, (source), (buffer), (size))
640 #define TPM_PT_PCR_Unmarshal(target, buffer, size)                               \
641     Unmarshal(TPM_PT_PCR_MARSHAL_REF, (target), (buffer), (size))
642 #define TPM_PT_PCR_Marshal(source, buffer, size)                                 \
643     Marshal(TPM_PT_PCR_MARSHAL_REF, (source), (buffer), (size))
644 #define TPM_PS_Marshal(source, buffer, size)                                     \
645     Marshal(TPM_PS_MARSHAL_REF, (source), (buffer), (size))
646 #define TPM_HANDLE_Unmarshal(target, buffer, size)                               \
647     Unmarshal(TPM_HANDLE_MARSHAL_REF, (target), (buffer), (size))
648 #define TPM_HANDLE_Marshal(source, buffer, size)                                 \
649     Marshal(TPM_HANDLE_MARSHAL_REF, (source), (buffer), (size))
650 #define TPM_HT_Unmarshal(target, buffer, size)                                   \
651     Unmarshal(TPM_HT_MARSHAL_REF, (target), (buffer), (size))
652 #define TPM_HT_Marshal(source, buffer, size)                                     \
653     Marshal(TPM_HT_MARSHAL_REF, (source), (buffer), (size))
654 #define TPM_RH_Unmarshal(target, buffer, size)                                   \
655     Unmarshal(TPM_RH_MARSHAL_REF, (target), (buffer), (size))
656 #define TPM_RH_Marshal(source, buffer, size)                                     \
657     Marshal(TPM_RH_MARSHAL_REF, (source), (buffer), (size))
658 #define TPM_HC_Unmarshal(target, buffer, size)                                   \
659     Unmarshal(TPM_HC_MARSHAL_REF, (target), (buffer), (size))
660 #define TPM_HC_Marshal(source, buffer, size)                                     \
661     Marshal(TPM_HC_MARSHAL_REF, (source), (buffer), (size))
662 #define TPMA_ALGORITHM_Unmarshal(target, buffer, size)                           \
663     Unmarshal(TPMA_ALGORITHM_MARSHAL_REF, (target), (buffer), (size))
664 #define TPMA_ALGORITHM_Marshal(source, buffer, size)                             \
665     Marshal(TPMA_ALGORITHM_MARSHAL_REF, (source), (buffer), (size))
666 #define TPMA_OBJECT_Unmarshal(target, buffer, size)                              \
667     Unmarshal(TPMA_OBJECT_MARSHAL_REF, (target), (buffer), (size))
668 #define TPMA_OBJECT_Marshal(source, buffer, size)                                \
669     Marshal(TPMA_OBJECT_MARSHAL_REF, (source), (buffer), (size))
670 #define TPMA_SESSION_Unmarshal(target, buffer, size)                             \
671     Unmarshal(TPMA_SESSION_MARSHAL_REF, (target), (buffer), (size))
672 #define TPMA_SESSION_Marshal(source, buffer, size)                               \
673     Marshal(TPMA_SESSION_MARSHAL_REF, (source), (buffer), (size))
674 #define TPMA_LOCALITY_Unmarshal(target, buffer, size)                            \
675     Unmarshal(TPMA_LOCALITY_MARSHAL_REF, (target), (buffer), (size))
676 #define TPMA_LOCALITY_Marshal(source, buffer, size)                              \
677     Marshal(TPMA_LOCALITY_MARSHAL_REF, (source), (buffer), (size))
678 #define TPMA_PERMANENT_Marshal(source, buffer, size)                             \
679     Marshal(TPMA_PERMANENT_MARSHAL_REF, (source), (buffer), (size))
680 #define TPMA_STARTUP_CLEAR_Marshal(source, buffer, size)                         \
681     Marshal(TPMA_STARTUP_CLEAR_MARSHAL_REF, (source), (buffer), (size))
682 #define TPMA_MEMORY_Marshal(source, buffer, size)                                \
683     Marshal(TPMA_MEMORY_MARSHAL_REF, (source), (buffer), (size))
684 #define TPMA_CC_Marshal(source, buffer, size)                                    \
685     Marshal(TPMA_CC_MARSHAL_REF, (source), (buffer), (size))
686 #define TPMA_MODES_Marshal(source, buffer, size)                                 \
687     Marshal(TPMA_MODES_MARSHAL_REF, (source), (buffer), (size))
688 #define TPMA_X509_KEY_USAGE_Marshal(source, buffer, size)                        \
689     Marshal(TPMA_X509_KEY_USAGE_MARSHAL_REF, (source), (buffer), (size))
690 #define TPMA_ACT_Unmarshal(target, buffer, size)                                 \
691     Unmarshal(TPMA_ACT_MARSHAL_REF, (target), (buffer), (size))
692 #define TPMA_ACT_Marshal(source, buffer, size)                                   \
693     Marshal(TPMA_ACT_MARSHAL_REF, (source), (buffer), (size))
694 #define TPMI_YES_NO_Unmarshal(target, buffer, size)                              \
695     Unmarshal(TPMI_YES_NO_MARSHAL_REF, (target), (buffer), (size))
696 #define TPMI_YES_NO_Marshal(source, buffer, size)                                \
697     Marshal(TPMI_YES_NO_MARSHAL_REF, (source), (buffer), (size))
698 #define TPMI_DH_OBJECT_Unmarshal(target, buffer, size, flag)                     \
699     Unmarshal(TPMI_DH_OBJECT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),\
700       (size))
701 #define TPMI_DH_OBJECT_Marshal(source, buffer, size)                             \
702     Marshal(TPMI_DH_OBJECT_MARSHAL_REF, (source), (buffer), (size))
703 #define TPMI_DH_PARENT_Unmarshal(target, buffer, size, flag)                     \
```

```
704         Unmarshal(TPMI_DH_PARENT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),\
705             (size))
706     #define TPMI_DH_PARENT_Marshal(source, buffer, size)                              \
707         Marshal(TPMI_DH_PARENT_MARSHAL_REF, (source), (buffer), (size))
708     #define TPMI_DH_PERSISTENT_Unmarshal(target, buffer, size)                        \
709         Unmarshal(TPMI_DH_PERSISTENT_MARSHAL_REF, (target), (buffer), (size))
710     #define TPMI_DH_PERSISTENT_Marshal(source, buffer, size)                          \
711         Marshal(TPMI_DH_PERSISTENT_MARSHAL_REF, (source), (buffer), (size))
712     #define TPMI_DH_ENTITY_Unmarshal(target, buffer, size, flag)                      \
713         Unmarshal(TPMI_DH_ENTITY_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),\
714             (size))
715     #define TPMI_DH_PCR_Unmarshal(target, buffer, size, flag)                         \
716         Unmarshal(TPMI_DH_PCR_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),   \
717             (size))
718     #define TPMI_SH_AUTH_SESSION_Unmarshal(target, buffer, size, flag)                \
719         Unmarshal(TPMI_SH_AUTH_SESSION_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),    \
720             (buffer), (size))
721     #define TPMI_SH_AUTH_SESSION_Marshal(source, buffer, size)                        \
722         Marshal(TPMI_SH_AUTH_SESSION_MARSHAL_REF, (source), (buffer), (size))
723     #define TPMI_SH_HMAC_Unmarshal(target, buffer, size)                              \
724         Unmarshal(TPMI_SH_HMAC_MARSHAL_REF, (target), (buffer), (size))
725     #define TPMI_SH_HMAC_Marshal(source, buffer, size)                                \
726         Marshal(TPMI_SH_HMAC_MARSHAL_REF, (source), (buffer), (size))
727     #define TPMI_SH_POLICY_Unmarshal(target, buffer, size)                            \
728         Unmarshal(TPMI_SH_POLICY_MARSHAL_REF, (target), (buffer), (size))
729     #define TPMI_SH_POLICY_Marshal(source, buffer, size)                              \
730         Marshal(TPMI_SH_POLICY_MARSHAL_REF, (source), (buffer), (size))
731     #define TPMI_DH_CONTEXT_Unmarshal(target, buffer, size)                           \
732         Unmarshal(TPMI_DH_CONTEXT_MARSHAL_REF, (target), (buffer), (size))
733     #define TPMI_DH_CONTEXT_Marshal(source, buffer, size)                             \
734         Marshal(TPMI_DH_CONTEXT_MARSHAL_REF, (source), (buffer), (size))
735     #define TPMI_DH_SAVED_Unmarshal(target, buffer, size)                             \
736         Unmarshal(TPMI_DH_SAVED_MARSHAL_REF, (target), (buffer), (size))
737     #define TPMI_DH_SAVED_Marshal(source, buffer, size)                               \
738         Marshal(TPMI_DH_SAVED_MARSHAL_REF, (source), (buffer), (size))
739     #define TPMI_RH_HIERARCHY_Unmarshal(target, buffer, size, flag)                   \
740         Unmarshal(TPMI_RH_HIERARCHY_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),       \
741             (buffer), (size))
742     #define TPMI_RH_HIERARCHY_Marshal(source, buffer, size)                           \
743         Marshal(TPMI_RH_HIERARCHY_MARSHAL_REF, (source), (buffer), (size))
744     #define TPMI_RH_ENABLES_Unmarshal(target, buffer, size, flag)                     \
745         Unmarshal(TPMI_RH_ENABLES_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),         \
746             (buffer), (size))
747     #define TPMI_RH_ENABLES_Marshal(source, buffer, size)                             \
748         Marshal(TPMI_RH_ENABLES_MARSHAL_REF, (source), (buffer), (size))
749     #define TPMI_RH_HIERARCHY_AUTH_Unmarshal(target, buffer, size)                    \
750         Unmarshal(TPMI_RH_HIERARCHY_AUTH_MARSHAL_REF, (target), (buffer), (size))
751     #define TPMI_RH_HIERARCHY_POLICY_Unmarshal(target, buffer, size)                  \
752         Unmarshal(TPMI_RH_HIERARCHY_POLICY_MARSHAL_REF, (target), (buffer), (size))
753     #define TPMI_RH_PLATFORM_Unmarshal(target, buffer, size)                          \
754         Unmarshal(TPMI_RH_PLATFORM_MARSHAL_REF, (target), (buffer), (size))
755     #define TPMI_RH_OWNER_Unmarshal(target, buffer, size, flag)                       \
756         Unmarshal(TPMI_RH_OWNER_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \
757             (size))
758     #define TPMI_RH_ENDORSEMENT_Unmarshal(target, buffer, size, flag)                 \
759         Unmarshal(TPMI_RH_ENDORSEMENT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),     \
760             (buffer), (size))
761     #define TPMI_RH_PROVISION_Unmarshal(target, buffer, size)                         \
762         Unmarshal(TPMI_RH_PROVISION_MARSHAL_REF, (target), (buffer), (size))
763     #define TPMI_RH_CLEAR_Unmarshal(target, buffer, size)                             \
764         Unmarshal(TPMI_RH_CLEAR_MARSHAL_REF, (target), (buffer), (size))
765     #define TPMI_RH_NV_AUTH_Unmarshal(target, buffer, size)                           \
766         Unmarshal(TPMI_RH_NV_AUTH_MARSHAL_REF, (target), (buffer), (size))
767     #define TPMI_RH_LOCKOUT_Unmarshal(target, buffer, size)                           \
768         Unmarshal(TPMI_RH_LOCKOUT_MARSHAL_REF, (target), (buffer), (size))
769     #define TPMI_RH_NV_INDEX_Unmarshal(target, buffer, size)                          \
```

```
770         Unmarshal(TPMI_RH_NV_INDEX_MARSHAL_REF, (target), (buffer), (size))
771     #define TPMI_RH_NV_INDEX_Marshal(source, buffer, size)                           \
772         Marshal(TPMI_RH_NV_INDEX_MARSHAL_REF, (source), (buffer), (size))
773     #define TPMI_RH_AC_Unmarshal(target, buffer, size)                               \
774         Unmarshal(TPMI_RH_AC_MARSHAL_REF, (target), (buffer), (size))
775     #define TPMI_RH_ACT_Unmarshal(target, buffer, size)                              \
776         Unmarshal(TPMI_RH_ACT_MARSHAL_REF, (target), (buffer), (size))
777     #define TPMI_RH_ACT_Marshal(source, buffer, size)                                \
778         Marshal(TPMI_RH_ACT_MARSHAL_REF, (source), (buffer), (size))
779     #define TPMI_ALG_HASH_Unmarshal(target, buffer, size, flag)                      \
780         Unmarshal(TPMI_ALG_HASH_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \
781           (size))
782     #define TPMI_ALG_HASH_Marshal(source, buffer, size)                              \
783         Marshal(TPMI_ALG_HASH_MARSHAL_REF, (source), (buffer), (size))
784     #define TPMI_ALG_ASYM_Unmarshal(target, buffer, size, flag)                      \
785         Unmarshal(TPMI_ALG_ASYM_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \
786           (size))
787     #define TPMI_ALG_ASYM_Marshal(source, buffer, size)                              \
788         Marshal(TPMI_ALG_ASYM_MARSHAL_REF, (source), (buffer), (size))
789     #define TPMI_ALG_SYM_Unmarshal(target, buffer, size, flag)                       \
790         Unmarshal(TPMI_ALG_SYM_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \
791           (size))
792     #define TPMI_ALG_SYM_Marshal(source, buffer, size)                               \
793         Marshal(TPMI_ALG_SYM_MARSHAL_REF, (source), (buffer), (size))
794     #define TPMI_ALG_SYM_OBJECT_Unmarshal(target, buffer, size, flag)                \
795         Unmarshal(TPMI_ALG_SYM_OBJECT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
796           (buffer), (size))
797     #define TPMI_ALG_SYM_OBJECT_Marshal(source, buffer, size)                        \
798         Marshal(TPMI_ALG_SYM_OBJECT_MARSHAL_REF, (source), (buffer), (size))
799     #define TPMI_ALG_SYM_MODE_Unmarshal(target, buffer, size, flag)                  \
800         Unmarshal(TPMI_ALG_SYM_MODE_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),   \
801           (buffer), (size))
802     #define TPMI_ALG_SYM_MODE_Marshal(source, buffer, size)                          \
803         Marshal(TPMI_ALG_SYM_MODE_MARSHAL_REF, (source), (buffer), (size))
804     #define TPMI_ALG_KDF_Unmarshal(target, buffer, size, flag)                       \
805         Unmarshal(TPMI_ALG_KDF_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \
806           (size))
807     #define TPMI_ALG_KDF_Marshal(source, buffer, size)                               \
808         Marshal(TPMI_ALG_KDF_MARSHAL_REF, (source), (buffer), (size))
809     #define TPMI_ALG_SIG_SCHEME_Unmarshal(target, buffer, size, flag)                \
810         Unmarshal(TPMI_ALG_SIG_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
811           (buffer), (size))
812     #define TPMI_ALG_SIG_SCHEME_Marshal(source, buffer, size)                        \
813         Marshal(TPMI_ALG_SIG_SCHEME_MARSHAL_REF, (source), (buffer), (size))
814     #define TPMI_ECC_KEY_EXCHANGE_Unmarshal(target, buffer, size, flag)              \
815         Unmarshal(TPMI_ECC_KEY_EXCHANGE_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
816           (buffer), (size))
817     #define TPMI_ECC_KEY_EXCHANGE_Marshal(source, buffer, size)                      \
818         Marshal(TPMI_ECC_KEY_EXCHANGE_MARSHAL_REF, (source), (buffer), (size))
819     #define TPMI_ST_COMMAND_TAG_Unmarshal(target, buffer, size)                      \
820         Unmarshal(TPMI_ST_COMMAND_TAG_MARSHAL_REF, (target), (buffer), (size))
821     #define TPMI_ST_COMMAND_TAG_Marshal(source, buffer, size)                        \
822         Marshal(TPMI_ST_COMMAND_TAG_MARSHAL_REF, (source), (buffer), (size))
823     #define TPMI_ALG_MAC_SCHEME_Unmarshal(target, buffer, size, flag)                \
824         Unmarshal(TPMI_ALG_MAC_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
825           (buffer), (size))
826     #define TPMI_ALG_MAC_SCHEME_Marshal(source, buffer, size)                        \
827         Marshal(TPMI_ALG_MAC_SCHEME_MARSHAL_REF, (source), (buffer), (size))
828     #define TPMI_ALG_CIPHER_MODE_Unmarshal(target, buffer, size, flag)               \
829         Unmarshal(TPMI_ALG_CIPHER_MODE_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
830           (buffer), (size))
831     #define TPMI_ALG_CIPHER_MODE_Marshal(source, buffer, size)                       \
832         Marshal(TPMI_ALG_CIPHER_MODE_MARSHAL_REF, (source), (buffer), (size))
833     #define TPMS_EMPTY_Unmarshal(target, buffer, size)                               \
834         Unmarshal(TPMS_EMPTY_MARSHAL_REF, (target), (buffer), (size))
835     #define TPMS_EMPTY_Marshal(source, buffer, size)                                 \
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
836         Marshal(TPMS_EMPTY_MARSHAL_REF, (source), (buffer), (size))
837     #define TPMS_ALGORITHM_DESCRIPTION_Marshal(source, buffer, size)                  \
838         Marshal(TPMS_ALGORITHM_DESCRIPTION_MARSHAL_REF, (source), (buffer), (size))
839     #define TPMU_HA_Unmarshal(target, buffer, size, selector)                         \
840         UnmarshalUnion(TPMU_HA_MARSHAL_REF, (target), (buffer), (size), (selector))
841     #define TPMU_HA_Marshal(source, buffer, size, selector)                           \
842         MarshalUnion(TPMU_HA_MARSHAL_REF, (target), (buffer), (size), (selector))
843     #define TPMT_HA_Unmarshal(target, buffer, size, flag)                             \
844         Unmarshal(TPMT_HA_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),     \
845           (size))
846     #define TPMT_HA_Marshal(source, buffer, size)                                     \
847         Marshal(TPMT_HA_MARSHAL_REF, (source), (buffer), (size))
848     #define TPM2B_DIGEST_Unmarshal(target, buffer, size)                              \
849         Unmarshal(TPM2B_DIGEST_MARSHAL_REF, (target), (buffer), (size))
850     #define TPM2B_DIGEST_Marshal(source, buffer, size)                                \
851         Marshal(TPM2B_DIGEST_MARSHAL_REF, (source), (buffer), (size))
852     #define TPM2B_DATA_Unmarshal(target, buffer, size)                                \
853         Unmarshal(TPM2B_DATA_MARSHAL_REF, (target), (buffer), (size))
854     #define TPM2B_DATA_Marshal(source, buffer, size)                                  \
855         Marshal(TPM2B_DATA_MARSHAL_REF, (source), (buffer), (size))
856     #define TPM2B_NONCE_Unmarshal(target, buffer, size)                               \
857         Unmarshal(TPM2B_NONCE_MARSHAL_REF, (target), (buffer), (size))
858     #define TPM2B_NONCE_Marshal(source, buffer, size)                                 \
859         Marshal(TPM2B_NONCE_MARSHAL_REF, (source), (buffer), (size))
860     #define TPM2B_AUTH_Unmarshal(target, buffer, size)                                \
861         Unmarshal(TPM2B_AUTH_MARSHAL_REF, (target), (buffer), (size))
862     #define TPM2B_AUTH_Marshal(source, buffer, size)                                  \
863         Marshal(TPM2B_AUTH_MARSHAL_REF, (source), (buffer), (size))
864     #define TPM2B_OPERAND_Unmarshal(target, buffer, size)                             \
865         Unmarshal(TPM2B_OPERAND_MARSHAL_REF, (target), (buffer), (size))
866     #define TPM2B_OPERAND_Marshal(source, buffer, size)                               \
867         Marshal(TPM2B_OPERAND_MARSHAL_REF, (source), (buffer), (size))
868     #define TPM2B_EVENT_Unmarshal(target, buffer, size)                               \
869         Unmarshal(TPM2B_EVENT_MARSHAL_REF, (target), (buffer), (size))
870     #define TPM2B_EVENT_Marshal(source, buffer, size)                                 \
871         Marshal(TPM2B_EVENT_MARSHAL_REF, (source), (buffer), (size))
872     #define TPM2B_MAX_BUFFER_Unmarshal(target, buffer, size)                          \
873         Unmarshal(TPM2B_MAX_BUFFER_MARSHAL_REF, (target), (buffer), (size))
874     #define TPM2B_MAX_BUFFER_Marshal(source, buffer, size)                            \
875         Marshal(TPM2B_MAX_BUFFER_MARSHAL_REF, (source), (buffer), (size))
876     #define TPM2B_MAX_NV_BUFFER_Unmarshal(target, buffer, size)                       \
877         Unmarshal(TPM2B_MAX_NV_BUFFER_MARSHAL_REF, (target), (buffer), (size))
878     #define TPM2B_MAX_NV_BUFFER_Marshal(source, buffer, size)                         \
879         Marshal(TPM2B_MAX_NV_BUFFER_MARSHAL_REF, (source), (buffer), (size))
880     #define TPM2B_TIMEOUT_Unmarshal(target, buffer, size)                             \
881         Unmarshal(TPM2B_TIMEOUT_MARSHAL_REF, (target), (buffer), (size))
882     #define TPM2B_TIMEOUT_Marshal(source, buffer, size)                               \
883         Marshal(TPM2B_TIMEOUT_MARSHAL_REF, (source), (buffer), (size))
884     #define TPM2B_IV_Unmarshal(target, buffer, size)                                  \
885         Unmarshal(TPM2B_IV_MARSHAL_REF, (target), (buffer), (size))
886     #define TPM2B_IV_Marshal(source, buffer, size)                                    \
887         Marshal(TPM2B_IV_MARSHAL_REF, (source), (buffer), (size))
888     #define TPM2B_NAME_Unmarshal(target, buffer, size)                                \
889         Unmarshal(TPM2B_NAME_MARSHAL_REF, (target), (buffer), (size))
890     #define TPM2B_NAME_Marshal(source, buffer, size)                                  \
891         Marshal(TPM2B_NAME_MARSHAL_REF, (source), (buffer), (size))
892     #define TPMS_PCR_SELECT_Unmarshal(target, buffer, size)                           \
893         Unmarshal(TPMS_PCR_SELECT_MARSHAL_REF, (target), (buffer), (size))
894     #define TPMS_PCR_SELECT_Marshal(source, buffer, size)                             \
895         Marshal(TPMS_PCR_SELECT_MARSHAL_REF, (source), (buffer), (size))
896     #define TPMS_PCR_SELECTION_Unmarshal(target, buffer, size)                        \
897         Unmarshal(TPMS_PCR_SELECTION_MARSHAL_REF, (target), (buffer), (size))
898     #define TPMS_PCR_SELECTION_Marshal(source, buffer, size)                          \
899         Marshal(TPMS_PCR_SELECTION_MARSHAL_REF, (source), (buffer), (size))
900     #define TPMT_TK_CREATION_Unmarshal(target, buffer, size)                          \
901         Unmarshal(TPMT_TK_CREATION_MARSHAL_REF, (target), (buffer), (size))
```

 Copyright © TCG 2006-2020

```
902     #define TPMT_TK_CREATION_Marshal(source, buffer, size)                        \
903         Marshal(TPMT_TK_CREATION_MARSHAL_REF, (source), (buffer), (size))
904     #define TPMT_TK_VERIFIED_Unmarshal(target, buffer, size)                      \
905         Unmarshal(TPMT_TK_VERIFIED_MARSHAL_REF, (target), (buffer), (size))
906     #define TPMT_TK_VERIFIED_Marshal(source, buffer, size)                        \
907         Marshal(TPMT_TK_VERIFIED_MARSHAL_REF, (source), (buffer), (size))
908     #define TPMT_TK_AUTH_Unmarshal(target, buffer, size)                          \
909         Unmarshal(TPMT_TK_AUTH_MARSHAL_REF, (target), (buffer), (size))
910     #define TPMT_TK_AUTH_Marshal(source, buffer, size)                            \
911         Marshal(TPMT_TK_AUTH_MARSHAL_REF, (source), (buffer), (size))
912     #define TPMT_TK_HASHCHECK_Unmarshal(target, buffer, size)                     \
913         Unmarshal(TPMT_TK_HASHCHECK_MARSHAL_REF, (target), (buffer), (size))
914     #define TPMT_TK_HASHCHECK_Marshal(source, buffer, size)                       \
915         Marshal(TPMT_TK_HASHCHECK_MARSHAL_REF, (source), (buffer), (size))
916     #define TPMS_ALG_PROPERTY_Marshal(source, buffer, size)                       \
917         Marshal(TPMS_ALG_PROPERTY_MARSHAL_REF, (source), (buffer), (size))
918     #define TPMS_TAGGED_PROPERTY_Marshal(source, buffer, size)                    \
919         Marshal(TPMS_TAGGED_PROPERTY_MARSHAL_REF, (source), (buffer), (size))
920     #define TPMS_TAGGED_PCR_SELECT_Marshal(source, buffer, size)                  \
921         Marshal(TPMS_TAGGED_PCR_SELECT_MARSHAL_REF, (source), (buffer), (size))
922     #define TPMS_TAGGED_POLICY_Marshal(source, buffer, size)                      \
923         Marshal(TPMS_TAGGED_POLICY_MARSHAL_REF, (source), (buffer), (size))
924     #define TPMS_ACT_DATA_Marshal(source, buffer, size)                           \
925         Marshal(TPMS_ACT_DATA_MARSHAL_REF, (source), (buffer), (size))
926     #define TPML_CC_Unmarshal(target, buffer, size)                               \
927         Unmarshal(TPML_CC_MARSHAL_REF, (target), (buffer), (size))
928     #define TPML_CC_Marshal(source, buffer, size)                                 \
929         Marshal(TPML_CC_MARSHAL_REF, (source), (buffer), (size))
930     #define TPML_CCA_Marshal(source, buffer, size)                                \
931         Marshal(TPML_CCA_MARSHAL_REF, (source), (buffer), (size))
932     #define TPML_ALG_Unmarshal(target, buffer, size)                              \
933         Unmarshal(TPML_ALG_MARSHAL_REF, (target), (buffer), (size))
934     #define TPML_ALG_Marshal(source, buffer, size)                                \
935         Marshal(TPML_ALG_MARSHAL_REF, (source), (buffer), (size))
936     #define TPML_HANDLE_Marshal(source, buffer, size)                             \
937         Marshal(TPML_HANDLE_MARSHAL_REF, (source), (buffer), (size))
938     #define TPML_DIGEST_Unmarshal(target, buffer, size)                           \
939         Unmarshal(TPML_DIGEST_MARSHAL_REF, (target), (buffer), (size))
940     #define TPML_DIGEST_Marshal(source, buffer, size)                             \
941         Marshal(TPML_DIGEST_MARSHAL_REF, (source), (buffer), (size))
942     #define TPML_DIGEST_VALUES_Unmarshal(target, buffer, size)                    \
943         Unmarshal(TPML_DIGEST_VALUES_MARSHAL_REF, (target), (buffer), (size))
944     #define TPML_DIGEST_VALUES_Marshal(source, buffer, size)                      \
945         Marshal(TPML_DIGEST_VALUES_MARSHAL_REF, (source), (buffer), (size))
946     #define TPML_PCR_SELECTION_Unmarshal(target, buffer, size)                    \
947         Unmarshal(TPML_PCR_SELECTION_MARSHAL_REF, (target), (buffer), (size))
948     #define TPML_PCR_SELECTION_Marshal(source, buffer, size)                      \
949         Marshal(TPML_PCR_SELECTION_MARSHAL_REF, (source), (buffer), (size))
950     #define TPML_ALG_PROPERTY_Marshal(source, buffer, size)                       \
951         Marshal(TPML_ALG_PROPERTY_MARSHAL_REF, (source), (buffer), (size))
952     #define TPML_TAGGED_TPM_PROPERTY_Marshal(source, buffer, size)                \
953         Marshal(TPML_TAGGED_TPM_PROPERTY_MARSHAL_REF, (source), (buffer), (size))
954     #define TPML_TAGGED_PCR_PROPERTY_Marshal(source, buffer, size)                \
955         Marshal(TPML_TAGGED_PCR_PROPERTY_MARSHAL_REF, (source), (buffer), (size))
956     #define TPML_ECC_CURVE_Marshal(source, buffer, size)                          \
957         Marshal(TPML_ECC_CURVE_MARSHAL_REF, (source), (buffer), (size))
958     #define TPML_TAGGED_POLICY_Marshal(source, buffer, size)                      \
959         Marshal(TPML_TAGGED_POLICY_MARSHAL_REF, (source), (buffer), (size))
960     #define TPML_ACT_DATA_Marshal(source, buffer, size)                           \
961         Marshal(TPML_ACT_DATA_MARSHAL_REF, (source), (buffer), (size))
962     #define TPMU_CAPABILITIES_Marshal(source, buffer, size, selector)             \
963         MarshalUnion(TPMU_CAPABILITIES_MARSHAL_REF, (target), (buffer), (size),   \
964           (selector))
965     #define TPMS_CAPABILITY_DATA_Marshal(source, buffer, size)                    \
966         Marshal(TPMS_CAPABILITY_DATA_MARSHAL_REF, (source), (buffer), (size))
967     #define TPMS_CLOCK_INFO_Unmarshal(target, buffer, size)                       \
```

```
968       Unmarshal(TPMS_CLOCK_INFO_MARSHAL_REF, (target), (buffer), (size))
969   #define TPMS_CLOCK_INFO_Marshal(source, buffer, size)                              \
970       Marshal(TPMS_CLOCK_INFO_MARSHAL_REF, (source), (buffer), (size))
971   #define TPMS_TIME_INFO_Unmarshal(target, buffer, size)                             \
972       Unmarshal(TPMS_TIME_INFO_MARSHAL_REF, (target), (buffer), (size))
973   #define TPMS_TIME_INFO_Marshal(source, buffer, size)                               \
974       Marshal(TPMS_TIME_INFO_MARSHAL_REF, (source), (buffer), (size))
975   #define TPMS_TIME_ATTEST_INFO_Marshal(source, buffer, size)                        \
976       Marshal(TPMS_TIME_ATTEST_INFO_MARSHAL_REF, (source), (buffer), (size))
977   #define TPMS_CERTIFY_INFO_Marshal(source, buffer, size)                            \
978       Marshal(TPMS_CERTIFY_INFO_MARSHAL_REF, (source), (buffer), (size))
979   #define TPMS_QUOTE_INFO_Marshal(source, buffer, size)                              \
980       Marshal(TPMS_QUOTE_INFO_MARSHAL_REF, (source), (buffer), (size))
981   #define TPMS_COMMAND_AUDIT_INFO_Marshal(source, buffer, size)                      \
982       Marshal(TPMS_COMMAND_AUDIT_INFO_MARSHAL_REF, (source), (buffer), (size))
983   #define TPMS_SESSION_AUDIT_INFO_Marshal(source, buffer, size)                      \
984       Marshal(TPMS_SESSION_AUDIT_INFO_MARSHAL_REF, (source), (buffer), (size))
985   #define TPMS_CREATION_INFO_Marshal(source, buffer, size)                           \
986       Marshal(TPMS_CREATION_INFO_MARSHAL_REF, (source), (buffer), (size))
987   #define TPMS_NV_CERTIFY_INFO_Marshal(source, buffer, size)                         \
988       Marshal(TPMS_NV_CERTIFY_INFO_MARSHAL_REF, (source), (buffer), (size))
989   #define TPMS_NV_DIGEST_CERTIFY_INFO_Marshal(source, buffer, size)                  \
990       Marshal(TPMS_NV_DIGEST_CERTIFY_INFO_MARSHAL_REF, (source), (buffer), (size))
991   #define TPMI_ST_ATTEST_Marshal(source, buffer, size)                               \
992       Marshal(TPMI_ST_ATTEST_MARSHAL_REF, (source), (buffer), (size))
993   #define TPMU_ATTEST_Marshal(source, buffer, size, selector)                        \
994       MarshalUnion(TPMU_ATTEST_MARSHAL_REF, (target), (buffer), (size), (selector))
995   #define TPMS_ATTEST_Marshal(source, buffer, size)                                  \
996       Marshal(TPMS_ATTEST_MARSHAL_REF, (source), (buffer), (size))
997   #define TPM2B_ATTEST_Marshal(source, buffer, size)                                 \
998       Marshal(TPM2B_ATTEST_MARSHAL_REF, (source), (buffer), (size))
999   #define TPMS_AUTH_COMMAND_Unmarshal(target, buffer, size)                          \
1000      Unmarshal(TPMS_AUTH_COMMAND_MARSHAL_REF, (target), (buffer), (size))
1001  #define TPMS_AUTH_RESPONSE_Marshal(source, buffer, size)                           \
1002      Marshal(TPMS_AUTH_RESPONSE_MARSHAL_REF, (source), (buffer), (size))
1003  #define TPMI_TDES_KEY_BITS_Unmarshal(target, buffer, size)                         \
1004      Unmarshal(TPMI_TDES_KEY_BITS_MARSHAL_REF, (target), (buffer), (size))
1005  #define TPMI_TDES_KEY_BITS_Marshal(source, buffer, size)                           \
1006      Marshal(TPMI_TDES_KEY_BITS_MARSHAL_REF, (source), (buffer), (size))
1007  #define TPMI_AES_KEY_BITS_Unmarshal(target, buffer, size)                          \
1008      Unmarshal(TPMI_AES_KEY_BITS_MARSHAL_REF, (target), (buffer), (size))
1009  #define TPMI_AES_KEY_BITS_Marshal(source, buffer, size)                            \
1010      Marshal(TPMI_AES_KEY_BITS_MARSHAL_REF, (source), (buffer), (size))
1011  #define TPMI_SM4_KEY_BITS_Unmarshal(target, buffer, size)                          \
1012      Unmarshal(TPMI_SM4_KEY_BITS_MARSHAL_REF, (target), (buffer), (size))
1013  #define TPMI_SM4_KEY_BITS_Marshal(source, buffer, size)                            \
1014      Marshal(TPMI_SM4_KEY_BITS_MARSHAL_REF, (source), (buffer), (size))
1015  #define TPMI_CAMELLIA_KEY_BITS_Unmarshal(target, buffer, size)                     \
1016      Unmarshal(TPMI_CAMELLIA_KEY_BITS_MARSHAL_REF, (target), (buffer), (size))
1017  #define TPMI_CAMELLIA_KEY_BITS_Marshal(source, buffer, size)                       \
1018      Marshal(TPMI_CAMELLIA_KEY_BITS_MARSHAL_REF, (source), (buffer), (size))
1019  #define TPMU_SYM_KEY_BITS_Unmarshal(target, buffer, size, selector)                \
1020      UnmarshalUnion(TPMU_SYM_KEY_BITS_MARSHAL_REF, (target), (buffer), (size),     \
1021        (selector))
1022  #define TPMU_SYM_KEY_BITS_Marshal(source, buffer, size, selector)                  \
1023      MarshalUnion(TPMU_SYM_KEY_BITS_MARSHAL_REF, (target), (buffer), (size),       \
1024        (selector))
1025  #define TPMU_SYM_MODE_Unmarshal(target, buffer, size, selector)                    \
1026      UnmarshalUnion(TPMU_SYM_MODE_MARSHAL_REF, (target), (buffer), (size),         \
1027        (selector))
1028  #define TPMU_SYM_MODE_Marshal(source, buffer, size, selector)                      \
1029      MarshalUnion(TPMU_SYM_MODE_MARSHAL_REF, (target), (buffer), (size), (selector))
1030  #define TPMT_SYM_DEF_Unmarshal(target, buffer, size, flag)                         \
1031      Unmarshal(TPMT_SYM_DEF_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \
1032        (size))
1033  #define TPMT_SYM_DEF_Marshal(source, buffer, size)                                 \
```

```
1034        Marshal(TPMT_SYM_DEF_MARSHAL_REF, (source), (buffer), (size))
1035    #define TPMT_SYM_DEF_OBJECT_Unmarshal(target, buffer, size, flag)              \
1036        Unmarshal(TPMT_SYM_DEF_OBJECT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),    \
1037          (buffer), (size))
1038    #define TPMT_SYM_DEF_OBJECT_Marshal(source, buffer, size)                      \
1039        Marshal(TPMT_SYM_DEF_OBJECT_MARSHAL_REF, (source), (buffer), (size))
1040    #define TPM2B_SYM_KEY_Unmarshal(target, buffer, size)                          \
1041        Unmarshal(TPM2B_SYM_KEY_MARSHAL_REF, (target), (buffer), (size))
1042    #define TPM2B_SYM_KEY_Marshal(source, buffer, size)                            \
1043        Marshal(TPM2B_SYM_KEY_MARSHAL_REF, (source), (buffer), (size))
1044    #define TPMS_SYMCIPHER_PARMS_Unmarshal(target, buffer, size)                   \
1045        Unmarshal(TPMS_SYMCIPHER_PARMS_MARSHAL_REF, (target), (buffer), (size))
1046    #define TPMS_SYMCIPHER_PARMS_Marshal(source, buffer, size)                     \
1047        Marshal(TPMS_SYMCIPHER_PARMS_MARSHAL_REF, (source), (buffer), (size))
1048    #define TPM2B_LABEL_Unmarshal(target, buffer, size)                            \
1049        Unmarshal(TPM2B_LABEL_MARSHAL_REF, (target), (buffer), (size))
1050    #define TPM2B_LABEL_Marshal(source, buffer, size)                              \
1051        Marshal(TPM2B_LABEL_MARSHAL_REF, (source), (buffer), (size))
1052    #define TPMS_DERIVE_Unmarshal(target, buffer, size)                            \
1053        Unmarshal(TPMS_DERIVE_MARSHAL_REF, (target), (buffer), (size))
1054    #define TPMS_DERIVE_Marshal(source, buffer, size)                              \
1055        Marshal(TPMS_DERIVE_MARSHAL_REF, (source), (buffer), (size))
1056    #define TPM2B_DERIVE_Unmarshal(target, buffer, size)                           \
1057        Unmarshal(TPM2B_DERIVE_MARSHAL_REF, (target), (buffer), (size))
1058    #define TPM2B_DERIVE_Marshal(source, buffer, size)                             \
1059        Marshal(TPM2B_DERIVE_MARSHAL_REF, (source), (buffer), (size))
1060    #define TPM2B_SENSITIVE_DATA_Unmarshal(target, buffer, size)                   \
1061        Unmarshal(TPM2B_SENSITIVE_DATA_MARSHAL_REF, (target), (buffer), (size))
1062    #define TPM2B_SENSITIVE_DATA_Marshal(source, buffer, size)                     \
1063        Marshal(TPM2B_SENSITIVE_DATA_MARSHAL_REF, (source), (buffer), (size))
1064    #define TPMS_SENSITIVE_CREATE_Unmarshal(target, buffer, size)                  \
1065        Unmarshal(TPMS_SENSITIVE_CREATE_MARSHAL_REF, (target), (buffer), (size))
1066    #define TPM2B_SENSITIVE_CREATE_Unmarshal(target, buffer, size)                 \
1067        Unmarshal(TPM2B_SENSITIVE_CREATE_MARSHAL_REF, (target), (buffer), (size))
1068    #define TPMS_SCHEME_HASH_Unmarshal(target, buffer, size)                       \
1069        Unmarshal(TPMS_SCHEME_HASH_MARSHAL_REF, (target), (buffer), (size))
1070    #define TPMS_SCHEME_HASH_Marshal(source, buffer, size)                         \
1071        Marshal(TPMS_SCHEME_HASH_MARSHAL_REF, (source), (buffer), (size))
1072    #define TPMS_SCHEME_ECDAA_Unmarshal(target, buffer, size)                      \
1073        Unmarshal(TPMS_SCHEME_ECDAA_MARSHAL_REF, (target), (buffer), (size))
1074    #define TPMS_SCHEME_ECDAA_Marshal(source, buffer, size)                        \
1075        Marshal(TPMS_SCHEME_ECDAA_MARSHAL_REF, (source), (buffer), (size))
1076    #define TPMI_ALG_KEYEDHASH_SCHEME_Unmarshal(target, buffer, size, flag)        \
1077        Unmarshal(TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0),    \
1078          (target), (buffer), (size))
1079    #define TPMI_ALG_KEYEDHASH_SCHEME_Marshal(source, buffer, size)                \
1080        Marshal(TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1081    #define TPMS_SCHEME_HMAC_Unmarshal(target, buffer, size)                       \
1082        Unmarshal(TPMS_SCHEME_HMAC_MARSHAL_REF, (target), (buffer), (size))
1083    #define TPMS_SCHEME_HMAC_Marshal(source, buffer, size)                         \
1084        Marshal(TPMS_SCHEME_HMAC_MARSHAL_REF, (source), (buffer), (size))
1085    #define TPMS_SCHEME_XOR_Unmarshal(target, buffer, size)                        \
1086        Unmarshal(TPMS_SCHEME_XOR_MARSHAL_REF, (target), (buffer), (size))
1087    #define TPMS_SCHEME_XOR_Marshal(source, buffer, size)                          \
1088        Marshal(TPMS_SCHEME_XOR_MARSHAL_REF, (source), (buffer), (size))
1089    #define TPMU_SCHEME_KEYEDHASH_Unmarshal(target, buffer, size, selector)        \
1090        UnmarshalUnion(TPMU_SCHEME_KEYEDHASH_MARSHAL_REF, (target), (buffer), (size),   \
1091          (selector))
1092    #define TPMU_SCHEME_KEYEDHASH_Marshal(source, buffer, size, selector)          \
1093        MarshalUnion(TPMU_SCHEME_KEYEDHASH_MARSHAL_REF, (target), (buffer), (size),     \
1094          (selector))
1095    #define TPMT_KEYEDHASH_SCHEME_Unmarshal(target, buffer, size, flag)            \
1096        Unmarshal(TPMT_KEYEDHASH_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),  \
1097          (buffer), (size))
1098    #define TPMT_KEYEDHASH_SCHEME_Marshal(source, buffer, size)                    \
1099        Marshal(TPMT_KEYEDHASH_SCHEME_MARSHAL_REF, (source), (buffer), (size))
```

```
1100 #define TPMS_SIG_SCHEME_RSASSA_Unmarshal(target, buffer, size)                  \
1101     Unmarshal(TPMS_SIG_SCHEME_RSASSA_MARSHAL_REF, (target), (buffer), (size))
1102 #define TPMS_SIG_SCHEME_RSASSA_Marshal(source, buffer, size)                    \
1103     Marshal(TPMS_SIG_SCHEME_RSASSA_MARSHAL_REF, (source), (buffer), (size))
1104 #define TPMS_SIG_SCHEME_RSAPSS_Unmarshal(target, buffer, size)                  \
1105     Unmarshal(TPMS_SIG_SCHEME_RSAPSS_MARSHAL_REF, (target), (buffer), (size))
1106 #define TPMS_SIG_SCHEME_RSAPSS_Marshal(source, buffer, size)                    \
1107     Marshal(TPMS_SIG_SCHEME_RSAPSS_MARSHAL_REF, (source), (buffer), (size))
1108 #define TPMS_SIG_SCHEME_ECDSA_Unmarshal(target, buffer, size)                   \
1109     Unmarshal(TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF, (target), (buffer), (size))
1110 #define TPMS_SIG_SCHEME_ECDSA_Marshal(source, buffer, size)                     \
1111     Marshal(TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF, (source), (buffer), (size))
1112 #define TPMS_SIG_SCHEME_SM2_Unmarshal(target, buffer, size)                     \
1113     Unmarshal(TPMS_SIG_SCHEME_SM2_MARSHAL_REF, (target), (buffer), (size))
1114 #define TPMS_SIG_SCHEME_SM2_Marshal(source, buffer, size)                       \
1115     Marshal(TPMS_SIG_SCHEME_SM2_MARSHAL_REF, (source), (buffer), (size))
1116 #define TPMS_SIG_SCHEME_ECSCHNORR_Unmarshal(target, buffer, size)               \
1117     Unmarshal(TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_REF, (target), (buffer), (size))
1118 #define TPMS_SIG_SCHEME_ECSCHNORR_Marshal(source, buffer, size)                 \
1119     Marshal(TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_REF, (source), (buffer), (size))
1120 #define TPMS_SIG_SCHEME_ECDAA_Unmarshal(target, buffer, size)                   \
1121     Unmarshal(TPMS_SIG_SCHEME_ECDAA_MARSHAL_REF, (target), (buffer), (size))
1122 #define TPMS_SIG_SCHEME_ECDAA_Marshal(source, buffer, size)                     \
1123     Marshal(TPMS_SIG_SCHEME_ECDAA_MARSHAL_REF, (source), (buffer), (size))
1124 #define TPMU_SIG_SCHEME_Unmarshal(target, buffer, size, selector)               \
1125     UnmarshalUnion(TPMU_SIG_SCHEME_MARSHAL_REF, (target), (buffer), (size),     \
1126       (selector))
1127 #define TPMU_SIG_SCHEME_Marshal(source, buffer, size, selector)                 \
1128     MarshalUnion(TPMU_SIG_SCHEME_MARSHAL_REF, (target), (buffer), (size),       \
1129       (selector))
1130 #define TPMT_SIG_SCHEME_Unmarshal(target, buffer, size, flag)                   \
1131     Unmarshal(TPMT_SIG_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),     \
1132       (buffer), (size))
1133 #define TPMT_SIG_SCHEME_Marshal(source, buffer, size)                           \
1134     Marshal(TPMT_SIG_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1135 #define TPMS_ENC_SCHEME_OAEP_Unmarshal(target, buffer, size)                    \
1136     Unmarshal(TPMS_ENC_SCHEME_OAEP_MARSHAL_REF, (target), (buffer), (size))
1137 #define TPMS_ENC_SCHEME_OAEP_Marshal(source, buffer, size)                      \
1138     Marshal(TPMS_ENC_SCHEME_OAEP_MARSHAL_REF, (source), (buffer), (size))
1139 #define TPMS_ENC_SCHEME_RSAES_Unmarshal(target, buffer, size)                   \
1140     Unmarshal(TPMS_ENC_SCHEME_RSAES_MARSHAL_REF, (target), (buffer), (size))
1141 #define TPMS_ENC_SCHEME_RSAES_Marshal(source, buffer, size)                     \
1142     Marshal(TPMS_ENC_SCHEME_RSAES_MARSHAL_REF, (source), (buffer), (size))
1143 #define TPMS_KEY_SCHEME_ECDH_Unmarshal(target, buffer, size)                    \
1144     Unmarshal(TPMS_KEY_SCHEME_ECDH_MARSHAL_REF, (target), (buffer), (size))
1145 #define TPMS_KEY_SCHEME_ECDH_Marshal(source, buffer, size)                      \
1146     Marshal(TPMS_KEY_SCHEME_ECDH_MARSHAL_REF, (source), (buffer), (size))
1147 #define TPMS_KEY_SCHEME_ECMQV_Unmarshal(target, buffer, size)                   \
1148     Unmarshal(TPMS_KEY_SCHEME_ECMQV_MARSHAL_REF, (target), (buffer), (size))
1149 #define TPMS_KEY_SCHEME_ECMQV_Marshal(source, buffer, size)                     \
1150     Marshal(TPMS_KEY_SCHEME_ECMQV_MARSHAL_REF, (source), (buffer), (size))
1151 #define TPMS_SCHEME_MGF1_Unmarshal(target, buffer, size)                        \
1152     Unmarshal(TPMS_SCHEME_MGF1_MARSHAL_REF, (target), (buffer), (size))
1153 #define TPMS_SCHEME_MGF1_Marshal(source, buffer, size)                          \
1154     Marshal(TPMS_SCHEME_MGF1_MARSHAL_REF, (source), (buffer), (size))
1155 #define TPMS_SCHEME_KDF1_SP800_56A_Unmarshal(target, buffer, size)              \
1156     Unmarshal(TPMS_SCHEME_KDF1_SP800_56A_MARSHAL_REF, (target), (buffer), (size))
1157 #define TPMS_SCHEME_KDF1_SP800_56A_Marshal(source, buffer, size)                \
1158     Marshal(TPMS_SCHEME_KDF1_SP800_56A_MARSHAL_REF, (source), (buffer), (size))
1159 #define TPMS_SCHEME_KDF2_Unmarshal(target, buffer, size)                        \
1160     Unmarshal(TPMS_SCHEME_KDF2_MARSHAL_REF, (target), (buffer), (size))
1161 #define TPMS_SCHEME_KDF2_Marshal(source, buffer, size)                          \
1162     Marshal(TPMS_SCHEME_KDF2_MARSHAL_REF, (source), (buffer), (size))
1163 #define TPMS_SCHEME_KDF1_SP800_108_Unmarshal(target, buffer, size)              \
1164     Unmarshal(TPMS_SCHEME_KDF1_SP800_108_MARSHAL_REF, (target), (buffer), (size))
1165 #define TPMS_SCHEME_KDF1_SP800_108_Marshal(source, buffer, size)                \
```

```
1166        Marshal(TPMS_SCHEME_KDF1_SP800_108_MARSHAL_REF, (source), (buffer), (size))
1167    #define TPMU_KDF_SCHEME_Unmarshal(target, buffer, size, selector)           \
1168        UnmarshalUnion(TPMU_KDF_SCHEME_MARSHAL_REF, (target), (buffer), (size),   \
1169          (selector))
1170    #define TPMU_KDF_SCHEME_Marshal(source, buffer, size, selector)             \
1171        MarshalUnion(TPMU_KDF_SCHEME_MARSHAL_REF, (target), (buffer), (size),     \
1172          (selector))
1173    #define TPMT_KDF_SCHEME_Unmarshal(target, buffer, size, flag)               \
1174        Unmarshal(TPMT_KDF_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),   \
1175          (buffer), (size))
1176    #define TPMT_KDF_SCHEME_Marshal(source, buffer, size)                       \
1177        Marshal(TPMT_KDF_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1178    #define TPMI_ALG_ASYM_SCHEME_Unmarshal(target, buffer, size, flag)          \
1179        Unmarshal(TPMI_ALG_ASYM_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),    \
1180          (buffer), (size))
1181    #define TPMI_ALG_ASYM_SCHEME_Marshal(source, buffer, size)                  \
1182        Marshal(TPMI_ALG_ASYM_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1183    #define TPMU_ASYM_SCHEME_Unmarshal(target, buffer, size, selector)          \
1184        UnmarshalUnion(TPMU_ASYM_SCHEME_MARSHAL_REF, (target), (buffer), (size),  \
1185          (selector))
1186    #define TPMU_ASYM_SCHEME_Marshal(source, buffer, size, selector)            \
1187        MarshalUnion(TPMU_ASYM_SCHEME_MARSHAL_REF, (target), (buffer), (size),    \
1188          (selector))
1189    #define TPMI_ALG_RSA_SCHEME_Unmarshal(target, buffer, size, flag)           \
1190        Unmarshal(TPMI_ALG_RSA_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),     \
1191          (buffer), (size))
1192    #define TPMI_ALG_RSA_SCHEME_Marshal(source, buffer, size)                   \
1193        Marshal(TPMI_ALG_RSA_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1194    #define TPMT_RSA_SCHEME_Unmarshal(target, buffer, size, flag)               \
1195        Unmarshal(TPMT_RSA_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),   \
1196          (buffer), (size))
1197    #define TPMT_RSA_SCHEME_Marshal(source, buffer, size)                       \
1198        Marshal(TPMT_RSA_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1199    #define TPMI_ALG_RSA_DECRYPT_Unmarshal(target, buffer, size, flag)          \
1200        Unmarshal(TPMI_ALG_RSA_DECRYPT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),    \
1201          (buffer), (size))
1202    #define TPMI_ALG_RSA_DECRYPT_Marshal(source, buffer, size)                  \
1203        Marshal(TPMI_ALG_RSA_DECRYPT_MARSHAL_REF, (source), (buffer), (size))
1204    #define TPMT_RSA_DECRYPT_Unmarshal(target, buffer, size, flag)              \
1205        Unmarshal(TPMT_RSA_DECRYPT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),  \
1206          (buffer), (size))
1207    #define TPMT_RSA_DECRYPT_Marshal(source, buffer, size)                      \
1208        Marshal(TPMT_RSA_DECRYPT_MARSHAL_REF, (source), (buffer), (size))
1209    #define TPM2B_PUBLIC_KEY_RSA_Unmarshal(target, buffer, size)                \
1210        Unmarshal(TPM2B_PUBLIC_KEY_RSA_MARSHAL_REF, (target), (buffer), (size))
1211    #define TPM2B_PUBLIC_KEY_RSA_Marshal(source, buffer, size)                  \
1212        Marshal(TPM2B_PUBLIC_KEY_RSA_MARSHAL_REF, (source), (buffer), (size))
1213    #define TPMI_RSA_KEY_BITS_Unmarshal(target, buffer, size)                   \
1214        Unmarshal(TPMI_RSA_KEY_BITS_MARSHAL_REF, (target), (buffer), (size))
1215    #define TPMI_RSA_KEY_BITS_Marshal(source, buffer, size)                     \
1216        Marshal(TPMI_RSA_KEY_BITS_MARSHAL_REF, (source), (buffer), (size))
1217    #define TPM2B_PRIVATE_KEY_RSA_Unmarshal(target, buffer, size)               \
1218        Unmarshal(TPM2B_PRIVATE_KEY_RSA_MARSHAL_REF, (target), (buffer), (size))
1219    #define TPM2B_PRIVATE_KEY_RSA_Marshal(source, buffer, size)                 \
1220        Marshal(TPM2B_PRIVATE_KEY_RSA_MARSHAL_REF, (source), (buffer), (size))
1221    #define TPM2B_ECC_PARAMETER_Unmarshal(target, buffer, size)                 \
1222        Unmarshal(TPM2B_ECC_PARAMETER_MARSHAL_REF, (target), (buffer), (size))
1223    #define TPM2B_ECC_PARAMETER_Marshal(source, buffer, size)                   \
1224        Marshal(TPM2B_ECC_PARAMETER_MARSHAL_REF, (source), (buffer), (size))
1225    #define TPMS_ECC_POINT_Unmarshal(target, buffer, size)                      \
1226        Unmarshal(TPMS_ECC_POINT_MARSHAL_REF, (target), (buffer), (size))
1227    #define TPMS_ECC_POINT_Marshal(source, buffer, size)                        \
1228        Marshal(TPMS_ECC_POINT_MARSHAL_REF, (source), (buffer), (size))
1229    #define TPM2B_ECC_POINT_Unmarshal(target, buffer, size)                     \
1230        Unmarshal(TPM2B_ECC_POINT_MARSHAL_REF, (target), (buffer), (size))
1231    #define TPM2B_ECC_POINT_Marshal(source, buffer, size)                       \
```

 Copyright © TCG 2006-2020

```
1232        Marshal(TPM2B_ECC_POINT_MARSHAL_REF, (source), (buffer), (size))
1233    #define TPMI_ALG_ECC_SCHEME_Unmarshal(target, buffer, size, flag)              \
1234        Unmarshal(TPMI_ALG_ECC_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),    \
1235          (buffer), (size))
1236    #define TPMI_ALG_ECC_SCHEME_Marshal(source, buffer, size)                      \
1237        Marshal(TPMI_ALG_ECC_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1238    #define TPMI_ECC_CURVE_Unmarshal(target, buffer, size)                         \
1239        Unmarshal(TPMI_ECC_CURVE_MARSHAL_REF, (target), (buffer), (size))
1240    #define TPMI_ECC_CURVE_Marshal(source, buffer, size)                           \
1241        Marshal(TPMI_ECC_CURVE_MARSHAL_REF, (source), (buffer), (size))
1242    #define TPMT_ECC_SCHEME_Unmarshal(target, buffer, size, flag)                  \
1243        Unmarshal(TPMT_ECC_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),        \
1244          (buffer), (size))
1245    #define TPMT_ECC_SCHEME_Marshal(source, buffer, size)                          \
1246        Marshal(TPMT_ECC_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1247    #define TPMS_ALGORITHM_DETAIL_ECC_Marshal(source, buffer, size)                \
1248        Marshal(TPMS_ALGORITHM_DETAIL_ECC_MARSHAL_REF, (source), (buffer), (size))
1249    #define TPMS_SIGNATURE_RSA_Unmarshal(target, buffer, size)                     \
1250        Unmarshal(TPMS_SIGNATURE_RSA_MARSHAL_REF, (target), (buffer), (size))
1251    #define TPMS_SIGNATURE_RSA_Marshal(source, buffer, size)                       \
1252        Marshal(TPMS_SIGNATURE_RSA_MARSHAL_REF, (source), (buffer), (size))
1253    #define TPMS_SIGNATURE_RSASSA_Unmarshal(target, buffer, size)                  \
1254        Unmarshal(TPMS_SIGNATURE_RSASSA_MARSHAL_REF, (target), (buffer), (size))
1255    #define TPMS_SIGNATURE_RSASSA_Marshal(source, buffer, size)                    \
1256        Marshal(TPMS_SIGNATURE_RSASSA_MARSHAL_REF, (source), (buffer), (size))
1257    #define TPMS_SIGNATURE_RSAPSS_Unmarshal(target, buffer, size)                  \
1258        Unmarshal(TPMS_SIGNATURE_RSAPSS_MARSHAL_REF, (target), (buffer), (size))
1259    #define TPMS_SIGNATURE_RSAPSS_Marshal(source, buffer, size)                    \
1260        Marshal(TPMS_SIGNATURE_RSAPSS_MARSHAL_REF, (source), (buffer), (size))
1261    #define TPMS_SIGNATURE_ECC_Unmarshal(target, buffer, size)                     \
1262        Unmarshal(TPMS_SIGNATURE_ECC_MARSHAL_REF, (target), (buffer), (size))
1263    #define TPMS_SIGNATURE_ECC_Marshal(source, buffer, size)                       \
1264        Marshal(TPMS_SIGNATURE_ECC_MARSHAL_REF, (source), (buffer), (size))
1265    #define TPMS_SIGNATURE_ECDAA_Unmarshal(target, buffer, size)                   \
1266        Unmarshal(TPMS_SIGNATURE_ECDAA_MARSHAL_REF, (target), (buffer), (size))
1267    #define TPMS_SIGNATURE_ECDAA_Marshal(source, buffer, size)                     \
1268        Marshal(TPMS_SIGNATURE_ECDAA_MARSHAL_REF, (source), (buffer), (size))
1269    #define TPMS_SIGNATURE_ECDSA_Unmarshal(target, buffer, size)                   \
1270        Unmarshal(TPMS_SIGNATURE_ECDSA_MARSHAL_REF, (target), (buffer), (size))
1271    #define TPMS_SIGNATURE_ECDSA_Marshal(source, buffer, size)                     \
1272        Marshal(TPMS_SIGNATURE_ECDSA_MARSHAL_REF, (source), (buffer), (size))
1273    #define TPMS_SIGNATURE_SM2_Unmarshal(target, buffer, size)                     \
1274        Unmarshal(TPMS_SIGNATURE_SM2_MARSHAL_REF, (target), (buffer), (size))
1275    #define TPMS_SIGNATURE_SM2_Marshal(source, buffer, size)                       \
1276        Marshal(TPMS_SIGNATURE_SM2_MARSHAL_REF, (source), (buffer), (size))
1277    #define TPMS_SIGNATURE_ECSCHNORR_Unmarshal(target, buffer, size)               \
1278        Unmarshal(TPMS_SIGNATURE_ECSCHNORR_MARSHAL_REF, (target), (buffer), (size))
1279    #define TPMS_SIGNATURE_ECSCHNORR_Marshal(source, buffer, size)                 \
1280        Marshal(TPMS_SIGNATURE_ECSCHNORR_MARSHAL_REF, (source), (buffer), (size))
1281    #define TPMU_SIGNATURE_Unmarshal(target, buffer, size, selector)               \
1282        UnmarshalUnion(TPMU_SIGNATURE_MARSHAL_REF, (target), (buffer), (size),         \
1283          (selector))
1284    #define TPMU_SIGNATURE_Marshal(source, buffer, size, selector)                 \
1285        MarshalUnion(TPMU_SIGNATURE_MARSHAL_REF, (target), (buffer), (size), (selector))
1286    #define TPMT_SIGNATURE_Unmarshal(target, buffer, size, flag)                   \
1287        Unmarshal(TPMT_SIGNATURE_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),\
1288          (size))
1289    #define TPMT_SIGNATURE_Marshal(source, buffer, size)                           \
1290        Marshal(TPMT_SIGNATURE_MARSHAL_REF, (source), (buffer), (size))
1291    #define TPMU_ENCRYPTED_SECRET_Unmarshal(target, buffer, size, selector)        \
1292        UnmarshalUnion(TPMU_ENCRYPTED_SECRET_MARSHAL_REF, (target), (buffer), (size),  \
1293          (selector))
1294    #define TPMU_ENCRYPTED_SECRET_Marshal(source, buffer, size, selector)          \
1295        MarshalUnion(TPMU_ENCRYPTED_SECRET_MARSHAL_REF, (target), (buffer), (size),    \
1296          (selector))
1297    #define TPM2B_ENCRYPTED_SECRET_Unmarshal(target, buffer, size)                 \
```


 Copyright © TCG 2006-2020

```
1298        Unmarshal(TPM2B_ENCRYPTED_SECRET_MARSHAL_REF, (target), (buffer), (size))
1299    #define TPM2B_ENCRYPTED_SECRET_Marshal(source, buffer, size)                 \
1300        Marshal(TPM2B_ENCRYPTED_SECRET_MARSHAL_REF, (source), (buffer), (size))
1301    #define TPMI_ALG_PUBLIC_Unmarshal(target, buffer, size)                      \
1302        Unmarshal(TPMI_ALG_PUBLIC_MARSHAL_REF, (target), (buffer), (size))
1303    #define TPMI_ALG_PUBLIC_Marshal(source, buffer, size)                        \
1304        Marshal(TPMI_ALG_PUBLIC_MARSHAL_REF, (source), (buffer), (size))
1305    #define TPMU_PUBLIC_ID_Unmarshal(target, buffer, size, selector)             \
1306        UnmarshalUnion(TPMU_PUBLIC_ID_MARSHAL_REF, (target), (buffer), (size),   \
1307          (selector))
1308    #define TPMU_PUBLIC_ID_Marshal(source, buffer, size, selector)               \
1309        MarshalUnion(TPMU_PUBLIC_ID_MARSHAL_REF, (target), (buffer), (size), (selector))
1310    #define TPMS_KEYEDHASH_PARMS_Unmarshal(target, buffer, size)                 \
1311        Unmarshal(TPMS_KEYEDHASH_PARMS_MARSHAL_REF, (target), (buffer), (size))
1312    #define TPMS_KEYEDHASH_PARMS_Marshal(source, buffer, size)                   \
1313        Marshal(TPMS_KEYEDHASH_PARMS_MARSHAL_REF, (source), (buffer), (size))
1314    #define TPMS_RSA_PARMS_Unmarshal(target, buffer, size)                       \
1315        Unmarshal(TPMS_RSA_PARMS_MARSHAL_REF, (target), (buffer), (size))
1316    #define TPMS_RSA_PARMS_Marshal(source, buffer, size)                         \
1317        Marshal(TPMS_RSA_PARMS_MARSHAL_REF, (source), (buffer), (size))
1318    #define TPMS_ECC_PARMS_Unmarshal(target, buffer, size)                       \
1319        Unmarshal(TPMS_ECC_PARMS_MARSHAL_REF, (target), (buffer), (size))
1320    #define TPMS_ECC_PARMS_Marshal(source, buffer, size)                         \
1321        Marshal(TPMS_ECC_PARMS_MARSHAL_REF, (source), (buffer), (size))
1322    #define TPMU_PUBLIC_PARMS_Unmarshal(target, buffer, size, selector)          \
1323        UnmarshalUnion(TPMU_PUBLIC_PARMS_MARSHAL_REF, (target), (buffer), (size), \
1324          (selector))
1325    #define TPMU_PUBLIC_PARMS_Marshal(source, buffer, size, selector)            \
1326        MarshalUnion(TPMU_PUBLIC_PARMS_MARSHAL_REF, (target), (buffer), (size),  \
1327          (selector))
1328    #define TPMT_PUBLIC_PARMS_Unmarshal(target, buffer, size)                    \
1329        Unmarshal(TPMT_PUBLIC_PARMS_MARSHAL_REF, (target), (buffer), (size))
1330    #define TPMT_PUBLIC_PARMS_Marshal(source, buffer, size)                      \
1331        Marshal(TPMT_PUBLIC_PARMS_MARSHAL_REF, (source), (buffer), (size))
1332    #define TPMT_PUBLIC_Unmarshal(target, buffer, size, flag)                    \
1333        Unmarshal(TPMT_PUBLIC_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),  \
1334          (size))
1335    #define TPMT_PUBLIC_Marshal(source, buffer, size)                            \
1336        Marshal(TPMT_PUBLIC_MARSHAL_REF, (source), (buffer), (size))
1337    #define TPM2B_PUBLIC_Unmarshal(target, buffer, size, flag)                   \
1338        Unmarshal(TPM2B_PUBLIC_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),  \
1339          (size))
1340    #define TPM2B_PUBLIC_Marshal(source, buffer, size)                           \
1341        Marshal(TPM2B_PUBLIC_MARSHAL_REF, (source), (buffer), (size))
1342    #define TPM2B_TEMPLATE_Unmarshal(target, buffer, size)                       \
1343        Unmarshal(TPM2B_TEMPLATE_MARSHAL_REF, (target), (buffer), (size))
1344    #define TPM2B_TEMPLATE_Marshal(source, buffer, size)                         \
1345        Marshal(TPM2B_TEMPLATE_MARSHAL_REF, (source), (buffer), (size))
1346    #define TPM2B_PRIVATE_VENDOR_SPECIFIC_Unmarshal(target, buffer, size)        \
1347        Unmarshal(TPM2B_PRIVATE_VENDOR_SPECIFIC_MARSHAL_REF, (target), (buffer), (size))
1348    #define TPM2B_PRIVATE_VENDOR_SPECIFIC_Marshal(source, buffer, size)          \
1349        Marshal(TPM2B_PRIVATE_VENDOR_SPECIFIC_MARSHAL_REF, (source), (buffer), (size))
1350    #define TPMU_SENSITIVE_COMPOSITE_Unmarshal(target, buffer, size, selector)   \
1351        UnmarshalUnion(TPMU_SENSITIVE_COMPOSITE_MARSHAL_REF, (target), (buffer), (size),\
1352          (selector))
1353    #define TPMU_SENSITIVE_COMPOSITE_Marshal(source, buffer, size, selector)     \
1354        MarshalUnion(TPMU_SENSITIVE_COMPOSITE_MARSHAL_REF, (target), (buffer), (size),  \
1355          (selector))
1356    #define TPMT_SENSITIVE_Unmarshal(target, buffer, size)                       \
1357        Unmarshal(TPMT_SENSITIVE_MARSHAL_REF, (target), (buffer), (size))
1358    #define TPMT_SENSITIVE_Marshal(source, buffer, size)                         \
1359        Marshal(TPMT_SENSITIVE_MARSHAL_REF, (source), (buffer), (size))
1360    #define TPM2B_SENSITIVE_Unmarshal(target, buffer, size)                      \
1361        Unmarshal(TPM2B_SENSITIVE_MARSHAL_REF, (target), (buffer), (size))
1362    #define TPM2B_SENSITIVE_Marshal(source, buffer, size)                        \
1363        Marshal(TPM2B_SENSITIVE_MARSHAL_REF, (source), (buffer), (size))
```

 Copyright © TCG 2006-2020

```
1364 #define TPM2B_PRIVATE_Unmarshal(target, buffer, size)                              \
1365     Unmarshal(TPM2B_PRIVATE_MARSHAL_REF, (target), (buffer), (size))
1366 #define TPM2B_PRIVATE_Marshal(source, buffer, size)                                \
1367     Marshal(TPM2B_PRIVATE_MARSHAL_REF, (source), (buffer), (size))
1368 #define TPM2B_ID_OBJECT_Unmarshal(target, buffer, size)                            \
1369     Unmarshal(TPM2B_ID_OBJECT_MARSHAL_REF, (target), (buffer), (size))
1370 #define TPM2B_ID_OBJECT_Marshal(source, buffer, size)                              \
1371     Marshal(TPM2B_ID_OBJECT_MARSHAL_REF, (source), (buffer), (size))
1372 #define TPM_NV_INDEX_Marshal(source, buffer, size)                                 \
1373     Marshal(TPM_NV_INDEX_MARSHAL_REF, (source), (buffer), (size))
1374 #define TPMS_NV_PIN_COUNTER_PARAMETERS_Unmarshal(target, buffer, size)             \
1375     Unmarshal(TPMS_NV_PIN_COUNTER_PARAMETERS_MARSHAL_REF, (target), (buffer),      \
1376       (size))
1377 #define TPMS_NV_PIN_COUNTER_PARAMETERS_Marshal(source, buffer, size)               \
1378     Marshal(TPMS_NV_PIN_COUNTER_PARAMETERS_MARSHAL_REF, (source), (buffer), (size))
1379 #define TPMA_NV_Unmarshal(target, buffer, size)                                    \
1380     Unmarshal(TPMA_NV_MARSHAL_REF, (target), (buffer), (size))
1381 #define TPMA_NV_Marshal(source, buffer, size)                                      \
1382     Marshal(TPMA_NV_MARSHAL_REF, (source), (buffer), (size))
1383 #define TPMS_NV_PUBLIC_Unmarshal(target, buffer, size)                             \
1384     Unmarshal(TPMS_NV_PUBLIC_MARSHAL_REF, (target), (buffer), (size))
1385 #define TPMS_NV_PUBLIC_Marshal(source, buffer, size)                               \
1386     Marshal(TPMS_NV_PUBLIC_MARSHAL_REF, (source), (buffer), (size))
1387 #define TPM2B_NV_PUBLIC_Unmarshal(target, buffer, size)                            \
1388     Unmarshal(TPM2B_NV_PUBLIC_MARSHAL_REF, (target), (buffer), (size))
1389 #define TPM2B_NV_PUBLIC_Marshal(source, buffer, size)                              \
1390     Marshal(TPM2B_NV_PUBLIC_MARSHAL_REF, (source), (buffer), (size))
1391 #define TPM2B_CONTEXT_SENSITIVE_Unmarshal(target, buffer, size)                    \
1392     Unmarshal(TPM2B_CONTEXT_SENSITIVE_MARSHAL_REF, (target), (buffer), (size))
1393 #define TPM2B_CONTEXT_SENSITIVE_Marshal(source, buffer, size)                      \
1394     Marshal(TPM2B_CONTEXT_SENSITIVE_MARSHAL_REF, (source), (buffer), (size))
1395 #define TPMS_CONTEXT_DATA_Unmarshal(target, buffer, size)                          \
1396     Unmarshal(TPMS_CONTEXT_DATA_MARSHAL_REF, (target), (buffer), (size))
1397 #define TPMS_CONTEXT_DATA_Marshal(source, buffer, size)                            \
1398     Marshal(TPMS_CONTEXT_DATA_MARSHAL_REF, (source), (buffer), (size))
1399 #define TPM2B_CONTEXT_DATA_Unmarshal(target, buffer, size)                         \
1400     Unmarshal(TPM2B_CONTEXT_DATA_MARSHAL_REF, (target), (buffer), (size))
1401 #define TPM2B_CONTEXT_DATA_Marshal(source, buffer, size)                           \
1402     Marshal(TPM2B_CONTEXT_DATA_MARSHAL_REF, (source), (buffer), (size))
1403 #define TPMS_CONTEXT_Unmarshal(target, buffer, size)                               \
1404     Unmarshal(TPMS_CONTEXT_MARSHAL_REF, (target), (buffer), (size))
1405 #define TPMS_CONTEXT_Marshal(source, buffer, size)                                 \
1406     Marshal(TPMS_CONTEXT_MARSHAL_REF, (source), (buffer), (size))
1407 #define TPMS_CREATION_DATA_Marshal(source, buffer, size)                           \
1408     Marshal(TPMS_CREATION_DATA_MARSHAL_REF, (source), (buffer), (size))
1409 #define TPM2B_CREATION_DATA_Marshal(source, buffer, size)                          \
1410     Marshal(TPM2B_CREATION_DATA_MARSHAL_REF, (source), (buffer), (size))
1411 #define TPM_AT_Unmarshal(target, buffer, size)                                     \
1412     Unmarshal(TPM_AT_MARSHAL_REF, (target), (buffer), (size))
1413 #define TPM_AT_Marshal(source, buffer, size)                                       \
1414     Marshal(TPM_AT_MARSHAL_REF, (source), (buffer), (size))
1415 #define TPM_AE_Marshal(source, buffer, size)                                       \
1416     Marshal(TPM_AE_MARSHAL_REF, (source), (buffer), (size))
1417 #define TPMS_AC_OUTPUT_Marshal(source, buffer, size)                               \
1418     Marshal(TPMS_AC_OUTPUT_MARSHAL_REF, (source), (buffer), (size))
1419 #define TPML_AC_CAPABILITIES_Marshal(source, buffer, size)                         \
1420     Marshal(TPML_AC_CAPABILITIES_MARSHAL_REF, (source), (buffer), (size))
1421 #endif // _TABLE_MARSHAL_DEFINES_H_
```

9.10.7.4 TableMarshalTypes.h

```
1 #ifndef _TABLE_MARSHAL_TYPES_H_
2 #define _TABLE_MARSHAL_TYPES_H_
3 typedef UINT16      marshalIndex_t;
```


 Copyright © TCG 2006-2020

9.10.7.4.1.1 Structure Entries

A structure contains a list of elements to unmarshal. Each of the entries is a UINT16. The structure descriptor is: The *values* array contains indicators for the things to marshal. The *elements* parameter indicates how many different entities are unmarshaled. This number nominally corresponds to the number of rows in the Part 2 table that describes the structure (the number of rows minus the title row and any error code rows). A schematic of a simple structure entry is shown here but the values are not actually in a structure. As shown, the third value is the offset in the structure where the value is placed when unmarshaled, or fetched from when marshaling. This is sufficient when the element type indicated by *index* is always a simple type and never a union or array.This is just shown for illustrative purposes.

```
4   typedef struct simpleStructureEntry_t {
5       UINT16          qualifiers;         // indicates the type of entry (array, union
6                                           // etc.)
7       marshalIndex_t  index;              // the index into the appropriate array of
8                                           //  the descriptor of this type
9       UINT16          offset;             // where this comes from or is placed
10  } simpleStructureEntry_t;
11  typedef const struct UintMarshal_mst
12  {
13      UINT8           marshalType;        // UINT_MTYPE
14      UINT8           modifiers;          // size and signed indicator.
15  } UintMarshal_mst;
16  typedef struct UnionMarshal_mst
17  {
18      UINT8           countOfselectors;
19      UINT8           modifiers;          // NULL_SELECTOR
20      UINT16          offsetOfUnmarshalTypes;
21      UINT32          selectors[1];
22  //    UINT16          marshalingTypes[1]; // This is not part of the prototypical
23                                            //    entry. It is here to show where the
24                                            //    marshaling types will be in a union
25  } UnionMarshal_mst;
26  typedef struct NullUnionMarshal_mst
27  {
28      UINT8           count;
29  } NullUnionMarshal_mst;
30  typedef struct MarshalHeader_mst
31  {
32      UINT8           marshalType;        // VALUES_MTYPE
33      UINT8           modifiers;
34      UINT8           errorCode;
35  } MarshalHeader_mst;
36  typedef const struct ArrayMarshal_mst   // used in a structure
37  {
38      marshalIndex_t  type;
39      UINT16          stride;
40  } ArrayMarshal_mst;
41  typedef const struct StructMarshal_mst
42  {
43      UINT8           marshalType;        // STRUCTURE_MTYPE
44      UINT8           elements;
45      UINT16          values[1];          // three times elements
46  } StructMarshal_mst;
47  typedef const struct ValuesMarshal_mst
48  {
49      UINT8           marshalType;        // VALUES_MTYPE
50      UINT8           modifiers;
51      UINT8           errorCode;
52      UINT8           ranges;
53      UINT8           singles;
54      UINT32          values[1];
55  } ValuesMarshal_mst;
```

 Copyright © TCG 2006-2020

```
56   typedef const struct TableMarshal_mst
57   {
58       UINT8           marshalType;         // TABLE_MTYPE
59       UINT8           modifiers;
60       UINT8           errorCode;
61       UINT8           singles;
62       UINT32          values[1];
63   } TableMarshal_mst;
64   typedef const struct MinMaxMarshal_mst
65   {
66       UINT8           marshalType;         // MIN_MAX_MTYPE
67       UINT8           modifiers;
68       UINT8           errorCode;
69       UINT32          values[2];
70   } MinMaxMarshal_mst;
71   typedef const struct Tpm2bMarshal_mst
72   {
73       UINT8           unmarshalType;       // TPM2B_MTYPE
74       UINT16          sizeIndex;           // reference to type for this size value
75   } Tpm2bMarshal_mst;
76   typedef const struct Tpm2bsMarshal_mst
77   {
78       UINT8           unmarshalType;       // TPM2BS_MTYPE
79       UINT8           modifiers;           // size= and offset (2 - 7)
80       UINT16          sizeIndex;           // index of the size value;
81       UINT16          dataIndex;           // the structure
82   } Tpm2bsMarshal_mst;
83   typedef const struct ListMarshal_mst
84   {
85       UINT8           unmarshalType;       // LIST_MTYPE (for TPML)
86       UINT8           modifiers;           // size offset 2-7
87       UINT16          sizeIndex;           // reference to the minmax structure that
88                                            //      unmarshals the size parameter
89       UINT16          arrayRef;            // reference to an array definition (type
90                                            //  and stride)
91   } ListMarshal_mst;
92   typedef const struct AttributesMarshal_mst
93   {
94       UINT8           unmarashalType;      // ATTRIBUTE_MTYPE
95       UINT8           modifiers;           // size (ONE_BYTES, TWO_BYTES, or FOUR_BYTES
96       UINT32          attributeMask;       // the values that must be zero.
97   } AttributesMarshal_mst;
98   typedef const struct CompositeMarshal_mst
99   {
100      UINT8           unmashalType;        // COMPOSITE_MTYPE
101      UINT8           modifiers;           // number of entries and size
102      marshalIndex_t  types[1];            // array of unmarshaling types
103  } CompositeMarshal_mst;
104  typedef const struct TPM_ECC_CURVE_mst {
105      UINT8       marshalType;
106      UINT8       modifiers;
107      UINT8       errorCode;
108      UINT32      values[4];
109  } TPM_ECC_CURVE_mst;
110  typedef const struct TPM_CLOCK_ADJUST_mst {
111      UINT8       marshalType;
112      UINT8       modifiers;
113      UINT8       errorCode;
114      UINT32      values[2];
115  } TPM_CLOCK_ADJUST_mst;
116  typedef const struct TPM_EO_mst {
117      UINT8       marshalType;
118      UINT8       modifiers;
119      UINT8       errorCode;
120      UINT32      values[2];
121  } TPM_EO_mst;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
122     typedef const struct TPM_SU_mst {
123         UINT8           marshalType;
124         UINT8           modifiers;
125         UINT8           errorCode;
126         UINT8           entries;
127         UINT32          values[2];
128     } TPM_SU_mst;
129     typedef const struct TPM_SE_mst {
130         UINT8           marshalType;
131         UINT8           modifiers;
132         UINT8           errorCode;
133         UINT8           entries;
134         UINT32          values[3];
135     } TPM_SE_mst;
136     typedef const struct TPM_CAP_mst {
137         UINT8           marshalType;
138         UINT8           modifiers;
139         UINT8           errorCode;
140         UINT8           ranges;
141         UINT8           singles;
142         UINT32          values[3];
143     } TPM_CAP_mst;
144     typedef const struct TPMI_YES_NO_mst {
145         UINT8           marshalType;
146         UINT8           modifiers;
147         UINT8           errorCode;
148         UINT8           entries;
149         UINT32          values[2];
150     } TPMI_YES_NO_mst;
151     typedef const struct TPMI_DH_OBJECT_mst {
152         UINT8           marshalType;
153         UINT8           modifiers;
154         UINT8           errorCode;
155         UINT8           ranges;
156         UINT8           singles;
157         UINT32          values[5];
158     } TPMI_DH_OBJECT_mst;
159     typedef const struct TPMI_DH_PARENT_mst {
160         UINT8           marshalType;
161         UINT8           modifiers;
162         UINT8           errorCode;
163         UINT8           ranges;
164         UINT8           singles;
165         UINT32          values[8];
166     } TPMI_DH_PARENT_mst;
167     typedef const struct TPMI_DH_PERSISTENT_mst {
168         UINT8         marshalType;
169         UINT8         modifiers;
170         UINT8         errorCode;
171         UINT32        values[2];
172     } TPMI_DH_PERSISTENT_mst;
173     typedef const struct TPMI_DH_ENTITY_mst {
174         UINT8           marshalType;
175         UINT8           modifiers;
176         UINT8           errorCode;
177         UINT8           ranges;
178         UINT8           singles;
179         UINT32          values[15];
180     } TPMI_DH_ENTITY_mst;
181     typedef const struct TPMI_DH_PCR_mst {
182         UINT8         marshalType;
183         UINT8         modifiers;
184         UINT8         errorCode;
185         UINT32        values[3];
186     } TPMI_DH_PCR_mst;
187     typedef const struct TPMI_SH_AUTH_SESSION_mst {
```

```
188        UINT8           marshalType;
189        UINT8           modifiers;
190        UINT8           errorCode;
191        UINT8           ranges;
192        UINT8           singles;
193        UINT32          values[5];
194    } TPMI_SH_AUTH_SESSION_mst;
195    typedef const struct TPMI_SH_HMAC_mst {
196        UINT8         marshalType;
197        UINT8         modifiers;
198        UINT8         errorCode;
199        UINT32        values[2];
200    } TPMI_SH_HMAC_mst;
201    typedef const struct TPMI_SH_POLICY_mst {
202        UINT8         marshalType;
203        UINT8         modifiers;
204        UINT8         errorCode;
205        UINT32        values[2];
206    } TPMI_SH_POLICY_mst;
207    typedef const struct TPMI_DH_CONTEXT_mst {
208        UINT8           marshalType;
209        UINT8           modifiers;
210        UINT8           errorCode;
211        UINT8           ranges;
212        UINT8           singles;
213        UINT32          values[6];
214    } TPMI_DH_CONTEXT_mst;
215    typedef const struct TPMI_DH_SAVED_mst {
216        UINT8           marshalType;
217        UINT8           modifiers;
218        UINT8           errorCode;
219        UINT8           ranges;
220        UINT8           singles;
221        UINT32          values[7];
222    } TPMI_DH_SAVED_mst;
223    typedef const struct TPMI_RH_HIERARCHY_mst {
224        UINT8           marshalType;
225        UINT8           modifiers;
226        UINT8           errorCode;
227        UINT8           entries;
228        UINT32          values[4];
229    } TPMI_RH_HIERARCHY_mst;
230    typedef const struct TPMI_RH_ENABLES_mst {
231        UINT8           marshalType;
232        UINT8           modifiers;
233        UINT8           errorCode;
234        UINT8           entries;
235        UINT32          values[5];
236    } TPMI_RH_ENABLES_mst;
237    typedef const struct TPMI_RH_HIERARCHY_AUTH_mst {
238        UINT8           marshalType;
239        UINT8           modifiers;
240        UINT8           errorCode;
241        UINT8           entries;
242        UINT32          values[4];
243    } TPMI_RH_HIERARCHY_AUTH_mst;
244    typedef const struct TPMI_RH_HIERARCHY_POLICY_mst {
245        UINT8           marshalType;
246        UINT8           modifiers;
247        UINT8           errorCode;
248        UINT8           ranges;
249        UINT8           singles;
250        UINT32          values[6];
251    } TPMI_RH_HIERARCHY_POLICY_mst;
252    typedef const struct TPMI_RH_PLATFORM_mst {
253        UINT8           marshalType;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
254         UINT8          modifiers;
255         UINT8          errorCode;
256         UINT8          entries;
257         UINT32         values[1];
258     } TPMI_RH_PLATFORM_mst;
259     typedef const struct TPMI_RH_OWNER_mst {
260         UINT8          marshalType;
261         UINT8          modifiers;
262         UINT8          errorCode;
263         UINT8          entries;
264         UINT32         values[2];
265     } TPMI_RH_OWNER_mst;
266     typedef const struct TPMI_RH_ENDORSEMENT_mst {
267         UINT8          marshalType;
268         UINT8          modifiers;
269         UINT8          errorCode;
270         UINT8          entries;
271         UINT32         values[2];
272     } TPMI_RH_ENDORSEMENT_mst;
273     typedef const struct TPMI_RH_PROVISION_mst {
274         UINT8          marshalType;
275         UINT8          modifiers;
276         UINT8          errorCode;
277         UINT8          entries;
278         UINT32         values[2];
279     } TPMI_RH_PROVISION_mst;
280     typedef const struct TPMI_RH_CLEAR_mst {
281         UINT8          marshalType;
282         UINT8          modifiers;
283         UINT8          errorCode;
284         UINT8          entries;
285         UINT32         values[2];
286     } TPMI_RH_CLEAR_mst;
287     typedef const struct TPMI_RH_NV_AUTH_mst {
288         UINT8          marshalType;
289         UINT8          modifiers;
290         UINT8          errorCode;
291         UINT8          ranges;
292         UINT8          singles;
293         UINT32         values[4];
294     } TPMI_RH_NV_AUTH_mst;
295     typedef const struct TPMI_RH_LOCKOUT_mst {
296         UINT8          marshalType;
297         UINT8          modifiers;
298         UINT8          errorCode;
299         UINT8          entries;
300         UINT32         values[1];
301     } TPMI_RH_LOCKOUT_mst;
302     typedef const struct TPMI_RH_NV_INDEX_mst {
303         UINT8        marshalType;
304         UINT8        modifiers;
305         UINT8        errorCode;
306         UINT32       values[2];
307     } TPMI_RH_NV_INDEX_mst;
308     typedef const struct TPMI_RH_AC_mst {
309         UINT8        marshalType;
310         UINT8        modifiers;
311         UINT8        errorCode;
312         UINT32       values[2];
313     } TPMI_RH_AC_mst;
314     typedef const struct TPMI_RH_ACT_mst {
315         UINT8        marshalType;
316         UINT8        modifiers;
317         UINT8        errorCode;
318         UINT32       values[2];
319     } TPMI_RH_ACT_mst;
```

 Copyright © TCG 2006-2020

```
320 typedef const struct TPMI_ALG_HASH_mst {
321     UINT8       marshalType;
322     UINT8       modifiers;
323     UINT8       errorCode;
324     UINT32      values[5];
325 } TPMI_ALG_HASH_mst;
326 typedef const struct TPMI_ALG_ASYM_mst {
327     UINT8       marshalType;
328     UINT8       modifiers;
329     UINT8       errorCode;
330     UINT32      values[5];
331 } TPMI_ALG_ASYM_mst;
332 typedef const struct TPMI_ALG_SYM_mst {
333     UINT8       marshalType;
334     UINT8       modifiers;
335     UINT8       errorCode;
336     UINT32      values[5];
337 } TPMI_ALG_SYM_mst;
338 typedef const struct TPMI_ALG_SYM_OBJECT_mst {
339     UINT8       marshalType;
340     UINT8       modifiers;
341     UINT8       errorCode;
342     UINT32      values[5];
343 } TPMI_ALG_SYM_OBJECT_mst;
344 typedef const struct TPMI_ALG_SYM_MODE_mst {
345     UINT8       marshalType;
346     UINT8       modifiers;
347     UINT8       errorCode;
348     UINT32      values[4];
349 } TPMI_ALG_SYM_MODE_mst;
350 typedef const struct TPMI_ALG_KDF_mst {
351     UINT8       marshalType;
352     UINT8       modifiers;
353     UINT8       errorCode;
354     UINT32      values[4];
355 } TPMI_ALG_KDF_mst;
356 typedef const struct TPMI_ALG_SIG_SCHEME_mst {
357     UINT8       marshalType;
358     UINT8       modifiers;
359     UINT8       errorCode;
360     UINT32      values[4];
361 } TPMI_ALG_SIG_SCHEME_mst;
362 typedef const struct TPMI_ECC_KEY_EXCHANGE_mst {
363     UINT8       marshalType;
364     UINT8       modifiers;
365     UINT8       errorCode;
366     UINT32      values[4];
367 } TPMI_ECC_KEY_EXCHANGE_mst;
368 typedef const struct TPMI_ST_COMMAND_TAG_mst {
369     UINT8         marshalType;
370     UINT8         modifiers;
371     UINT8         errorCode;
372     UINT8         entries;
373     UINT32        values[2];
374 } TPMI_ST_COMMAND_TAG_mst;
375 typedef const struct TPMI_ALG_MAC_SCHEME_mst {
376     UINT8       marshalType;
377     UINT8       modifiers;
378     UINT8       errorCode;
379     UINT32      values[5];
380 } TPMI_ALG_MAC_SCHEME_mst;
381 typedef const struct TPMI_ALG_CIPHER_MODE_mst {
382     UINT8       marshalType;
383     UINT8       modifiers;
384     UINT8       errorCode;
385     UINT32      values[4];
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
386    } TPMI_ALG_CIPHER_MODE_mst;
387    typedef const struct TPMS_EMPTY_mst
388    {
389        UINT8     marshalType;
390        UINT8     elements;
391        UINT16    values[3];
392    } TPMS_EMPTY_mst;
393    typedef const struct TPMS_ALGORITHM_DESCRIPTION_mst
394    {
395        UINT8     marshalType;
396        UINT8     elements;
397        UINT16    values[6];
398    } TPMS_ALGORITHM_DESCRIPTION_mst;
399    typedef struct TPMU_HA_mst
400    {
401        BYTE              countOfselectors;
402        BYTE              modifiers;
403        UINT16          offsetOfUnmarshalTypes;
404        UINT32          selectors[9];
405        UINT16          marshalingTypes[9];
406    } TPMU_HA_mst;
407    typedef const struct TPMT_HA_mst
408    {
409        UINT8     marshalType;
410        UINT8     elements;
411        UINT16    values[6];
412    } TPMT_HA_mst;
413    typedef const struct TPMS_PCR_SELECT_mst
414    {
415        UINT8     marshalType;
416        UINT8     elements;
417        UINT16    values[6];
418    } TPMS_PCR_SELECT_mst;
419    typedef const struct TPMS_PCR_SELECTION_mst
420    {
421        UINT8     marshalType;
422        UINT8     elements;
423        UINT16    values[9];
424    } TPMS_PCR_SELECTION_mst;
425    typedef const struct TPMT_TK_CREATION_mst
426    {
427        UINT8     marshalType;
428        UINT8     elements;
429        UINT16    values[9];
430    } TPMT_TK_CREATION_mst;
431    typedef const struct TPMT_TK_VERIFIED_mst
432    {
433        UINT8     marshalType;
434        UINT8     elements;
435        UINT16    values[9];
436    } TPMT_TK_VERIFIED_mst;
437    typedef const struct TPMT_TK_AUTH_mst
438    {
439        UINT8     marshalType;
440        UINT8     elements;
441        UINT16    values[9];
442    } TPMT_TK_AUTH_mst;
443    typedef const struct TPMT_TK_HASHCHECK_mst
444    {
445        UINT8     marshalType;
446        UINT8     elements;
447        UINT16    values[9];
448    } TPMT_TK_HASHCHECK_mst;
449    typedef const struct TPMS_ALG_PROPERTY_mst
450    {
451        UINT8     marshalType;
```

 Copyright © TCG 2006-2020

```
452         UINT8      elements;
453         UINT16     values[6];
454     } TPMS_ALG_PROPERTY_mst;
455     typedef const struct TPMS_TAGGED_PROPERTY_mst
456     {
457         UINT8      marshalType;
458         UINT8      elements;
459         UINT16     values[6];
460     } TPMS_TAGGED_PROPERTY_mst;
461     typedef const struct TPMS_TAGGED_PCR_SELECT_mst
462     {
463         UINT8      marshalType;
464         UINT8      elements;
465         UINT16     values[9];
466     } TPMS_TAGGED_PCR_SELECT_mst;
467     typedef const struct TPMS_TAGGED_POLICY_mst
468     {
469         UINT8      marshalType;
470         UINT8      elements;
471         UINT16     values[6];
472     } TPMS_TAGGED_POLICY_mst;
473     typedef const struct TPMS_ACT_DATA_mst
474     {
475         UINT8      marshalType;
476         UINT8      elements;
477         UINT16     values[9];
478     } TPMS_ACT_DATA_mst;
479     typedef struct TPMU_CAPABILITIES_mst
480     {
481         BYTE             countOfselectors;
482         BYTE             modifiers;
483         UINT16          offsetOfUnmarshalTypes;
484         UINT32          selectors[11];
485         UINT16          marshalingTypes[11];
486     } TPMU_CAPABILITIES_mst;
487     typedef const struct TPMS_CAPABILITY_DATA_mst
488     {
489         UINT8      marshalType;
490         UINT8      elements;
491         UINT16     values[6];
492     } TPMS_CAPABILITY_DATA_mst;
493     typedef const struct TPMS_CLOCK_INFO_mst
494     {
495         UINT8      marshalType;
496         UINT8      elements;
497         UINT16     values[12];
498     } TPMS_CLOCK_INFO_mst;
499     typedef const struct TPMS_TIME_INFO_mst
500     {
501         UINT8      marshalType;
502         UINT8      elements;
503         UINT16     values[6];
504     } TPMS_TIME_INFO_mst;
505     typedef const struct TPMS_TIME_ATTEST_INFO_mst
506     {
507         UINT8      marshalType;
508         UINT8      elements;
509         UINT16     values[6];
510     } TPMS_TIME_ATTEST_INFO_mst;
511     typedef const struct TPMS_CERTIFY_INFO_mst
512     {
513         UINT8      marshalType;
514         UINT8      elements;
515         UINT16     values[6];
516     } TPMS_CERTIFY_INFO_mst;
517     typedef const struct TPMS_QUOTE_INFO_mst
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
518 {
519     UINT8     marshalType;
520     UINT8     elements;
521     UINT16    values[6];
522 } TPMS_QUOTE_INFO_mst;
523 typedef const struct TPMS_COMMAND_AUDIT_INFO_mst
524 {
525     UINT8     marshalType;
526     UINT8     elements;
527     UINT16    values[12];
528 } TPMS_COMMAND_AUDIT_INFO_mst;
529 typedef const struct TPMS_SESSION_AUDIT_INFO_mst
530 {
531     UINT8     marshalType;
532     UINT8     elements;
533     UINT16    values[6];
534 } TPMS_SESSION_AUDIT_INFO_mst;
535 typedef const struct TPMS_CREATION_INFO_mst
536 {
537     UINT8     marshalType;
538     UINT8     elements;
539     UINT16    values[6];
540 } TPMS_CREATION_INFO_mst;
541 typedef const struct TPMS_NV_CERTIFY_INFO_mst
542 {
543     UINT8     marshalType;
544     UINT8     elements;
545     UINT16    values[9];
546 } TPMS_NV_CERTIFY_INFO_mst;
547 typedef const struct TPMS_NV_DIGEST_CERTIFY_INFO_mst
548 {
549     UINT8     marshalType;
550     UINT8     elements;
551     UINT16    values[6];
552 } TPMS_NV_DIGEST_CERTIFY_INFO_mst;
553 typedef const struct TPMI_ST_ATTEST_mst {
554     UINT8         marshalType;
555     UINT8         modifiers;
556     UINT8         errorCode;
557     UINT8         ranges;
558     UINT8         singles;
559     UINT32        values[3];
560 } TPMI_ST_ATTEST_mst;
561 typedef struct TPMU_ATTEST_mst
562 {
563     BYTE            countOfselectors;
564     BYTE            modifiers;
565     UINT16        offsetOfUnmarshalTypes;
566     UINT32        selectors[8];
567     UINT16        marshalingTypes[8];
568 } TPMU_ATTEST_mst;
569 typedef const struct TPMS_ATTEST_mst
570 {
571     UINT8     marshalType;
572     UINT8     elements;
573     UINT16    values[21];
574 } TPMS_ATTEST_mst;
575 typedef const struct TPMS_AUTH_COMMAND_mst
576 {
577     UINT8     marshalType;
578     UINT8     elements;
579     UINT16    values[12];
580 } TPMS_AUTH_COMMAND_mst;
581 typedef const struct TPMS_AUTH_RESPONSE_mst
582 {
583     UINT8     marshalType;
```

```
584         UINT8      elements;
585         UINT16     values[9];
586     } TPMS_AUTH_RESPONSE_mst;
587     typedef const struct TPMI_TDES_KEY_BITS_mst {
588         UINT8          marshalType;
589         UINT8          modifiers;
590         UINT8          errorCode;
591         UINT8          entries;
592         UINT32         values[1];
593     } TPMI_TDES_KEY_BITS_mst;
594     typedef const struct TPMI_AES_KEY_BITS_mst {
595         UINT8          marshalType;
596         UINT8          modifiers;
597         UINT8          errorCode;
598         UINT8          entries;
599         UINT32         values[3];
600     } TPMI_AES_KEY_BITS_mst;
601     typedef const struct TPMI_SM4_KEY_BITS_mst {
602         UINT8          marshalType;
603         UINT8          modifiers;
604         UINT8          errorCode;
605         UINT8          entries;
606         UINT32         values[1];
607     } TPMI_SM4_KEY_BITS_mst;
608     typedef const struct TPMI_CAMELLIA_KEY_BITS_mst {
609         UINT8          marshalType;
610         UINT8          modifiers;
611         UINT8          errorCode;
612         UINT8          entries;
613         UINT32         values[3];
614     } TPMI_CAMELLIA_KEY_BITS_mst;
615     typedef struct TPMU_SYM_KEY_BITS_mst
616     {
617         BYTE             countOfselectors;
618         BYTE             modifiers;
619         UINT16         offsetOfUnmarshalTypes;
620         UINT32         selectors[6];
621         UINT16         marshalingTypes[6];
622     } TPMU_SYM_KEY_BITS_mst;
623     typedef struct TPMU_SYM_MODE_mst
624     {
625         BYTE             countOfselectors;
626         BYTE             modifiers;
627         UINT16         offsetOfUnmarshalTypes;
628         UINT32         selectors[6];
629         UINT16         marshalingTypes[6];
630     } TPMU_SYM_MODE_mst;
631     typedef const struct TPMT_SYM_DEF_mst
632     {
633         UINT8      marshalType;
634         UINT8      elements;
635         UINT16     values[9];
636     } TPMT_SYM_DEF_mst;
637     typedef const struct TPMT_SYM_DEF_OBJECT_mst
638     {
639         UINT8      marshalType;
640         UINT8      elements;
641         UINT16     values[9];
642     } TPMT_SYM_DEF_OBJECT_mst;
643     typedef const struct TPMS_SYMCIPHER_PARMS_mst
644     {
645         UINT8      marshalType;
646         UINT8      elements;
647         UINT16     values[3];
648     } TPMS_SYMCIPHER_PARMS_mst;
649     typedef const struct TPMS_DERIVE_mst
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
650   {
651       UINT8     marshalType;
652       UINT8     elements;
653       UINT16    values[6];
654   } TPMS_DERIVE_mst;
655   typedef const struct TPMS_SENSITIVE_CREATE_mst
656   {
657       UINT8     marshalType;
658       UINT8     elements;
659       UINT16    values[6];
660   } TPMS_SENSITIVE_CREATE_mst;
661   typedef const struct TPMS_SCHEME_HASH_mst
662   {
663       UINT8     marshalType;
664       UINT8     elements;
665       UINT16    values[3];
666   } TPMS_SCHEME_HASH_mst;
667   typedef const struct TPMS_SCHEME_ECDAA_mst
668   {
669       UINT8     marshalType;
670       UINT8     elements;
671       UINT16    values[6];
672   } TPMS_SCHEME_ECDAA_mst;
673   typedef const struct TPMI_ALG_KEYEDHASH_SCHEME_mst {
674       UINT8       marshalType;
675       UINT8       modifiers;
676       UINT8       errorCode;
677       UINT32      values[4];
678   } TPMI_ALG_KEYEDHASH_SCHEME_mst;
679   typedef const struct TPMS_SCHEME_XOR_mst
680   {
681       UINT8     marshalType;
682       UINT8     elements;
683       UINT16    values[6];
684   } TPMS_SCHEME_XOR_mst;
685   typedef struct TPMU_SCHEME_KEYEDHASH_mst
686   {
687       BYTE            countOfselectors;
688       BYTE            modifiers;
689       UINT16        offsetOfUnmarshalTypes;
690       UINT32        selectors[3];
691       UINT16        marshalingTypes[3];
692   } TPMU_SCHEME_KEYEDHASH_mst;
693   typedef const struct TPMT_KEYEDHASH_SCHEME_mst
694   {
695       UINT8     marshalType;
696       UINT8     elements;
697       UINT16    values[6];
698   } TPMT_KEYEDHASH_SCHEME_mst;
699   typedef struct TPMU_SIG_SCHEME_mst
700   {
701       BYTE            countOfselectors;
702       BYTE            modifiers;
703       UINT16        offsetOfUnmarshalTypes;
704       UINT32        selectors[8];
705       UINT16        marshalingTypes[8];
706   } TPMU_SIG_SCHEME_mst;
707   typedef const struct TPMT_SIG_SCHEME_mst
708   {
709       UINT8     marshalType;
710       UINT8     elements;
711       UINT16    values[6];
712   } TPMT_SIG_SCHEME_mst;
713   typedef struct TPMU_KDF_SCHEME_mst
714   {
715       BYTE            countOfselectors;
```

 Copyright © TCG 2006-2020

```
716     BYTE             modifiers;
717     UINT16         offsetOfUnmarshalTypes;
718     UINT32         selectors[5];
719     UINT16         marshalingTypes[5];
720 } TPMU_KDF_SCHEME_mst;
721 typedef const struct TPMT_KDF_SCHEME_mst
722 {
723     UINT8     marshalType;
724     UINT8     elements;
725     UINT16    values[6];
726 } TPMT_KDF_SCHEME_mst;
727 typedef const struct TPMI_ALG_ASYM_SCHEME_mst {
728     UINT8       marshalType;
729     UINT8       modifiers;
730     UINT8       errorCode;
731     UINT32      values[4];
732 } TPMI_ALG_ASYM_SCHEME_mst;
733 typedef struct TPMU_ASYM_SCHEME_mst
734 {
735     BYTE             countOfselectors;
736     BYTE             modifiers;
737     UINT16         offsetOfUnmarshalTypes;
738     UINT32         selectors[11];
739     UINT16         marshalingTypes[11];
740 } TPMU_ASYM_SCHEME_mst;
741 typedef const struct TPMI_ALG_RSA_SCHEME_mst {
742     UINT8       marshalType;
743     UINT8       modifiers;
744     UINT8       errorCode;
745     UINT32      values[4];
746 } TPMI_ALG_RSA_SCHEME_mst;
747 typedef const struct TPMT_RSA_SCHEME_mst
748 {
749     UINT8     marshalType;
750     UINT8     elements;
751     UINT16    values[6];
752 } TPMT_RSA_SCHEME_mst;
753 typedef const struct TPMI_ALG_RSA_DECRYPT_mst {
754     UINT8       marshalType;
755     UINT8       modifiers;
756     UINT8       errorCode;
757     UINT32      values[4];
758 } TPMI_ALG_RSA_DECRYPT_mst;
759 typedef const struct TPMT_RSA_DECRYPT_mst
760 {
761     UINT8     marshalType;
762     UINT8     elements;
763     UINT16    values[6];
764 } TPMT_RSA_DECRYPT_mst;
765 typedef const struct TPMI_RSA_KEY_BITS_mst {
766     UINT8         marshalType;
767     UINT8         modifiers;
768     UINT8         errorCode;
769     UINT8         entries;
770     UINT32        values[3];
771 } TPMI_RSA_KEY_BITS_mst;
772 typedef const struct TPMS_ECC_POINT_mst
773 {
774     UINT8     marshalType;
775     UINT8     elements;
776     UINT16    values[6];
777 } TPMS_ECC_POINT_mst;
778 typedef const struct TPMI_ALG_ECC_SCHEME_mst {
779     UINT8       marshalType;
780     UINT8       modifiers;
781     UINT8       errorCode;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
782         UINT32       values[4];
783     } TPMI_ALG_ECC_SCHEME_mst;
784     typedef const struct TPMI_ECC_CURVE_mst {
785         UINT8        marshalType;
786         UINT8        modifiers;
787         UINT8        errorCode;
788         UINT32       values[3];
789     } TPMI_ECC_CURVE_mst;
790     typedef const struct TPMT_ECC_SCHEME_mst
791     {
792         UINT8      marshalType;
793         UINT8      elements;
794         UINT16     values[6];
795     } TPMT_ECC_SCHEME_mst;
796     typedef const struct TPMS_ALGORITHM_DETAIL_ECC_mst
797     {
798         UINT8      marshalType;
799         UINT8      elements;
800         UINT16     values[33];
801     } TPMS_ALGORITHM_DETAIL_ECC_mst;
802     typedef const struct TPMS_SIGNATURE_RSA_mst
803     {
804         UINT8      marshalType;
805         UINT8      elements;
806         UINT16     values[6];
807     } TPMS_SIGNATURE_RSA_mst;
808     typedef const struct TPMS_SIGNATURE_ECC_mst
809     {
810         UINT8      marshalType;
811         UINT8      elements;
812         UINT16     values[9];
813     } TPMS_SIGNATURE_ECC_mst;
814     typedef struct TPMU_SIGNATURE_mst
815     {
816         BYTE             countOfselectors;
817         BYTE             modifiers;
818         UINT16         offsetOfUnmarshalTypes;
819         UINT32         selectors[8];
820         UINT16         marshalingTypes[8];
821     } TPMU_SIGNATURE_mst;
822     typedef const struct TPMT_SIGNATURE_mst
823     {
824         UINT8      marshalType;
825         UINT8      elements;
826         UINT16     values[6];
827     } TPMT_SIGNATURE_mst;
828     typedef struct TPMU_ENCRYPTED_SECRET_mst
829     {
830         BYTE             countOfselectors;
831         BYTE             modifiers;
832         UINT16         offsetOfUnmarshalTypes;
833         UINT32         selectors[4];
834         UINT16         marshalingTypes[4];
835     } TPMU_ENCRYPTED_SECRET_mst;
836     typedef const struct TPMI_ALG_PUBLIC_mst {
837         UINT8        marshalType;
838         UINT8        modifiers;
839         UINT8        errorCode;
840         UINT32       values[4];
841     } TPMI_ALG_PUBLIC_mst;
842     typedef struct TPMU_PUBLIC_ID_mst
843     {
844         BYTE             countOfselectors;
845         BYTE             modifiers;
846         UINT16         offsetOfUnmarshalTypes;
847         UINT32         selectors[4];
```

```
848         UINT16          marshalingTypes[4];
849     } TPMU_PUBLIC_ID_mst;
850     typedef const struct TPMS_KEYEDHASH_PARMS_mst
851     {
852         UINT8       marshalType;
853         UINT8       elements;
854         UINT16      values[3];
855     } TPMS_KEYEDHASH_PARMS_mst;
856     typedef const struct TPMS_RSA_PARMS_mst
857     {
858         UINT8       marshalType;
859         UINT8       elements;
860         UINT16      values[12];
861     } TPMS_RSA_PARMS_mst;
862     typedef const struct TPMS_ECC_PARMS_mst
863     {
864         UINT8       marshalType;
865         UINT8       elements;
866         UINT16      values[12];
867     } TPMS_ECC_PARMS_mst;
868     typedef struct TPMU_PUBLIC_PARMS_mst
869     {
870         BYTE                countOfselectors;
871         BYTE                modifiers;
872         UINT16          offsetOfUnmarshalTypes;
873         UINT32          selectors[4];
874         UINT16          marshalingTypes[4];
875     } TPMU_PUBLIC_PARMS_mst;
876     typedef const struct TPMT_PUBLIC_PARMS_mst
877     {
878         UINT8       marshalType;
879         UINT8       elements;
880         UINT16      values[6];
881     } TPMT_PUBLIC_PARMS_mst;
882     typedef const struct TPMT_PUBLIC_mst
883     {
884         UINT8       marshalType;
885         UINT8       elements;
886         UINT16      values[18];
887     } TPMT_PUBLIC_mst;
888     typedef struct TPMU_SENSITIVE_COMPOSITE_mst
889     {
890         BYTE                countOfselectors;
891         BYTE                modifiers;
892         UINT16          offsetOfUnmarshalTypes;
893         UINT32          selectors[4];
894         UINT16          marshalingTypes[4];
895     } TPMU_SENSITIVE_COMPOSITE_mst;
896     typedef const struct TPMT_SENSITIVE_mst
897     {
898         UINT8       marshalType;
899         UINT8       elements;
900         UINT16      values[12];
901     } TPMT_SENSITIVE_mst;
902     typedef const struct TPMS_NV_PIN_COUNTER_PARAMETERS_mst
903     {
904         UINT8       marshalType;
905         UINT8       elements;
906         UINT16      values[6];
907     } TPMS_NV_PIN_COUNTER_PARAMETERS_mst;
908     typedef const struct TPMS_NV_PUBLIC_mst
909     {
910         UINT8       marshalType;
911         UINT8       elements;
912         UINT16      values[15];
913     } TPMS_NV_PUBLIC_mst;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
914 typedef const struct TPMS_CONTEXT_DATA_mst
915 {
916     UINT8     marshalType;
917     UINT8     elements;
918     UINT16    values[6];
919 } TPMS_CONTEXT_DATA_mst;
920 typedef const struct TPMS_CONTEXT_mst
921 {
922     UINT8     marshalType;
923     UINT8     elements;
924     UINT16    values[12];
925 } TPMS_CONTEXT_mst;
926 typedef const struct TPMS_CREATION_DATA_mst
927 {
928     UINT8     marshalType;
929     UINT8     elements;
930     UINT16    values[21];
931 } TPMS_CREATION_DATA_mst;
932 typedef const struct TPM_AT_mst {
933     UINT8         marshalType;
934     UINT8         modifiers;
935     UINT8         errorCode;
936     UINT8         entries;
937     UINT32        values[4];
938 } TPM_AT_mst;
939 typedef const struct TPMS_AC_OUTPUT_mst
940 {
941     UINT8     marshalType;
942     UINT8     elements;
943     UINT16    values[6];
944 } TPMS_AC_OUTPUT_mst;
945 typedef const struct Type02_mst {
946     UINT8       marshalType;
947     UINT8       modifiers;
948     UINT8       errorCode;
949     UINT32      values[2];
950 } Type02_mst;
951 typedef const struct Type03_mst {
952     UINT8       marshalType;
953     UINT8       modifiers;
954     UINT8       errorCode;
955     UINT32      values[2];
956 } Type03_mst;
957 typedef const struct Type04_mst {
958     UINT8       marshalType;
959     UINT8       modifiers;
960     UINT8       errorCode;
961     UINT32      values[2];
962 } Type04_mst;
963 typedef const struct Type06_mst {
964     UINT8       marshalType;
965     UINT8       modifiers;
966     UINT8       errorCode;
967     UINT32      values[2];
968 } Type06_mst;
969 typedef const struct Type08_mst {
970     UINT8       marshalType;
971     UINT8       modifiers;
972     UINT8       errorCode;
973     UINT32      values[2];
974 } Type08_mst;
975 typedef const struct Type10_mst {
976     UINT8         marshalType;
977     UINT8         modifiers;
978     UINT8         errorCode;
979     UINT8         entries;
```

 Copyright © TCG 2006-2020

```
980      UINT32        values[1];
981  } Type10_mst;
982  typedef const struct Type11_mst {
983      UINT8         marshalType;
984      UINT8         modifiers;
985      UINT8         errorCode;
986      UINT8         entries;
987      UINT32        values[1];
988  } Type11_mst;
989  typedef const struct Type12_mst {
990      UINT8         marshalType;
991      UINT8         modifiers;
992      UINT8         errorCode;
993      UINT8         entries;
994      UINT32        values[2];
995  } Type12_mst;
996  typedef const struct Type13_mst {
997      UINT8         marshalType;
998      UINT8         modifiers;
999      UINT8         errorCode;
1000     UINT8         entries;
1001     UINT32        values[1];
1002 } Type13_mst;
1003 typedef const struct Type15_mst {
1004     UINT8       marshalType;
1005     UINT8       modifiers;
1006     UINT8       errorCode;
1007     UINT32      values[2];
1008 } Type15_mst;
1009 typedef const struct Type17_mst {
1010     UINT8       marshalType;
1011     UINT8       modifiers;
1012     UINT8       errorCode;
1013     UINT32      values[2];
1014 } Type17_mst;
1015 typedef const struct Type18_mst {
1016     UINT8       marshalType;
1017     UINT8       modifiers;
1018     UINT8       errorCode;
1019     UINT32      values[2];
1020 } Type18_mst;
1021 typedef const struct Type19_mst {
1022     UINT8       marshalType;
1023     UINT8       modifiers;
1024     UINT8       errorCode;
1025     UINT32      values[2];
1026 } Type19_mst;
1027 typedef const struct Type20_mst {
1028     UINT8       marshalType;
1029     UINT8       modifiers;
1030     UINT8       errorCode;
1031     UINT32      values[2];
1032 } Type20_mst;
1033 typedef const struct Type22_mst {
1034     UINT8       marshalType;
1035     UINT8       modifiers;
1036     UINT8       errorCode;
1037     UINT32      values[2];
1038 } Type22_mst;
1039 typedef const struct Type23_mst {
1040     UINT8       marshalType;
1041     UINT8       modifiers;
1042     UINT8       errorCode;
1043     UINT32      values[2];
1044 } Type23_mst;
1045 typedef const struct Type24_mst {
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1046        UINT8        marshalType;
1047        UINT8        modifiers;
1048        UINT8        errorCode;
1049        UINT32       values[2];
1050    } Type24_mst;
1051    typedef const struct Type25_mst {
1052        UINT8        marshalType;
1053        UINT8        modifiers;
1054        UINT8        errorCode;
1055        UINT32       values[2];
1056    } Type25_mst;
1057    typedef const struct Type26_mst {
1058        UINT8        marshalType;
1059        UINT8        modifiers;
1060        UINT8        errorCode;
1061        UINT32       values[2];
1062    } Type26_mst;
1063    typedef const struct Type27_mst {
1064        UINT8        marshalType;
1065        UINT8        modifiers;
1066        UINT8        errorCode;
1067        UINT32       values[2];
1068    } Type27_mst;
1069    typedef const struct Type29_mst {
1070        UINT8        marshalType;
1071        UINT8        modifiers;
1072        UINT8        errorCode;
1073        UINT32       values[2];
1074    } Type29_mst;
1075    typedef const struct Type30_mst {
1076        UINT8        marshalType;
1077        UINT8        modifiers;
1078        UINT8        errorCode;
1079        UINT32       values[2];
1080    } Type30_mst;
1081    typedef const struct Type33_mst {
1082        UINT8        marshalType;
1083        UINT8        modifiers;
1084        UINT8        errorCode;
1085        UINT32       values[2];
1086    } Type33_mst;
1087    typedef const struct Type34_mst {
1088        UINT8        marshalType;
1089        UINT8        modifiers;
1090        UINT8        errorCode;
1091        UINT32       values[2];
1092    } Type34_mst;
1093    typedef const struct Type35_mst {
1094        UINT8        marshalType;
1095        UINT8        modifiers;
1096        UINT8        errorCode;
1097        UINT32       values[2];
1098    } Type35_mst;
1099    typedef const struct Type38_mst {
1100        UINT8        marshalType;
1101        UINT8        modifiers;
1102        UINT8        errorCode;
1103        UINT32       values[2];
1104    } Type38_mst;
1105    typedef const struct Type41_mst {
1106        UINT8        marshalType;
1107        UINT8        modifiers;
1108        UINT8        errorCode;
1109        UINT32       values[2];
1110    } Type41_mst;
1111    typedef const struct Type42_mst {
```

 Copyright © TCG 2006-2020

```
1112        UINT8       marshalType;
1113        UINT8       modifiers;
1114        UINT8       errorCode;
1115        UINT32      values[2];
1116    } Type42_mst;
1117    typedef const struct Type44_mst {
1118        UINT8       marshalType;
1119        UINT8       modifiers;
1120        UINT8       errorCode;
1121        UINT32      values[2];
1122    } Type44_mst;
```

This structure combines all the individual marshaling structures to build something that can be referenced by offset rather than full address

```
1123    typedef const struct MarshalData_st {
1124        UintMarshal_mst                 UINT8_DATA;
1125        UintMarshal_mst                 UINT16_DATA;
1126        UintMarshal_mst                 UINT32_DATA;
1127        UintMarshal_mst                 UINT64_DATA;
1128        UintMarshal_mst                 INT8_DATA;
1129        UintMarshal_mst                 INT16_DATA;
1130        UintMarshal_mst                 INT32_DATA;
1131        UintMarshal_mst                 INT64_DATA;
1132        UintMarshal_mst                 UINT0_DATA;
1133        TPM_ECC_CURVE_mst               TPM_ECC_CURVE_DATA;
1134        TPM_CLOCK_ADJUST_mst            TPM_CLOCK_ADJUST_DATA;
1135        TPM_EO_mst                      TPM_EO_DATA;
1136        TPM_SU_mst                      TPM_SU_DATA;
1137        TPM_SE_mst                      TPM_SE_DATA;
1138        TPM_CAP_mst                     TPM_CAP_DATA;
1139        AttributesMarshal_mst           TPMA_ALGORITHM_DATA;
1140        AttributesMarshal_mst           TPMA_OBJECT_DATA;
1141        AttributesMarshal_mst           TPMA_SESSION_DATA;
1142        AttributesMarshal_mst           TPMA_ACT_DATA;
1143        TPMI_YES_NO_mst                 TPMI_YES_NO_DATA;
1144        TPMI_DH_OBJECT_mst              TPMI_DH_OBJECT_DATA;
1145        TPMI_DH_PARENT_mst              TPMI_DH_PARENT_DATA;
1146        TPMI_DH_PERSISTENT_mst          TPMI_DH_PERSISTENT_DATA;
1147        TPMI_DH_ENTITY_mst              TPMI_DH_ENTITY_DATA;
1148        TPMI_DH_PCR_mst                 TPMI_DH_PCR_DATA;
1149        TPMI_SH_AUTH_SESSION_mst        TPMI_SH_AUTH_SESSION_DATA;
1150        TPMI_SH_HMAC_mst                TPMI_SH_HMAC_DATA;
1151        TPMI_SH_POLICY_mst              TPMI_SH_POLICY_DATA;
1152        TPMI_DH_CONTEXT_mst             TPMI_DH_CONTEXT_DATA;
1153        TPMI_DH_SAVED_mst               TPMI_DH_SAVED_DATA;
1154        TPMI_RH_HIERARCHY_mst           TPMI_RH_HIERARCHY_DATA;
1155        TPMI_RH_ENABLES_mst             TPMI_RH_ENABLES_DATA;
1156        TPMI_RH_HIERARCHY_AUTH_mst      TPMI_RH_HIERARCHY_AUTH_DATA;
1157        TPMI_RH_HIERARCHY_POLICY_mst    TPMI_RH_HIERARCHY_POLICY_DATA;
1158        TPMI_RH_PLATFORM_mst            TPMI_RH_PLATFORM_DATA;
1159        TPMI_RH_OWNER_mst               TPMI_RH_OWNER_DATA;
1160        TPMI_RH_ENDORSEMENT_mst         TPMI_RH_ENDORSEMENT_DATA;
1161        TPMI_RH_PROVISION_mst           TPMI_RH_PROVISION_DATA;
1162        TPMI_RH_CLEAR_mst               TPMI_RH_CLEAR_DATA;
1163        TPMI_RH_NV_AUTH_mst             TPMI_RH_NV_AUTH_DATA;
1164        TPMI_RH_LOCKOUT_mst             TPMI_RH_LOCKOUT_DATA;
1165        TPMI_RH_NV_INDEX_mst            TPMI_RH_NV_INDEX_DATA;
1166        TPMI_RH_AC_mst                  TPMI_RH_AC_DATA;
1167        TPMI_RH_ACT_mst                 TPMI_RH_ACT_DATA;
1168        TPMI_ALG_HASH_mst               TPMI_ALG_HASH_DATA;
1169        TPMI_ALG_ASYM_mst               TPMI_ALG_ASYM_DATA;
1170        TPMI_ALG_SYM_mst                TPMI_ALG_SYM_DATA;
1171        TPMI_ALG_SYM_OBJECT_mst         TPMI_ALG_SYM_OBJECT_DATA;
1172        TPMI_ALG_SYM_MODE_mst           TPMI_ALG_SYM_MODE_DATA;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1173        TPMI_ALG_KDF_mst                 TPMI_ALG_KDF_DATA;
1174        TPMI_ALG_SIG_SCHEME_mst          TPMI_ALG_SIG_SCHEME_DATA;
1175        TPMI_ECC_KEY_EXCHANGE_mst        TPMI_ECC_KEY_EXCHANGE_DATA;
1176        TPMI_ST_COMMAND_TAG_mst          TPMI_ST_COMMAND_TAG_DATA;
1177        TPMI_ALG_MAC_SCHEME_mst          TPMI_ALG_MAC_SCHEME_DATA;
1178        TPMI_ALG_CIPHER_MODE_mst         TPMI_ALG_CIPHER_MODE_DATA;
1179        TPMS_EMPTY_mst                   TPMS_EMPTY_DATA;
1180        TPMS_ALGORITHM_DESCRIPTION_mst   TPMS_ALGORITHM_DESCRIPTION_DATA;
1181        TPMU_HA_mst                      TPMU_HA_DATA;
1182        TPMT_HA_mst                      TPMT_HA_DATA;
1183        Tpm2bMarshal_mst                 TPM2B_DIGEST_DATA;
1184        Tpm2bMarshal_mst                 TPM2B_DATA_DATA;
1185        Tpm2bMarshal_mst                 TPM2B_EVENT_DATA;
1186        Tpm2bMarshal_mst                 TPM2B_MAX_BUFFER_DATA;
1187        Tpm2bMarshal_mst                 TPM2B_MAX_NV_BUFFER_DATA;
1188        Tpm2bMarshal_mst                 TPM2B_TIMEOUT_DATA;
1189        Tpm2bMarshal_mst                 TPM2B_IV_DATA;
1190        NullUnionMarshal_mst             NULL_UNION_DATA;
1191        Tpm2bMarshal_mst                 TPM2B_NAME_DATA;
1192        TPMS_PCR_SELECT_mst              TPMS_PCR_SELECT_DATA;
1193        TPMS_PCR_SELECTION_mst           TPMS_PCR_SELECTION_DATA;
1194        TPMT_TK_CREATION_mst             TPMT_TK_CREATION_DATA;
1195        TPMT_TK_VERIFIED_mst             TPMT_TK_VERIFIED_DATA;
1196        TPMT_TK_AUTH_mst                 TPMT_TK_AUTH_DATA;
1197        TPMT_TK_HASHCHECK_mst            TPMT_TK_HASHCHECK_DATA;
1198        TPMS_ALG_PROPERTY_mst            TPMS_ALG_PROPERTY_DATA;
1199        TPMS_TAGGED_PROPERTY_mst         TPMS_TAGGED_PROPERTY_DATA;
1200        TPMS_TAGGED_PCR_SELECT_mst       TPMS_TAGGED_PCR_SELECT_DATA;
1201        TPMS_TAGGED_POLICY_mst           TPMS_TAGGED_POLICY_DATA;
1202        TPMS_ACT_DATA_mst                TPMS_ACT_DATA_DATA;
1203        ListMarshal_mst                  TPML_CC_DATA;
1204        ListMarshal_mst                  TPML_CCA_DATA;
1205        ListMarshal_mst                  TPML_ALG_DATA;
1206        ListMarshal_mst                  TPML_HANDLE_DATA;
1207        ListMarshal_mst                  TPML_DIGEST_DATA;
1208        ListMarshal_mst                  TPML_DIGEST_VALUES_DATA;
1209        ListMarshal_mst                  TPML_PCR_SELECTION_DATA;
1210        ListMarshal_mst                  TPML_ALG_PROPERTY_DATA;
1211        ListMarshal_mst                  TPML_TAGGED_TPM_PROPERTY_DATA;
1212        ListMarshal_mst                  TPML_TAGGED_PCR_PROPERTY_DATA;
1213        ListMarshal_mst                  TPML_ECC_CURVE_DATA;
1214        ListMarshal_mst                  TPML_TAGGED_POLICY_DATA;
1215        ListMarshal_mst                  TPML_ACT_DATA_DATA;
1216        TPMU_CAPABILITIES_mst            TPMU_CAPABILITIES_DATA;
1217        TPMS_CAPABILITY_DATA_mst         TPMS_CAPABILITY_DATA_DATA;
1218        TPMS_CLOCK_INFO_mst              TPMS_CLOCK_INFO_DATA;
1219        TPMS_TIME_INFO_mst               TPMS_TIME_INFO_DATA;
1220        TPMS_TIME_ATTEST_INFO_mst        TPMS_TIME_ATTEST_INFO_DATA;
1221        TPMS_CERTIFY_INFO_mst            TPMS_CERTIFY_INFO_DATA;
1222        TPMS_QUOTE_INFO_mst              TPMS_QUOTE_INFO_DATA;
1223        TPMS_COMMAND_AUDIT_INFO_mst      TPMS_COMMAND_AUDIT_INFO_DATA;
1224        TPMS_SESSION_AUDIT_INFO_mst      TPMS_SESSION_AUDIT_INFO_DATA;
1225        TPMS_CREATION_INFO_mst           TPMS_CREATION_INFO_DATA;
1226        TPMS_NV_CERTIFY_INFO_mst         TPMS_NV_CERTIFY_INFO_DATA;
1227        TPMS_NV_DIGEST_CERTIFY_INFO_mst TPMS_NV_DIGEST_CERTIFY_INFO_DATA;
1228        TPMI_ST_ATTEST_mst               TPMI_ST_ATTEST_DATA;
1229        TPMU_ATTEST_mst                  TPMU_ATTEST_DATA;
1230        TPMS_ATTEST_mst                  TPMS_ATTEST_DATA;
1231        Tpm2bMarshal_mst                 TPM2B_ATTEST_DATA;
1232        TPMS_AUTH_COMMAND_mst            TPMS_AUTH_COMMAND_DATA;
1233        TPMS_AUTH_RESPONSE_mst           TPMS_AUTH_RESPONSE_DATA;
1234        TPMI_TDES_KEY_BITS_mst           TPMI_TDES_KEY_BITS_DATA;
1235        TPMI_AES_KEY_BITS_mst            TPMI_AES_KEY_BITS_DATA;
1236        TPMI_SM4_KEY_BITS_mst            TPMI_SM4_KEY_BITS_DATA;
1237        TPMI_CAMELLIA_KEY_BITS_mst       TPMI_CAMELLIA_KEY_BITS_DATA;
1238        TPMU_SYM_KEY_BITS_mst            TPMU_SYM_KEY_BITS_DATA;
```

 Copyright © TCG 2006-2020

```
1239      TPMU_SYM_MODE_mst                TPMU_SYM_MODE_DATA;
1240      TPMT_SYM_DEF_mst                 TPMT_SYM_DEF_DATA;
1241      TPMT_SYM_DEF_OBJECT_mst          TPMT_SYM_DEF_OBJECT_DATA;
1242      Tpm2bMarshal_mst                 TPM2B_SYM_KEY_DATA;
1243      TPMS_SYMCIPHER_PARMS_mst         TPMS_SYMCIPHER_PARMS_DATA;
1244      Tpm2bMarshal_mst                 TPM2B_LABEL_DATA;
1245      TPMS_DERIVE_mst                  TPMS_DERIVE_DATA;
1246      Tpm2bMarshal_mst                 TPM2B_DERIVE_DATA;
1247      Tpm2bMarshal_mst                 TPM2B_SENSITIVE_DATA_DATA;
1248      TPMS_SENSITIVE_CREATE_mst        TPMS_SENSITIVE_CREATE_DATA;
1249      Tpm2bsMarshal_mst                TPM2B_SENSITIVE_CREATE_DATA;
1250      TPMS_SCHEME_HASH_mst             TPMS_SCHEME_HASH_DATA;
1251      TPMS_SCHEME_ECDAA_mst            TPMS_SCHEME_ECDAA_DATA;
1252      TPMI_ALG_KEYEDHASH_SCHEME_mst    TPMI_ALG_KEYEDHASH_SCHEME_DATA;
1253      TPMS_SCHEME_XOR_mst              TPMS_SCHEME_XOR_DATA;
1254      TPMU_SCHEME_KEYEDHASH_mst        TPMU_SCHEME_KEYEDHASH_DATA;
1255      TPMT_KEYEDHASH_SCHEME_mst        TPMT_KEYEDHASH_SCHEME_DATA;
1256      TPMU_SIG_SCHEME_mst              TPMU_SIG_SCHEME_DATA;
1257      TPMT_SIG_SCHEME_mst              TPMT_SIG_SCHEME_DATA;
1258      TPMU_KDF_SCHEME_mst              TPMU_KDF_SCHEME_DATA;
1259      TPMT_KDF_SCHEME_mst              TPMT_KDF_SCHEME_DATA;
1260      TPMI_ALG_ASYM_SCHEME_mst         TPMI_ALG_ASYM_SCHEME_DATA;
1261      TPMU_ASYM_SCHEME_mst             TPMU_ASYM_SCHEME_DATA;
1262      TPMI_ALG_RSA_SCHEME_mst          TPMI_ALG_RSA_SCHEME_DATA;
1263      TPMT_RSA_SCHEME_mst              TPMT_RSA_SCHEME_DATA;
1264      TPMI_ALG_RSA_DECRYPT_mst         TPMI_ALG_RSA_DECRYPT_DATA;
1265      TPMT_RSA_DECRYPT_mst             TPMT_RSA_DECRYPT_DATA;
1266      Tpm2bMarshal_mst                 TPM2B_PUBLIC_KEY_RSA_DATA;
1267      TPMI_RSA_KEY_BITS_mst            TPMI_RSA_KEY_BITS_DATA;
1268      Tpm2bMarshal_mst                 TPM2B_PRIVATE_KEY_RSA_DATA;
1269      Tpm2bMarshal_mst                 TPM2B_ECC_PARAMETER_DATA;
1270      TPMS_ECC_POINT_mst               TPMS_ECC_POINT_DATA;
1271      Tpm2bsMarshal_mst                TPM2B_ECC_POINT_DATA;
1272      TPMI_ALG_ECC_SCHEME_mst          TPMI_ALG_ECC_SCHEME_DATA;
1273      TPMI_ECC_CURVE_mst               TPMI_ECC_CURVE_DATA;
1274      TPMT_ECC_SCHEME_mst              TPMT_ECC_SCHEME_DATA;
1275      TPMS_ALGORITHM_DETAIL_ECC_mst    TPMS_ALGORITHM_DETAIL_ECC_DATA;
1276      TPMS_SIGNATURE_RSA_mst           TPMS_SIGNATURE_RSA_DATA;
1277      TPMS_SIGNATURE_ECC_mst           TPMS_SIGNATURE_ECC_DATA;
1278      TPMU_SIGNATURE_mst               TPMU_SIGNATURE_DATA;
1279      TPMT_SIGNATURE_mst               TPMT_SIGNATURE_DATA;
1280      TPMU_ENCRYPTED_SECRET_mst        TPMU_ENCRYPTED_SECRET_DATA;
1281      Tpm2bMarshal_mst                 TPM2B_ENCRYPTED_SECRET_DATA;
1282      TPMI_ALG_PUBLIC_mst              TPMI_ALG_PUBLIC_DATA;
1283      TPMU_PUBLIC_ID_mst               TPMU_PUBLIC_ID_DATA;
1284      TPMS_KEYEDHASH_PARMS_mst         TPMS_KEYEDHASH_PARMS_DATA;
1285      TPMS_RSA_PARMS_mst               TPMS_RSA_PARMS_DATA;
1286      TPMS_ECC_PARMS_mst               TPMS_ECC_PARMS_DATA;
1287      TPMU_PUBLIC_PARMS_mst            TPMU_PUBLIC_PARMS_DATA;
1288      TPMT_PUBLIC_PARMS_mst            TPMT_PUBLIC_PARMS_DATA;
1289      TPMT_PUBLIC_mst                  TPMT_PUBLIC_DATA;
1290      Tpm2bsMarshal_mst                TPM2B_PUBLIC_DATA;
1291      Tpm2bMarshal_mst                 TPM2B_TEMPLATE_DATA;
1292      Tpm2bMarshal_mst                 TPM2B_PRIVATE_VENDOR_SPECIFIC_DATA;
1293      TPMU_SENSITIVE_COMPOSITE_mst     TPMU_SENSITIVE_COMPOSITE_DATA;
1294      TPMT_SENSITIVE_mst               TPMT_SENSITIVE_DATA;
1295      Tpm2bsMarshal_mst                TPM2B_SENSITIVE_DATA;
1296      Tpm2bMarshal_mst                 TPM2B_PRIVATE_DATA;
1297      Tpm2bMarshal_mst                 TPM2B_ID_OBJECT_DATA;
1298      TPMS_NV_PIN_COUNTER_PARAMETERS_mst TPMS_NV_PIN_COUNTER_PARAMETERS_DATA;
1299      AttributesMarshal_mst            TPMA_NV_DATA;
1300      TPMS_NV_PUBLIC_mst               TPMS_NV_PUBLIC_DATA;
1301      Tpm2bsMarshal_mst                TPM2B_NV_PUBLIC_DATA;
1302      Tpm2bMarshal_mst                 TPM2B_CONTEXT_SENSITIVE_DATA;
1303      TPMS_CONTEXT_DATA_mst            TPMS_CONTEXT_DATA_DATA;
1304      Tpm2bMarshal_mst                 TPM2B_CONTEXT_DATA_DATA;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1305        TPMS_CONTEXT_mst              TPMS_CONTEXT_DATA;
1306        TPMS_CREATION_DATA_mst        TPMS_CREATION_DATA_DATA;
1307        Tpm2bsMarshal_mst             TPM2B_CREATION_DATA_DATA;
1308        TPM_AT_mst                    TPM_AT_DATA;
1309        TPMS_AC_OUTPUT_mst            TPMS_AC_OUTPUT_DATA;
1310        ListMarshal_mst               TPML_AC_CAPABILITIES_DATA;
1311        MinMaxMarshal_mst             Type00_DATA;
1312        MinMaxMarshal_mst             Type01_DATA;
1313        Type02_mst                    Type02_DATA;
1314        Type03_mst                    Type03_DATA;
1315        Type04_mst                    Type04_DATA;
1316        MinMaxMarshal_mst             Type05_DATA;
1317        Type06_mst                    Type06_DATA;
1318        MinMaxMarshal_mst             Type07_DATA;
1319        Type08_mst                    Type08_DATA;
1320        Type10_mst                    Type10_DATA;
1321        Type11_mst                    Type11_DATA;
1322        Type12_mst                    Type12_DATA;
1323        Type13_mst                    Type13_DATA;
1324        Type15_mst                    Type15_DATA;
1325        Type17_mst                    Type17_DATA;
1326        Type18_mst                    Type18_DATA;
1327        Type19_mst                    Type19_DATA;
1328        Type20_mst                    Type20_DATA;
1329        Type22_mst                    Type22_DATA;
1330        Type23_mst                    Type23_DATA;
1331        Type24_mst                    Type24_DATA;
1332        Type25_mst                    Type25_DATA;
1333        Type26_mst                    Type26_DATA;
1334        Type27_mst                    Type27_DATA;
1335        MinMaxMarshal_mst             Type28_DATA;
1336        Type29_mst                    Type29_DATA;
1337        Type30_mst                    Type30_DATA;
1338        MinMaxMarshal_mst             Type31_DATA;
1339        MinMaxMarshal_mst             Type32_DATA;
1340        Type33_mst                    Type33_DATA;
1341        Type34_mst                    Type34_DATA;
1342        Type35_mst                    Type35_DATA;
1343        MinMaxMarshal_mst             Type36_DATA;
1344        MinMaxMarshal_mst             Type37_DATA;
1345        Type38_mst                    Type38_DATA;
1346        MinMaxMarshal_mst             Type39_DATA;
1347        MinMaxMarshal_mst             Type40_DATA;
1348        Type41_mst                    Type41_DATA;
1349        Type42_mst                    Type42_DATA;
1350        MinMaxMarshal_mst             Type43_DATA;
1351        Type44_mst                    Type44_DATA;
1352    } MarshalData_st;
1353    #endif // _TABLE_MARSHAL_TYPES_H_
```

9.10.8 Table Marshal Source

9.10.8.1 TableDrivenMarshal.c

```
1   #include <assert.h>
2   #include "Tpm.h"
3   #include "Marshal.h"
4   #include "TableMarshal.h"
5   #if TABLE_DRIVEN_MARSHAL
6   extern ArrayMarshal_mst ArrayLookupTable[];
7
8   extern UINT16 MarshalLookupTable[];
9
10  typedef struct { int a; } External_Structure_t;
```

 Copyright © TCG 2006-2020

```
11
12  extern struct Exernal_Structure_t MarshalData;
13
14  #define IS_SUCCESS(UNMARSHAL_FUNCTION)                                        \
15      (TPM_RC_SUCCESS == (result = (UNMARSHAL_FUNCTION)))
16  marshalIndex_t IntegerDispatch[] = {
17      UINT8_MARSHAL_REF, UINT16_MARSHAL_REF, UINT32_MARSHAL_REF, UINT64_MARSHAL_REF,
18      INT8_MARSHAL_REF,  INT16_MARSHAL_REF,  INT32_MARSHAL_REF,  INT64_MARSHAL_REF
19  };
20
21  #if 1
22  #define GetDescriptor(reference)                                              \
23      ((MarshalHeader_mst *)(((BYTE *)(&MarshalData)) + (reference & NULL_MASK)))
24  #else
25  static const MarshalHeader_mst *GetDescriptor(marshalIndex_t index)
26  {
27      const MarshalHeader_mst    *mst = MarshalLookupTable[index & NULL_MASK];
28      return mst;
29  }
30  #endif
31  #define GetUnionDescriptor(_index_)                                           \
32      ((UnionMarshal_mst *)GetDescriptor(_index_))
33  #define GetArrayDescriptor(_index_)                                           \
34      ((ArrayMarshal_mst *))ArrayLookupTable[_index_ & NULL_MASK])
35
36  //*** GetUnmarshaledInteger()
37  // Gets the unmarshaled value and normalizes it to a UIN32 for other
38  // processing (comparisons and such).
39  static UINT32 GetUnmarshaledInteger(
40      marshalIndex_t       type,
41      const void          *target
42  )
43  {
44      int      size = (type & SIZE_MASK);
45  //
46      if(size == FOUR_BYTES)
47          return *((UINT32 *)target);
48      if(type & IS_SIGNED)
49      {
50          if(size == TWO_BYTES)
51              return (UINT32)*((int16_t *)target);
52          return (UINT32)*((int8_t *)target);
53      }
54      if(size == TWO_BYTES)
55          return (UINT32)*((UINT16 *)target);
56      return (UINT32)*((UINT8 *)target);
57  }
58  static UINT32 GetSelector(
59      void                 *structure,
60      const UINT16         *values,
61      UINT16                descriptor
62  )
63  {
64      uint                  sel = GET_ELEMENT_NUMBER(descriptor);
65      // Get the offset of the value in the unmarshaled structure
66      const UINT16         *entry = &values[(sel * 3)];
67  //
68      return GetUnmarshaledInteger(GET_ELEMENT_SIZE(entry[0]),
69                           ((UINT8 *)structure) + entry[2]);
70  }
71  static TPM_RC UnmarshalBytes(
72      UINT8                *target,         // IN/OUT: place to put the bytes
73      UINT8                **buffer,        // IN/OUT: source of the input data
74      INT32                *size,           // IN/OUT: remaining bytes in the input buffer
75      int                   count           // IN: number of bytes to get
```


 Copyright © TCG 2006-2020

```
76  )
77  {
78      if((*size -= count) >= 0)
79      {
80          memcpy(target, *buffer, count);
81          *buffer += count;
82          return TPM_RC_SUCCESS;
83      }
84      return TPM_RC_INSUFFICIENT;
85  }
```

9.10.8.1.1.1 MarshalBytes()

Marshal an array of bytes.

```
 86  static UINT16 MarshalBytes(
 87      UINT8               *source,
 88      UINT8               **buffer,
 89      INT32               *size,
 90      int32_t              count
 91  )
 92  {
 93      if(buffer != NULL)
 94      {
 95          if(size != NULL && (size -= count) < 0)
 96              return 0;
 97          memcpy(*buffer, source, count);
 98          *buffer += count;
 99      }
100      return (UINT16)count;
101  }
```

9.10.8.1.1.2 ArrayUnmarshal()

Unmarshal an array. The *index* is of the form: *type*_ARRAY_MARSHAL_INDEX.

```
102 static TPM_RC ArrayUnmarshal(
103     UINT16              index,          // IN: the type of the array
104     UINT8              *target,         // IN: target for the data
105     UINT8             **buffer,         // IN/OUT: place to get the data
106     INT32              *size,           // IN/OUT: remaining unmarshal data
107     UINT32              count           // IN: number of values of 'index' to
108                                         //       unmarshal
109 )
110 {
111     marshalIndex_t      which = ArrayLookupTable[index & NULL_MASK].type;
112     UINT16              stride = ArrayLookupTable[index & NULL_MASK].stride;
113     TPM_RC              result;
114 //
115     if(stride == 1) // A byte array
116         result = UnmarshalBytes(target, buffer, size, count);
117     else
118     {
119         which |= index & NULL_FLAG;
120         for(result = TPM_RC_SUCCESS; count > 0; target += stride, count--)
121             if(!IS_SUCCESS(Unmarshal(which, target, buffer, size)))
122                 break;
123     }
124     return result;
125 }
```

 Copyright © TCG 2006-2020

9.10.8.1.1.3 ArrayMarshal()

```
126  static UINT16 ArrayMarshal(
127      UINT16               index,          // IN: the type of the array
128      UINT8               *source,         // IN: source of the data
129      UINT8              **buffer,         // IN/OUT: place to put the data
130      INT32               *size,           // IN/OUT: amount of space for the data
131      UINT32               count           // IN: number of values of 'index' to marshal
132  )
133  {
134      marshalIndex_t      which = ArrayLookupTable[index & NULL_MASK].type;
135      UINT16              stride = ArrayLookupTable[index & NULL_MASK].stride;
136      UINT16              retVal;
137  //
138      if(stride == 1) // A byte array
139          return MarshalBytes(source, buffer, size, count);
140      which |= index & NULL_FLAG;
141      for(retVal = 0
142          ; count > 0
143          ; source += stride, count--)
144          retVal += Marshal(which, source, buffer, size);
145
146      return retVal;
147  }
```


 Copyright © TCG 2006-2020

9.10.8.1.1.4 UnmarshalUnion()

```
148 TPM_RC
149 UnmarshalUnion(
150     UINT16              typeIndex,          // IN: the thing to unmarshal
151     void               *target,             // IN: were the data goes to
152     UINT8              **buffer,            // IN/OUT: the data source buffer
153     INT32              *size,               // IN/OUT: the remaining size
154     UINT32              selector
155 )
156 {
157     int                 i;
158     UnionMarshal_mst   *ut = GetUnionDescriptor(typeIndex);
159     marshalIndex_t      selected;
160 //
161     for(i = 0; i < ut->countOfselectors; i++)
162     {
163         if(selector == ut->selectors[i])
164         {
165             UINT8           *offset = ((UINT8 *)ut) + ut->offsetOfUnmarshalTypes;
166             // Get the selected thing to unmarshal
167             selected = ((marshalIndex_t *)offset)[i];
168             if(ut->modifiers & IS_ARRAY_UNION)
169                 return UnmarshalBytes(target, buffer, size, selected);
170             else
171             {
172                 // Propagate NULL_FLAG if the null flag was
173                 // propagated to the structure containing the union
174                 selected |= (typeIndex & NULL_FLAG);
175                 return Unmarshal(selected, target, buffer, size);
176             }
177         }
178     }
179     // Didn't find the value.
180     return TPM_RC_SELECTOR;
181 }
```

9.10.8.1.1.5 MarshalUnion()

```
182 UINT16
183 MarshalUnion(
184     UINT16              typeIndex,          // IN: the thing to marshal
185     void               *source,             // IN: were the data comes from
186     UINT8             **buffer,             // IN/OUT: the data source buffer
187     INT32              *size,               // IN/OUT: the remaining size
188     UINT32              selector            // IN: the union selector
189 )
190 {
191     int                 i;
192     UnionMarshal_mst   *ut = GetUnionDescriptor(typeIndex);
193     marshalIndex_t      selected;
194 //
195     for(i = 0; i < ut->countOfselectors; i++)
196     {
197         if(selector == ut->selectors[i])
198         {
199             UINT8           *offset = ((UINT8 *)ut) + ut->offsetOfUnmarshalTypes;
200             // Get the selected thing to unmarshal
201             selected = ((marshalIndex_t *)offset)[i];
202             if(ut->modifiers & IS_ARRAY_UNION)
203                 return MarshalBytes(source, buffer, size, selected);
204             else
205                 return Marshal(selected, source, buffer, size);
206         }
207     }
208     if(size != NULL)
209         *size = -1;
210     return 0;
211 }
212 TPM_RC
213 UnmarshalInteger(
214     int                 iSize,              // IN: Number of bytes in the integer
215     void               *target,             // OUT: receives the integer
216     UINT8             **buffer,             // IN/OUT: source of the data
217     INT32              *size,               // IN/OUT: amount of data available
218     UINT32             *value               // OUT: optional copy of 'target'
219 )
220 {
221     // This is just to save typing
222 #define _MB_ (*buffer)
223     // The size is a power of two so convert to regular integer
224     int             bytes = (1 << (iSize & SIZE_MASK));
225 //
226     // Check to see if there is enough data to fulfill the request
227     if((*size -= bytes) >= 0)
228     {
229         // The most comon size
230         if(bytes == 4)
231         {
232             *((UINT32 *)target) = (UINT32)((((((_MB_[0] << 8) | _MB_[1]) << 8)
233                                         | _MB_[2]) << 8) | _MB_[3]);
234             // If a copy is needed, copy it.
235             if(value != NULL)
236                 *value = *((UINT32 *)target);
237         }
238         else if(bytes == 2)
239         {
240             *((UINT16 *)target) = (UINT16)((_MB_[0] << 8) | _MB_[1]);
241             // If a copy is needed, copy with the appropriate sign extension
242             if(value != NULL)
243             {
244                 if(iSize & IS_SIGNED)
```

```
245                         *value = (UINT32)(*((INT16 *)target));
246                     else
247                         *value = (UINT32)(*((UINT16 *)target));
248                 }
249         }
250         else if(bytes == 1)
251         {
252             *((UINT8*)target) = (UINT8)_MB_[0];
253             // If a copy is needed, copy with the appropriate sign extension
254             if(value != NULL)
255             {
256                 if(iSize & IS_SIGNED)
257                     *value = (UINT32)(*((INT8 *)target));
258                 else
259                     *value = (UINT32)(*((UINT8 *)target));
260             }
261         }
262         else
263         {
264             // There is no input type that is a 64-bit value other than a UINT64. So
265             // there is no reason to do anything other than unmarshal it.
266             *((UINT64 *)target) = BYTE_ARRAY_TO_UINT64(*buffer);
267         }
268         *buffer += bytes;
269         return TPM_RC_SUCCESS;
270 #undef _MB_
271     }
272     return TPM_RC_INSUFFICIENT;
273 }
```

 Copyright © TCG 2006-2020

9.10.8.1.1.6 Unmarshal()

This is the function that performs unmarshaling of different numbered types. Each TPM type has a number. The number is used to lookup the address of the data structure that describes how to unmarshal that data type.

```
274 TPM_RC
275 Unmarshal(
276     UINT16              typeIndex,          // IN: the thing to marshal
277     void                *target,            // IN: were the data goes from
278     UINT8               **buffer,           // IN/OUT: the data source buffer
279     INT32               *size               // IN/OUT: the remaining size
280 )
281 {
282     const MarshalHeader_mst     *sel;
283     TPM_RC                       result;
284 //
285 #define _target     ((UINT8 *)target)
286     sel = GetDescriptor(typeIndex);
287     switch(sel->marshalType)
288     {
289         case UINT_MTYPE:
290         {
291             // A simple signed or unsigned integer value.
292             return UnmarshalInteger(sel->modifiers, target,
293                                     buffer, size, NULL);
294         }
295         case VALUES_MTYPE:
296         {
297             // This is the general-purpose structure that can handle things like
298             // TPMI_DH_PARENT that has multiple ranges, multiple singles and a
299             // 'null' value. When things cover a large range with holes in the range
300             // they can be turned into multiple ranges. There is no option for a bit
301             // field.
302             // The structure is:
303             //  typedef const struct ValuesMarshal_mst
304             //  {
305             //      UINT8           marshalType;        // VALUES_MTYPE
306             //      UINT8           modifiers;
307             //      UINT8           errorCode;
308             //      UINT8           ranges;
309             //      UINT8           singles;
310             //      UINT32          values[1];
311             //  } ValuesMarshal_mst;
312             // Unmarshal the base type
313             UINT32                  val;
314             if(IS_SUCCESS(UnmarshalInteger(sel->modifiers, target,
315                                            buffer, size, &val)))
316             {
317                 ValuesMarshal_mst   *vmt = ((ValuesMarshal_mst *)sel);
318                 const UINT32        *check = vmt->values;
319             //
320                 // if the TAKES_NULL flag is set, then the first entry in the values
321                 // list is the NULL value. Iy is not included in the 'ranges' or
322                 // 'singles' count.
323                 if((vmt->modifiers & TAKES_NULL) && (val == *check++))
324                 {
325                     if((typeIndex & NULL_FLAG) == 0)
326                         result = (TPM_RC)(sel->errorCode);
327                 }
328                 // No NULL value or input is not the NULL value
329                 else
330                 {
331                     int             i;
```

```
332                 //
333                     // Check all the min-max ranges.
334                     for(i = vmt->ranges - 1; i >= 0; check = &check[2], i--)
335                         if((UINT32)(val - check[0]) <= check[1])
336                             break;
337                     // if the input is in a selected range, then i >= 0
338                     if(i < 0)
339                     {
340                         // Not in any range, so check sigles
341                         for(i = vmt->singles - 1; i >= 0; i--)
342                             if(val == check[i])
343                                 break;
344                     }
345                     // If input not in range and not in any single so return error
346                     if(i < 0)
347                         result = (TPM_RC)(sel->errorCode);
348                 }
349             }
350             break;
351         }
352         case TABLE_MTYPE:
353         {
354             // This is a table with or without bit checking. The input is checked
355             // against each value in the table. If the value is in the table, and
356             // a bits table is present, then the bit field is checked to see if the
357             // indiated value is implemented. For example, if there is a table of
358             // allowed RSA key sises and the 2nd entry matches, then the 2nd bit in
359             // thed bit field is check to see if that allowed size is implemented in
360             // this TPM.
361             //  typedef const struct TableMarshal_mst
362             //  {
363             //      UINT8           marshalType;        // TABLE_MTYPE
364             //      UINT8           modifiers;
365             //      UINT8           errorCode;
366             //      UINT8           singles;
367             //      UINT32          values[1];
368             //  } TableMarshal_mst;
369
370             UINT32                  val;
371         //
372             // Unmarshal the base type
373             if(IS_SUCCESS(UnmarshalInteger(sel->modifiers, target,
374                                            buffer, size, &val)))
375             {
376                 TableMarshal_mst    *tmt = ((TableMarshal_mst *)sel);
377                 const UINT32        *check = tmt->values;
378             //
379                 // If this type has a null value, then it is the first value in the
380                 // list of values. It does not count in the count of values
381                 if((tmt->modifiers & TAKES_NULL) && (val == *check++))
382                 {
383                     if((typeIndex & NULL_FLAG) == 0)
384                         result = (TPM_RC)(sel->errorCode);
385                 }
386                 else
387                 {
388                     int                i;
389                 //
390                     // Process the singles
391                     for(i = tmt->singles - 1; i >= 0; i--)
392                     {
393                         // does the input value match the value in the table
394                         if(val == check[i])
395                         {
396                             // If there is an associated bit table, make sure that the
corresponding
```

```
397                             // bit is SET
398                             if((HAS_BITS & tmt->modifiers)
399                                && (!IS_BIT_SET32(i, &(check[tmt->singles]))))
400                                 // if not SET, then this is a failure.
401                                 i = -1;
402                             break;
403                         }
404                     }
405                     // error if not found or bit not SET
406                     if(i < 0)
407                         result = (TPM_RC)(sel->errorCode);
408                 }
409             }
410             break;
411         }
412         case MIN_MAX_MTYPE:
413         {
414             // A MIN_MAX is a range. It can have a bit field and a NULL value that is
415             // outside of the range. If the input value is in the min-max range then
416             // it is valid unless there is an associated bit field. Otherwise, it
417             // it is only valid if the corresponding value in the bit field is SET.
418             // The min value is 'values[0]' or 'values[1]' if there is a NULL value.
419             // The max value is the value after min. The max value is in the table as
420             // max minus min. This allows 'val' to be subtracted from min and then
421             // checked against max with one unsigned comparison. If present, the bit
422             // field will be the first 'values' after max.
423             //  typedef const struct MinMaxMarshal_mst
424             //  {
425             //      UINT8           marshalType;        // MIN_MAX_MTYPE
426             //      UINT8           modifiers;
427             //      UINT8           errorCode;
428             //      UINT32          values[2];
429             //  } MinMaxMarshal_mst;
430             UINT32              val;
431         //
432             // A min-max has a range. It can have a bit-field that is indexed to the
433             // min value (something that matches min has a bit at 0. This is useful
434             // for algorithms. The min-max define a range of algorithms to be checked
435             // and the bit field can check to see if the algorithm in that range is
436             // allowed.
437             if(IS_SUCCESS(UnmarshalInteger(sel->modifiers, target,
438                                            buffer, size, &val)))
439             {
440                 MinMaxMarshal_mst   *mmt = (MinMaxMarshal_mst *)sel;
441                 const UINT32        *check = mmt->values;
442             //
443                 // If this type takes a NULL, see if it matches. This
444                 if((mmt->modifiers & TAKES_NULL) && (val == *check++))
445                 {
446                     if((typeIndex & NULL_FLAG) == 0)
447                         result = (TPM_RC)(mmt->errorCode);
448                 }
449                 else
450                 {
451                     val -= *check;
452                     if((val > check[1])
453                        || ((mmt->modifiers & HAS_BITS) &&
454                            !IS_BIT_SET32(val, &check[2])))
455                         result = (TPM_RC)(mmt->errorCode);
456                 }
457             }
458             break;
459         }
460         case ATTRIBUTES_MTYPE:
461         {
462             //  This is used for TPMA values.
```

```
463                 UINT32                   mask;
464                 AttributesMarshal_mst   *amt = (AttributesMarshal_mst *)sel;
465             //
466                 if(IS_SUCCESS(UnmarshalInteger(sel->modifiers, target,
467                                         buffer, size, &mask)))
468                 {
469                     if((mask & amt->attributeMask) != 0)
470                         result = TPM_RC_RESERVED_BITS;
471                 }
472                 break;
473             }
474             case STRUCTURE_MTYPE:
475             {
476                 // A structure (not a union). A structure has elements (one defined per
477                 // row). Three UINT16 values are used for each row. The first indicates
478                 // the type of the entry. They choices are: simple, union, or array. A
479                 // simple type can be a simple integer or another structure. It can also
480                 // be a specific "interface type." For example, when a structure entry is
481                 // a value that is used define the dimension of an array, the entry of
482                 // the structure will reference a "synthetic" interface type, most often
483                 // a min-max value. If the type of the entry is union or array, then the
484                 // first value indicates which of the previous elements provides the union
485                 // selector or the array dimension. That previous entry is referenced in
486                 // the unmarshaled structure in memory (Not the marshaled buffer). The
487                 // previous entry indicates the location in the structure of the value.
488                 // The second entry of each structure entry indicated the index of the
489                 // type associated with the entry. This is an index into the array of
490                 // arrays or the union table (merged with the normal table in this
491                 // implementation). The final entry is the offset in the unmarshaled
492                 // structure where the value is located. This is the offsetof(STRUcTURE,
493                 // element). This value is added to the input 'target' or 'source' value
494                 // to determine where the value goes.
495                 StructMarshal_mst   *mst = (StructMarshal_mst *)sel;
496                 int                  i;
497                 const UINT16        *value;
498             //
499                 for(result = TPM_RC_SUCCESS, value = mst->values,  i = mst->elements
500                     ; (TPM_RC_SUCCESS == result) && (i > 0)
501                     ; value = &value[3], i--)
502                 {
503                     UINT16            descriptor = value[0];
504                     marshalIndex_t    index = value[1];
505                     UINT8            *offset = _target + value[2];
506                 //
507                     index |= ((ELEMENT_PROPAGATE & descriptor)
508                                 << (NULL_SHIFT - PROPAGATE_SHIFT));
509                     switch(GET_ELEMENT_TYPE(descriptor))
510                     {
511                         case SIMPLE_STYPE:
512                         {
513                             result = Unmarshal(index, offset, buffer, size);
514                             break;
515                         }
516                         case UNION_STYPE:
517                         {
518                             UINT32                choice;
519                         //
520                             // Get the selector or array dimension value
521                             choice = GetSelector(target, mst->values, descriptor);
522                             result = UnmarshalUnion(index, offset, buffer, size, choice);
523                             break;
524                         }
525                         case ARRAY_STYPE:
526                         {
527                             UINT32             dimension;
528                         //
```

```
529                         dimension = GetSelector(target, mst->values, descriptor);
530                         result = ArrayUnmarshal(index, offset, buffer,
531                                                 size, dimension);
532                         break;
533                     }
534                     default:
535                         result = TPM_RC_FAILURE;
536                         break;
537                 }
538             }
539             break;
540         }
541         case TPM2B_MTYPE:
542         {
543             // A primitive TPM2B. A size and byte buffer. The single value (other than
544             // the tag) references the synthetic 'interface' value for the size
545             // parameter.
546             Tpm2bMarshal_mst        *m2bt = (Tpm2bMarshal_mst *)sel;
547         //
548             if(IS_SUCCESS(Unmarshal(m2bt->sizeIndex, target, buffer, size)))
549                 result = UnmarshalBytes(((TPM2B *)target)->buffer,
550                                buffer, size, *((UINT16 *)target));
551             break;
552         }
553         case TPM2BS_MTYPE:
554         {
555             // This is used when a TPM2B contains a structure.
556             Tpm2bsMarshal_mst    *m2bst = (Tpm2bsMarshal_mst *)sel;
557             INT32           count;
558         //
559             if(IS_SUCCESS(Unmarshal(m2bst->sizeIndex, target, buffer, size)))
560             {
561                 count = (int32_t)*((UINT16 *)_target);
562                 if(count == 0)
563                 {
564                     if(m2bst->modifiers & SIZE_EQUAL)
565                         result = TPM_RC_SIZE;
566                 }
567                 else if((*size -= count) >= 0)
568                 {
569                     marshalIndex_t  index = m2bst->dataIndex;
570                 //
571                     index |= (m2bst->modifiers & PROPAGATE_NULL)
572                                     << (NULL_SHIFT - PROPAGATE_SHIFT);
573                     if(IS_SUCCESS(Unmarshal(index,
574                                           _target + (m2bst->modifiers & SIGNED_MASK),
575                                           buffer, &count)))
576                     {
577                         if(count != 0)
578                             result = TPM_RC_SIZE;
579                     }
580                 }
581                 else
582                     result = TPM_RC_INSUFFICIENT;
583             }
584             break;
585         }
586         case LIST_MTYPE:
587         {
588             // Used for a list. A list is a qualified 32-bit 'count' value followed
589             // by a type indicator.
590             ListMarshal_mst     *mlt = (ListMarshal_mst *)sel;
591             marshalIndex_t      index = mlt->arrayRef;
592         //
593             if(IS_SUCCESS(Unmarshal(mlt->sizeIndex, target, buffer, size)))
594             {
```

```
595                     index |= (mlt->modifiers & PROPAGATE_NULL)
596                                 << (NULL_SHIFT - PROPAGATE_SHIFT);
597                     result = ArrayUnmarshal(index,
598                                             _target +(mlt->modifiers & SIGNED_MASK),
599                                             buffer, size,
600                                             *((UINT32 *)target));
601                 }
602                 break;
603             }
604             case NULL_MTYPE:
605             {
606                 result = TPM_RC_SUCCESS;
607                 break;
608             }
609             case COMPOSITE_MTYPE:
610             {
611                 CompositeMarshal_mst    *mct = (CompositeMarshal_mst *)sel;
612                 int                      i;
613                 UINT8                   *buf = *buffer;
614                 INT32                   sz = *size;
615             //
616                 result = TPM_RC_VALUE;
617                 for(i = GET_ELEMENT_COUNT(mct->modifiers) - 1; i <= 0; i--)
618                 {
619                     marshalIndex_t      index = mct->types[i];
620                 //
621                     // This type might take a null so set it in each called value, just
622                     // in case it is needed in that value. Only one value in each
623                     // composite should have the takes null SET.
624                     index |= typeIndex & NULL_MASK;
625                     result = Unmarshal(index, target, buffer, size);
626                     if(result == TPM_RC_SUCCESS)
627                         break;
628                     // Each of the composite values does its own unmarshaling. This
629                     // has some execution overhead if it is unmarshaled multiple times
630                     // but it saves code size in not having to reproduce the various
631                     // unmarshaling types that can be in a composite. So, what this means
632                     // is that the buffer pointer and size have to be reset for each
633                     // unmarshaled value.
634                     *buffer = buf;
635                     *size = sz;
636                 }
637                 break;
638             }
639             default:
640             {
641                 result = TPM_RC_FAILURE;
642                 break;
643             }
644         }
645         return result;
646     }
```

 Copyright © TCG 2006-2020

9.10.8.1.1.7 Marshal()

This is the function that drives marshaling of output. Because there is no validation of the output, there is a lot less code.

```
647 UINT16 Marshal(
648     UINT16              typeIndex,          // IN: the thing to marshal
649     void               *source,             // IN: were the data comes from
650     UINT8              **buffer,            // IN/OUT: the data source buffer
651     INT32              *size                // IN/OUT: the remaining size
652 )
653 {
654 #define _source   ((UINT8 *)source)
655
656     const MarshalHeader_mst     *sel;
657     UINT16                       retVal;
658 //
659     sel = GetDescriptor(typeIndex);
660     switch(sel->marshalType)
661     {
662         case VALUES_MTYPE:
663         case UINT_MTYPE:
664         case TABLE_MTYPE:
665         case MIN_MAX_MTYPE:
666         case ATTRIBUTES_MTYPE:
667         case COMPOSITE_MTYPE:
668         {
669 #if BIG_ENDIAN_TPM
670 #define MM16    0
671 #define MM32    0
672 #define MM64    0
673 #else
674 // These flip the constant index values so that they count in reverse order when doing
675 // little-endian stuff
676 #define MM16    1
677 #define MM32    3
678 #define MM64    7
679 #endif
680 // Just change the name and cast the type of the input parameters for typing purposes
681 #define mb (*buffer)
682 #define _source ((UINT8 *)source)
683             retVal = (1 << (sel->modifiers & SIZE_MASK));
684             if(buffer != NULL)
685             {
686                 if((size == NULL) || ((*size -= retVal) >= 0))
687                 {
688                     if(retVal == 4)
689                     {
690                         mb[0 ^ MM32] = _source[0];
691                         mb[1 ^ MM32] = _source[1];
692                         mb[2 ^ MM32] = _source[2];
693                         mb[3 ^ MM32] = _source[3];
694                     }
695                     else if(retVal == 2)
696                     {
697                         mb[0 ^ MM16] = _source[0];
698                         mb[1 ^ MM16] = _source[1];
699                     }
700                     else if(retVal == 1)
701                         mb[0] = _source[0];
702                     else
703                     {
704                         mb[0 ^ MM64] = _source[0];
705                         mb[1 ^ MM64] = _source[1];
```

 Copyright © TCG 2006-2020

```
706                          mb[2 ^ MM64] = _source[2];
707                          mb[3 ^ MM64] = _source[3];
708                          mb[4 ^ MM64] = _source[4];
709                          mb[5 ^ MM64] = _source[5];
710                          mb[6 ^ MM64] = _source[6];
711                          mb[7 ^ MM64] = _source[7];
712                      }
713                      *buffer += retVal;
714                  }
715              }
716              break;
717          }
718          case STRUCTURE_MTYPE:
719          {
720  //#define _mst  ((StructMarshal_mst *)sel)
721              StructMarshal_mst *mst = ((StructMarshal_mst *)sel);
722              int                i;
723              const UINT16   *value = mst->values;
724
725          //
726              for(retVal = 0, i = mst->elements; i > 0; value = &value[3], i--)
727              {
728                  UINT16           des = value[0];
729                  marshalIndex_t   index = value[1];
730                  UINT8           *offset = _source + value[2];
731              //
732                  switch(GET_ELEMENT_TYPE(des))
733                  {
734                      case UNION_STYPE:
735                      {
736                          UINT32           choice;
737                      //
738                          choice = GetSelector(source, mst->values, des);
739                          retVal += MarshalUnion(index, offset, buffer, size, choice);
740                          break;
741                      }
742                      case ARRAY_STYPE:
743                      {
744                          UINT32               count;
745                      //
746                          count = GetSelector(source, mst->values, des);
747                          retVal += ArrayMarshal(index, offset, buffer, size, count);
748                          break;
749                      }
750                      case SIMPLE_STYPE:
751                      default:
752                      {
753                          // This is either another structure or a simple type
754                          retVal += Marshal(index, offset, buffer, size);
755                          break;
756                      }
757                  }
758              }
759              break;
760          }
761          case TPM2B_MTYPE:
762          {
763              // Get the number of bytes being marshaled
764              INT32          val = (int32_t)*((UINT16 *)source);
765          //
766              retVal = Marshal(UINT16_MARSHAL_REF, source, buffer, size);
767
768              // This is a standard 2B with a byte buffer
769              retVal += MarshalBytes(((TPM2B *)_source)->buffer, buffer, size, val);
770              break;
771          }
```

```
772         case TPM2BS_MTYPE: // A structure in a TPM2B
773         {
774             Tpm2bsMarshal_mst       *m2bst = (Tpm2bsMarshal_mst *)sel;
775             UINT8                   *offset;
776             UINT16                   amount;
777             UINT8                   *marshaledSize;
778         //
779             // Save the address of where the size should go
780             marshaledSize = *buffer;
781
782             // marshal the size (checks the space and advanced the pointer)
783             retVal = Marshal(UINT16_MARSHAL_REF, source, buffer, size);
784
785             // This gets the offsetof the structure to marshal. It was placed in the
786             // modifiers byte because the offset from the start of the TPM2B to the
787             // start of the structure is going to be less than 8 and the modifiers
788             // byte isn't needed for anything else.
789             offset = _source + (m2bst->modifiers & SIGNED_MASK);
790
791             // Marshal the structure and get its size
792             amount = Marshal(m2bst->dataIndex, offset, buffer, size);
793
794             // put the size in the space used when the size was marshaled.
795             if(buffer != NULL)
796                 UINT16_TO_BYTE_ARRAY(amount, marshaledSize);
797             retVal += amount;
798             break;
799         }
800         case LIST_MTYPE:
801         {
802             ListMarshal_mst *    mlt = ((ListMarshal_mst *)sel);
803             UINT8               *offset = _source + (mlt->modifiers & SIGNED_MASK);
804             retVal = Marshal(UINT32_MARSHAL_REF, source, buffer, size);
805             retVal += ArrayMarshal((marshalIndex_t)(mlt->arrayRef), offset,
806                                    buffer, size, *((UINT32 *)source));
807             break;
808         }
809         case NULL_MTYPE:
810             retVal = 0;
811             break;
812        case ERROR_MTYPE:
813        default:
814         {
815             if(size != NULL)
816                *size = -1;
817             retVal = 0;
818             break;
819         }
820     }
821     return retVal;
822
823 }
824 #endif // TABLE_DRIVEN_MARSHAL
```

9.10.8.2 TableMarshalData.c

This file contains the data initializer used for the table-driven marshaling code.

```
1 #include "Tpm.h"
2 #if TABLE_DRIVEN_MARSHAL
3 #include "TableMarshal.h"
4 #include "Marshal.h"
```

The array marshaling table

```
5   ArrayMarshal_mst   ArrayLookupTable[] = {
6       ARRAY_MARSHAL_ENTRY(UINT8),
7       ARRAY_MARSHAL_ENTRY(TPM_CC),
8       ARRAY_MARSHAL_ENTRY(TPMA_CC),
9       ARRAY_MARSHAL_ENTRY(TPM_ALG_ID),
10      ARRAY_MARSHAL_ENTRY(TPM_HANDLE),
11      ARRAY_MARSHAL_ENTRY(TPM2B_DIGEST),
12      ARRAY_MARSHAL_ENTRY(TPMT_HA),
13      ARRAY_MARSHAL_ENTRY(TPMS_PCR_SELECTION),
14      ARRAY_MARSHAL_ENTRY(TPMS_ALG_PROPERTY),
15      ARRAY_MARSHAL_ENTRY(TPMS_TAGGED_PROPERTY),
16      ARRAY_MARSHAL_ENTRY(TPMS_TAGGED_PCR_SELECT),
17      ARRAY_MARSHAL_ENTRY(TPM_ECC_CURVE),
18      ARRAY_MARSHAL_ENTRY(TPMS_TAGGED_POLICY),
19      ARRAY_MARSHAL_ENTRY(TPMS_ACT_DATA),
20      ARRAY_MARSHAL_ENTRY(TPMS_AC_OUTPUT)};
```

The main marshaling structure

```
21  MarshalData_st MarshalData = {
22  // UINT8_DATA
23  {UINT_MTYPE, 0},
24  // UINT16_DATA
25  {UINT_MTYPE, 1},
26  // UINT32_DATA
27  {UINT_MTYPE, 2},
28  // UINT64_DATA
29  {UINT_MTYPE, 3},
30  // INT8_DATA
31  {UINT_MTYPE, 0 + IS_SIGNED},
32  // INT16_DATA
33  {UINT_MTYPE, 1 + IS_SIGNED},
34  // INT32_DATA
35  {UINT_MTYPE, 2 + IS_SIGNED},
36  // INT64_DATA
37  {UINT_MTYPE, 3 + IS_SIGNED},
38  // UINT0_DATA
39  {NULL_MTYPE, 0},
40  // TPM_ECC_CURVE_DATA
41  {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_CURVE,
42      {TPM_ECC_NONE,
43      RANGE(1, 32, UINT16),
44      ((ECC_NIST_P192 << 0) | (ECC_NIST_P224 << 1) | (ECC_NIST_P256 << 2) |
45       (ECC_NIST_P384 << 3) | (ECC_NIST_P521 << 4) | (ECC_BN_P256 << 15)  |
46       (ECC_BN_P638 << 16)  | (ECC_SM2_P256 << 31))}},
47  // TPM_CLOCK_ADJUST_DATA
48  {MIN_MAX_MTYPE, ONE_BYTES|IS_SIGNED, (UINT8)TPM_RC_VALUE,
49      {RANGE(TPM_CLOCK_COARSE_SLOWER, TPM_CLOCK_COARSE_FASTER, INT8)}},
50  // TPM_EO_DATA
51  {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE,
52      {RANGE(TPM_EO_EQ, TPM_EO_BITCLEAR, UINT16)}},
53  // TPM_SU_DATA
54  {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 2,
55      {TPM_SU_CLEAR, TPM_SU_STATE}},
56  // TPM_SE_DATA
57  {TABLE_MTYPE, ONE_BYTES, (UINT8)TPM_RC_VALUE, 3,
58      {TPM_SE_HMAC, TPM_SE_POLICY, TPM_SE_TRIAL}},
59  // TPM_CAP_DATA
60  {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1, 1,
61     {RANGE(TPM_CAP_ALGS, TPM_CAP_ACT, UINT32),
62      TPM_CAP_VENDOR_PROPERTY}},
63  // TPMA_ALGORITHM_DATA
64  {ATTRIBUTES_MTYPE, FOUR_BYTES, 0xFFFFF8F0},
65  // TPMA_OBJECT_DATA
66  {ATTRIBUTES_MTYPE, FOUR_BYTES, 0xFFF0F309},
```

 Copyright © TCG 2006-2020

```
67  // TPMA_SESSION_DATA
68  {ATTRIBUTES_MTYPE, ONE_BYTES, 0x00000018},
69  // TPMA_ACT_DATA
70  {ATTRIBUTES_MTYPE, FOUR_BYTES, 0xFFFFFFFC},
71  // TPMI_YES_NO_DATA
72  {TABLE_MTYPE, ONE_BYTES, (UINT8)TPM_RC_VALUE, 2,
73      {NO, YES}},
74  // TPMI_DH_OBJECT_DATA
75  {VALUES_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 2, 0,
76      {TPM_RH_NULL,
77      RANGE(TRANSIENT_FIRST, TRANSIENT_LAST, UINT32),
78      RANGE(PERSISTENT_FIRST, PERSISTENT_LAST, UINT32)}},
79  // TPMI_DH_PARENT_DATA
80  {VALUES_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 2, 3,
81      {TPM_RH_NULL,
82      RANGE(TRANSIENT_FIRST, TRANSIENT_LAST, UINT32),
83      RANGE(PERSISTENT_FIRST, PERSISTENT_LAST, UINT32),
84      TPM_RH_OWNER, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}},
85  // TPMI_DH_PERSISTENT_DATA
86  {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
87      {RANGE(PERSISTENT_FIRST, PERSISTENT_LAST, UINT32)}},
88  // TPMI_DH_ENTITY_DATA
89  {VALUES_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 5, 4,
90      {TPM_RH_NULL,
91      RANGE(TRANSIENT_FIRST, TRANSIENT_LAST, UINT32),
92      RANGE(PERSISTENT_FIRST, PERSISTENT_LAST, UINT32),
93      RANGE(NV_INDEX_FIRST, NV_INDEX_LAST, UINT32),
94      RANGE(PCR_FIRST, PCR_LAST, UINT32),
95      RANGE(TPM_RH_AUTH_00, TPM_RH_AUTH_FF, UINT32),
96      TPM_RH_OWNER, TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}},
97  // TPMI_DH_PCR_DATA
98  {MIN_MAX_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE,
99      {TPM_RH_NULL,
100     RANGE(PCR_FIRST, PCR_LAST, UINT32)}},
101 // TPMI_SH_AUTH_SESSION_DATA
102 {VALUES_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 2, 0,
103     {TPM_RS_PW,
104     RANGE(HMAC_SESSION_FIRST, HMAC_SESSION_LAST, UINT32),
105     RANGE(POLICY_SESSION_FIRST, POLICY_SESSION_LAST, UINT32)}},
106 // TPMI_SH_HMAC_DATA
107 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
108     {RANGE(HMAC_SESSION_FIRST, HMAC_SESSION_LAST, UINT32)}},
109 // TPMI_SH_POLICY_DATA
110 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
111     {RANGE(POLICY_SESSION_FIRST, POLICY_SESSION_LAST, UINT32)}},
112 // TPMI_DH_CONTEXT_DATA
113 {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 3, 0,
114     {RANGE(HMAC_SESSION_FIRST, HMAC_SESSION_LAST, UINT32),
115     RANGE(POLICY_SESSION_FIRST, POLICY_SESSION_LAST, UINT32),
116     RANGE(TRANSIENT_FIRST, TRANSIENT_LAST, UINT32)}},
117 // TPMI_DH_SAVED_DATA
118 {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 2, 3,
119     {RANGE(HMAC_SESSION_FIRST, HMAC_SESSION_LAST, UINT32),
120     RANGE(POLICY_SESSION_FIRST, POLICY_SESSION_LAST, UINT32),
121     0x80000000, 0x80000001, 0x80000002}},
122 // TPMI_RH_HIERARCHY_DATA
123 {TABLE_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 3,
124     {TPM_RH_NULL,
125     TPM_RH_OWNER, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}},
126 // TPMI_RH_ENABLES_DATA
127 {TABLE_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 4,
128     {TPM_RH_NULL,
129     TPM_RH_OWNER, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM, TPM_RH_PLATFORM_NV}},
130 // TPMI_RH_HIERARCHY_AUTH_DATA
131 {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 4,
132     {TPM_RH_OWNER, TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}},
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
133     // TPMI_RH_HIERARCHY_POLICY_DATA
134     {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1, 4,
135         {RANGE(TPM_RH_ACT_0, TPM_RH_ACT_F, UINT32),
136         TPM_RH_OWNER, TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}},
137     // TPMI_RH_PLATFORM_DATA
138     {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1,
139         {TPM_RH_PLATFORM}},
140     // TPMI_RH_OWNER_DATA
141     {TABLE_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 1,
142         {TPM_RH_NULL,
143         TPM_RH_OWNER}},
144     // TPMI_RH_ENDORSEMENT_DATA
145     {TABLE_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 1,
146         {TPM_RH_NULL,
147         TPM_RH_ENDORSEMENT}},
148     // TPMI_RH_PROVISION_DATA
149     {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 2,
150         {TPM_RH_OWNER, TPM_RH_PLATFORM}},
151     // TPMI_RH_CLEAR_DATA
152     {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 2,
153         {TPM_RH_LOCKOUT, TPM_RH_PLATFORM}},
154     // TPMI_RH_NV_AUTH_DATA
155     {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1, 2,
156         {RANGE(NV_INDEX_FIRST, NV_INDEX_LAST, UINT32),
157         TPM_RH_OWNER, TPM_RH_PLATFORM}},
158     // TPMI_RH_LOCKOUT_DATA
159     {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1,
160         {TPM_RH_LOCKOUT}},
161     // TPMI_RH_NV_INDEX_DATA
162     {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
163         {RANGE(NV_INDEX_FIRST, NV_INDEX_LAST, UINT32)}},
164     // TPMI_RH_AC_DATA
165     {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
166         {RANGE(AC_FIRST, AC_LAST, UINT32)}},
167     // TPMI_RH_ACT_DATA
168     {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
169         {RANGE(TPM_RH_ACT_0, TPM_RH_ACT_F, UINT32)}},
170     // TPMI_ALG_HASH_DATA
171     {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_HASH,
172         {TPM_ALG_NULL,
173         RANGE(4, 41, UINT16),
174         ((ALG_SHA1 << 0)   | (ALG_SHA256 << 7) | (ALG_SHA384 << 8) |
175          (ALG_SHA512 << 9) | (ALG_SM3_256 << 14)),
176         ((ALG_SHA3_256 << 3)|(ALG_SHA3_384 << 4)|(ALG_SHA3_512 << 5))}},
177     // TPMI_ALG_ASYM_DATA
178     {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_ASYMMETRIC,
179         {TPM_ALG_NULL,
180         RANGE(1, 35, UINT16),
181         ((ALG_RSA << 0)),
182         ((ALG_ECC << 2))}},
183     // TPMI_ALG_SYM_DATA
184     {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SYMMETRIC,
185         {TPM_ALG_NULL,
186         RANGE(3, 38, UINT16),
187         ((ALG_TDES << 0)|(ALG_AES << 3)|(ALG_XOR << 7)|(ALG_SM4 << 16)),
188         ((ALG_CAMELLIA << 3))}},
189     // TPMI_ALG_SYM_OBJECT_DATA
190     {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SYMMETRIC,
191         {TPM_ALG_NULL,
192         RANGE(3, 38, UINT16),
193         ((ALG_TDES << 0)|(ALG_AES << 3)|(ALG_SM4 << 16)),
194         ((ALG_CAMELLIA << 3))}},
195     // TPMI_ALG_SYM_MODE_DATA
196     {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_MODE,
197         {TPM_ALG_NULL,
198         RANGE(63, 68, UINT16),
```

```
199     ((ALG_CMAC << 0)  | (ALG_CTR << 1)   | (ALG_OFB << 2)   | (ALG_CBC << 3)   |
200      (ALG_CFB << 4)   | (ALG_ECB << 5))}},
201 // TPMI_ALG_KDF_DATA
202 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_KDF,
203     {TPM_ALG_NULL,
204     RANGE(7, 34, UINT16),
205     ((ALG_MGF1 << 0)            | (ALG_KDF1_SP800_56A << 25) |
206      (ALG_KDF2 << 26)           | (ALG_KDF1_SP800_108 << 27))}},
207 // TPMI_ALG_SIG_SCHEME_DATA
208 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SCHEME,
209     {TPM_ALG_NULL,
210     RANGE(5, 28, UINT16),
211     ((ALG_HMAC << 0)     | (ALG_RSASSA << 15) | (ALG_RSAPSS << 17) |
212      (ALG_ECDSA << 19)  | (ALG_ECDAA << 21)  | (ALG_SM2 << 22)     |
213      (ALG_ECSCHNORR << 23))}},
214 // TPMI_ECC_KEY_EXCHANGE_DATA
215 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SCHEME,
216     {TPM_ALG_NULL,
217     RANGE(25, 29, UINT16),
218     ((ALG_ECDH << 0)|(ALG_SM2 << 2)|(ALG_ECMQV << 4))}},
219 // TPMI_ST_COMMAND_TAG_DATA
220 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_BAD_TAG, 2,
221     {TPM_ST_NO_SESSIONS, TPM_ST_SESSIONS}},
222 // TPMI_ALG_MAC_SCHEME_DATA
223 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SYMMETRIC,
224     {TPM_ALG_NULL,
225     RANGE(4, 63, UINT16),
226     ((ALG_SHA1 << 0)    | (ALG_SHA256 << 7) | (ALG_SHA384 << 8) |
227      (ALG_SHA512 << 9) | (ALG_SM3_256 << 14)),
228     ((ALG_SHA3_256 << 3)|(ALG_SHA3_384 << 4)|(ALG_SHA3_512 << 5)|(ALG_CMAC << 27))}},
229 // TPMI_ALG_CIPHER_MODE_DATA
230 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_MODE,
231     {TPM_ALG_NULL,
232     RANGE(64, 68, UINT16),
233     ((ALG_CTR << 0)|(ALG_OFB << 1)|(ALG_CBC << 2)|(ALG_CFB << 3)|(ALG_ECB << 4))}},
234 // TPMS_EMPTY_DATA
235 {STRUCTURE_MTYPE, 1,
236      {SET_ELEMENT_TYPE(SIMPLE_STYPE), UINT0_MARSHAL_REF, 0}},
237 // TPMS_ALGORITHM_DESCRIPTION_DATA
238 {STRUCTURE_MTYPE, 2, {
239         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
240             TPM_ALG_ID_MARSHAL_REF,
241             (UINT16)(offsetof(TPMS_ALGORITHM_DESCRIPTION, alg)),
242         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
243             TPMA_ALGORITHM_MARSHAL_REF,
244             (UINT16)(offsetof(TPMS_ALGORITHM_DESCRIPTION, attributes))}},
245 // TPMU_HA_DATA
246 {9, IS_ARRAY_UNION, (UINT16)(offsetof(TPMU_HA_mst, marshalingTypes)),
247     {(UINT32)TPM_ALG_SHA1,      (UINT32)TPM_ALG_SHA256,    (UINT32)TPM_ALG_SHA384,
248      (UINT32)TPM_ALG_SHA512,    (UINT32)TPM_ALG_SM3_256,   (UINT32)TPM_ALG_SHA3_256,
249      (UINT32)TPM_ALG_SHA3_384, (UINT32)TPM_ALG_SHA3_512, (UINT32)TPM_ALG_NULL},
250     {(UINT16)(SHA1_DIGEST_SIZE),      (UINT16)(SHA256_DIGEST_SIZE),
251      (UINT16)(SHA384_DIGEST_SIZE),    (UINT16)(SHA512_DIGEST_SIZE),
252      (UINT16)(SM3_256_DIGEST_SIZE),   (UINT16)(SHA3_256_DIGEST_SIZE),
253      (UINT16)(SHA3_384_DIGEST_SIZE), (UINT16)(SHA3_512_DIGEST_SIZE),
254      (UINT16)(0)}
255 },
256 // TPMT_HA_DATA
257 {STRUCTURE_MTYPE, 2, {
258         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
259             TPMI_ALG_HASH_MARSHAL_REF,
260             (UINT16)(offsetof(TPMT_HA, hashAlg)),
261         SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
262             TPMU_HA_MARSHAL_REF,
263             (UINT16)(offsetof(TPMT_HA, digest))}},
264 // TPM2B_DIGEST_DATA
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
265     {TPM2B_MTYPE, Type00_MARSHAL_REF},
266     // TPM2B_DATA_DATA
267     {TPM2B_MTYPE, Type01_MARSHAL_REF},
268     // TPM2B_EVENT_DATA
269     {TPM2B_MTYPE, Type02_MARSHAL_REF},
270     // TPM2B_MAX_BUFFER_DATA
271     {TPM2B_MTYPE, Type03_MARSHAL_REF},
272     // TPM2B_MAX_NV_BUFFER_DATA
273     {TPM2B_MTYPE, Type04_MARSHAL_REF},
274     // TPM2B_TIMEOUT_DATA
275     {TPM2B_MTYPE, Type05_MARSHAL_REF},
276     // TPM2B_IV_DATA
277     {TPM2B_MTYPE, Type06_MARSHAL_REF},
278     // NULL_UNION_DATA
279     {0},
280     // TPM2B_NAME_DATA
281     {TPM2B_MTYPE, Type07_MARSHAL_REF},
282     // TPMS_PCR_SELECT_DATA
283     {STRUCTURE_MTYPE, 2, {
284             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(ONE_BYTES),
285                 Type08_MARSHAL_REF,
286                 (UINT16)(offsetof(TPMS_PCR_SELECT, sizeofSelect)),
287             SET_ELEMENT_TYPE(ARRAY_STYPE)|SET_ELEMENT_NUMBER(0),
288                 UINT8_ARRAY_MARSHAL_INDEX,
289                 (UINT16)(offsetof(TPMS_PCR_SELECT, pcrSelect))}},
290     // TPMS_PCR_SELECTION_DATA
291     {STRUCTURE_MTYPE, 3, {
292             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
293                 TPMI_ALG_HASH_MARSHAL_REF,
294                 (UINT16)(offsetof(TPMS_PCR_SELECTION, hash)),
295             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(ONE_BYTES),
296                 Type08_MARSHAL_REF,
297                 (UINT16)(offsetof(TPMS_PCR_SELECTION, sizeofSelect)),
298             SET_ELEMENT_TYPE(ARRAY_STYPE)|SET_ELEMENT_NUMBER(1),
299                 UINT8_ARRAY_MARSHAL_INDEX,
300                 (UINT16)(offsetof(TPMS_PCR_SELECTION, pcrSelect))}},
301     // TPMT_TK_CREATION_DATA
302     {STRUCTURE_MTYPE, 3, {
303             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
304                 Type10_MARSHAL_REF,
305                 (UINT16)(offsetof(TPMT_TK_CREATION, tag)),
306             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
307                 TPMI_RH_HIERARCHY_MARSHAL_REF|NULL_FLAG,
308                 (UINT16)(offsetof(TPMT_TK_CREATION, hierarchy)),
309             SET_ELEMENT_TYPE(SIMPLE_STYPE),
310                 TPM2B_DIGEST_MARSHAL_REF,
311                 (UINT16)(offsetof(TPMT_TK_CREATION, digest))}},
312     // TPMT_TK_VERIFIED_DATA
313     {STRUCTURE_MTYPE, 3, {
314             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
315                 Type11_MARSHAL_REF,
316                 (UINT16)(offsetof(TPMT_TK_VERIFIED, tag)),
317             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
318                 TPMI_RH_HIERARCHY_MARSHAL_REF|NULL_FLAG,
319                 (UINT16)(offsetof(TPMT_TK_VERIFIED, hierarchy)),
320             SET_ELEMENT_TYPE(SIMPLE_STYPE),
321                 TPM2B_DIGEST_MARSHAL_REF,
322                 (UINT16)(offsetof(TPMT_TK_VERIFIED, digest))}},
323     // TPMT_TK_AUTH_DATA
324     {STRUCTURE_MTYPE, 3, {
325             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
326                 Type12_MARSHAL_REF,
327                 (UINT16)(offsetof(TPMT_TK_AUTH, tag)),
328             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
329                 TPMI_RH_HIERARCHY_MARSHAL_REF|NULL_FLAG,
330                 (UINT16)(offsetof(TPMT_TK_AUTH, hierarchy)),
```

 Copyright © TCG 2006-2020

```
331         SET_ELEMENT_TYPE(SIMPLE_STYPE),
332             TPM2B_DIGEST_MARSHAL_REF,
333             (UINT16)(offsetof(TPMT_TK_AUTH, digest))}},
334 // TPMT_TK_HASHCHECK_DATA
335 {STRUCTURE_MTYPE, 3, {
336         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
337             Type13_MARSHAL_REF,
338             (UINT16)(offsetof(TPMT_TK_HASHCHECK, tag)),
339         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
340             TPMI_RH_HIERARCHY_MARSHAL_REF|NULL_FLAG,
341             (UINT16)(offsetof(TPMT_TK_HASHCHECK, hierarchy)),
342         SET_ELEMENT_TYPE(SIMPLE_STYPE),
343             TPM2B_DIGEST_MARSHAL_REF,
344             (UINT16)(offsetof(TPMT_TK_HASHCHECK, digest))}},
345 // TPMS_ALG_PROPERTY_DATA
346 {STRUCTURE_MTYPE, 2, {
347         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
348             TPM_ALG_ID_MARSHAL_REF,
349             (UINT16)(offsetof(TPMS_ALG_PROPERTY, alg)),
350         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
351             TPMA_ALGORITHM_MARSHAL_REF,
352             (UINT16)(offsetof(TPMS_ALG_PROPERTY, algProperties))}},
353 // TPMS_TAGGED_PROPERTY_DATA
354 {STRUCTURE_MTYPE, 2, {
355         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
356             TPM_PT_MARSHAL_REF,
357             (UINT16)(offsetof(TPMS_TAGGED_PROPERTY, property)),
358         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
359             UINT32_MARSHAL_REF,
360             (UINT16)(offsetof(TPMS_TAGGED_PROPERTY, value))}},
361 // TPMS_TAGGED_PCR_SELECT_DATA
362 {STRUCTURE_MTYPE, 3, {
363         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
364             TPM_PT_PCR_MARSHAL_REF,
365             (UINT16)(offsetof(TPMS_TAGGED_PCR_SELECT, tag)),
366         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(ONE_BYTES),
367             Type08_MARSHAL_REF,
368             (UINT16)(offsetof(TPMS_TAGGED_PCR_SELECT, sizeofSelect)),
369         SET_ELEMENT_TYPE(ARRAY_STYPE)|SET_ELEMENT_NUMBER(1),
370             UINT8_ARRAY_MARSHAL_INDEX,
371             (UINT16)(offsetof(TPMS_TAGGED_PCR_SELECT, pcrSelect))}},
372 // TPMS_TAGGED_POLICY_DATA
373 {STRUCTURE_MTYPE, 2, {
374         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
375             TPM_HANDLE_MARSHAL_REF,
376             (UINT16)(offsetof(TPMS_TAGGED_POLICY, handle)),
377         SET_ELEMENT_TYPE(SIMPLE_STYPE),
378             TPMT_HA_MARSHAL_REF,
379             (UINT16)(offsetof(TPMS_TAGGED_POLICY, policyHash))}},
380 // TPMS_ACT_DATA_DATA
381 {STRUCTURE_MTYPE, 3, {
382         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
383             TPM_HANDLE_MARSHAL_REF,
384             (UINT16)(offsetof(TPMS_ACT_DATA, handle)),
385         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
386             UINT32_MARSHAL_REF,
387             (UINT16)(offsetof(TPMS_ACT_DATA, timeout)),
388         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
389             TPMA_ACT_MARSHAL_REF,
390             (UINT16)(offsetof(TPMS_ACT_DATA, attributes))}},
391 // TPML_CC_DATA
392 {LIST_MTYPE,
393     (UINT8)(offsetof(TPML_CC, commandCodes)),
394     Type15_MARSHAL_REF,
395     TPM_CC_ARRAY_MARSHAL_INDEX},
396 // TPML_CCA_DATA
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
397     {LIST_MTYPE,
398         (UINT8)(offsetof(TPML_CCA, commandAttributes)),
399         Type15_MARSHAL_REF,
400         TPMA_CC_ARRAY_MARSHAL_INDEX},
401     // TPML_ALG_DATA
402     {LIST_MTYPE,
403         (UINT8)(offsetof(TPML_ALG, algorithms)),
404         Type17_MARSHAL_REF,
405         TPM_ALG_ID_ARRAY_MARSHAL_INDEX},
406     // TPML_HANDLE_DATA
407     {LIST_MTYPE,
408         (UINT8)(offsetof(TPML_HANDLE, handle)),
409         Type18_MARSHAL_REF,
410         TPM_HANDLE_ARRAY_MARSHAL_INDEX},
411     // TPML_DIGEST_DATA
412     {LIST_MTYPE,
413         (UINT8)(offsetof(TPML_DIGEST, digests)),
414         Type19_MARSHAL_REF,
415         TPM2B_DIGEST_ARRAY_MARSHAL_INDEX},
416     // TPML_DIGEST_VALUES_DATA
417     {LIST_MTYPE,
418         (UINT8)(offsetof(TPML_DIGEST_VALUES, digests)),
419         Type20_MARSHAL_REF,
420         TPMT_HA_ARRAY_MARSHAL_INDEX},
421     // TPML_PCR_SELECTION_DATA
422     {LIST_MTYPE,
423         (UINT8)(offsetof(TPML_PCR_SELECTION, pcrSelections)),
424         Type20_MARSHAL_REF,
425         TPMS_PCR_SELECTION_ARRAY_MARSHAL_INDEX},
426     // TPML_ALG_PROPERTY_DATA
427     {LIST_MTYPE,
428         (UINT8)(offsetof(TPML_ALG_PROPERTY, algProperties)),
429         Type22_MARSHAL_REF,
430         TPMS_ALG_PROPERTY_ARRAY_MARSHAL_INDEX},
431     // TPML_TAGGED_TPM_PROPERTY_DATA
432     {LIST_MTYPE,
433         (UINT8)(offsetof(TPML_TAGGED_TPM_PROPERTY, tpmProperty)),
434         Type23_MARSHAL_REF,
435         TPMS_TAGGED_PROPERTY_ARRAY_MARSHAL_INDEX},
436     // TPML_TAGGED_PCR_PROPERTY_DATA
437     {LIST_MTYPE,
438         (UINT8)(offsetof(TPML_TAGGED_PCR_PROPERTY, pcrProperty)),
439         Type24_MARSHAL_REF,
440         TPMS_TAGGED_PCR_SELECT_ARRAY_MARSHAL_INDEX},
441     // TPML_ECC_CURVE_DATA
442     {LIST_MTYPE,
443         (UINT8)(offsetof(TPML_ECC_CURVE, eccCurves)),
444         Type25_MARSHAL_REF,
445         TPM_ECC_CURVE_ARRAY_MARSHAL_INDEX},
446     // TPML_TAGGED_POLICY_DATA
447     {LIST_MTYPE,
448         (UINT8)(offsetof(TPML_TAGGED_POLICY, policies)),
449         Type26_MARSHAL_REF,
450         TPMS_TAGGED_POLICY_ARRAY_MARSHAL_INDEX},
451     // TPML_ACT_DATA_DATA
452     {LIST_MTYPE,
453         (UINT8)(offsetof(TPML_ACT_DATA, actData)),
454         Type27_MARSHAL_REF,
455         TPMS_ACT_DATA_ARRAY_MARSHAL_INDEX},
456     // TPMU_CAPABILITIES_DATA
457     {11, 0, (UINT16)(offsetof(TPMU_CAPABILITIES_mst, marshalingTypes)),
458         {(UINT32)TPM_CAP_ALGS,            (UINT32)TPM_CAP_HANDLES,
459          (UINT32)TPM_CAP_COMMANDS,        (UINT32)TPM_CAP_PP_COMMANDS,
460          (UINT32)TPM_CAP_AUDIT_COMMANDS, (UINT32)TPM_CAP_PCRS,
461          (UINT32)TPM_CAP_TPM_PROPERTIES, (UINT32)TPM_CAP_PCR_PROPERTIES,
462          (UINT32)TPM_CAP_ECC_CURVES,      (UINT32)TPM_CAP_AUTH_POLICIES,
```

 Copyright © TCG 2006-2020

```
463         (UINT32)TPM_CAP_ACT},
464        {(UINT16)(TPML_ALG_PROPERTY_MARSHAL_REF),
465         (UINT16)(TPML_HANDLE_MARSHAL_REF),
466         (UINT16)(TPML_CCA_MARSHAL_REF),
467         (UINT16)(TPML_CC_MARSHAL_REF),
468         (UINT16)(TPML_CC_MARSHAL_REF),
469         (UINT16)(TPML_PCR_SELECTION_MARSHAL_REF),
470         (UINT16)(TPML_TAGGED_TPM_PROPERTY_MARSHAL_REF),
471         (UINT16)(TPML_TAGGED_PCR_PROPERTY_MARSHAL_REF),
472         (UINT16)(TPML_ECC_CURVE_MARSHAL_REF),
473         (UINT16)(TPML_TAGGED_POLICY_MARSHAL_REF),
474         (UINT16)(TPML_ACT_DATA_MARSHAL_REF)}
475 },
476 // TPMS_CAPABILITY_DATA_DATA
477 {STRUCTURE_MTYPE, 2, {
478         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
479             TPM_CAP_MARSHAL_REF,
480             (UINT16)(offsetof(TPMS_CAPABILITY_DATA, capability)),
481         SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
482             TPMU_CAPABILITIES_MARSHAL_REF,
483             (UINT16)(offsetof(TPMS_CAPABILITY_DATA, data))}},
484 // TPMS_CLOCK_INFO_DATA
485 {STRUCTURE_MTYPE, 4, {
486         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(EIGHT_BYTES),
487             UINT64_MARSHAL_REF,
488             (UINT16)(offsetof(TPMS_CLOCK_INFO, clock)),
489         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
490             UINT32_MARSHAL_REF,
491             (UINT16)(offsetof(TPMS_CLOCK_INFO, resetCount)),
492         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
493             UINT32_MARSHAL_REF,
494             (UINT16)(offsetof(TPMS_CLOCK_INFO, restartCount)),
495         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(ONE_BYTES),
496             TPMI_YES_NO_MARSHAL_REF,
497             (UINT16)(offsetof(TPMS_CLOCK_INFO, safe))}},
498 // TPMS_TIME_INFO_DATA
499 {STRUCTURE_MTYPE, 2, {
500         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(EIGHT_BYTES),
501             UINT64_MARSHAL_REF,
502             (UINT16)(offsetof(TPMS_TIME_INFO, time)),
503         SET_ELEMENT_TYPE(SIMPLE_STYPE),
504             TPMS_CLOCK_INFO_MARSHAL_REF,
505             (UINT16)(offsetof(TPMS_TIME_INFO, clockInfo))}},
506 // TPMS_TIME_ATTEST_INFO_DATA
507 {STRUCTURE_MTYPE, 2, {
508         SET_ELEMENT_TYPE(SIMPLE_STYPE),
509             TPMS_TIME_INFO_MARSHAL_REF,
510             (UINT16)(offsetof(TPMS_TIME_ATTEST_INFO, time)),
511         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(EIGHT_BYTES),
512             UINT64_MARSHAL_REF,
513             (UINT16)(offsetof(TPMS_TIME_ATTEST_INFO, firmwareVersion))}},
514 // TPMS_CERTIFY_INFO_DATA
515 {STRUCTURE_MTYPE, 2, {
516         SET_ELEMENT_TYPE(SIMPLE_STYPE),
517             TPM2B_NAME_MARSHAL_REF,
518             (UINT16)(offsetof(TPMS_CERTIFY_INFO, name)),
519         SET_ELEMENT_TYPE(SIMPLE_STYPE),
520             TPM2B_NAME_MARSHAL_REF,
521             (UINT16)(offsetof(TPMS_CERTIFY_INFO, qualifiedName))}},
522 // TPMS_QUOTE_INFO_DATA
523 {STRUCTURE_MTYPE, 2, {
524         SET_ELEMENT_TYPE(SIMPLE_STYPE),
525             TPML_PCR_SELECTION_MARSHAL_REF,
526             (UINT16)(offsetof(TPMS_QUOTE_INFO, pcrSelect)),
527         SET_ELEMENT_TYPE(SIMPLE_STYPE),
528             TPM2B_DIGEST_MARSHAL_REF,
```

```
529                 (UINT16)(offsetof(TPMS_QUOTE_INFO, pcrDigest))}},
530     // TPMS_COMMAND_AUDIT_INFO_DATA
531     {STRUCTURE_MTYPE, 4, {
532             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(EIGHT_BYTES),
533                 UINT64_MARSHAL_REF,
534                 (UINT16)(offsetof(TPMS_COMMAND_AUDIT_INFO, auditCounter)),
535             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
536                 TPM_ALG_ID_MARSHAL_REF,
537                 (UINT16)(offsetof(TPMS_COMMAND_AUDIT_INFO, digestAlg)),
538             SET_ELEMENT_TYPE(SIMPLE_STYPE),
539                 TPM2B_DIGEST_MARSHAL_REF,
540                 (UINT16)(offsetof(TPMS_COMMAND_AUDIT_INFO, auditDigest)),
541             SET_ELEMENT_TYPE(SIMPLE_STYPE),
542                 TPM2B_DIGEST_MARSHAL_REF,
543                 (UINT16)(offsetof(TPMS_COMMAND_AUDIT_INFO, commandDigest))}},
544     // TPMS_SESSION_AUDIT_INFO_DATA
545     {STRUCTURE_MTYPE, 2, {
546             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(ONE_BYTES),
547                 TPMI_YES_NO_MARSHAL_REF,
548                 (UINT16)(offsetof(TPMS_SESSION_AUDIT_INFO, exclusiveSession)),
549             SET_ELEMENT_TYPE(SIMPLE_STYPE),
550                 TPM2B_DIGEST_MARSHAL_REF,
551                 (UINT16)(offsetof(TPMS_SESSION_AUDIT_INFO, sessionDigest))}},
552     // TPMS_CREATION_INFO_DATA
553     {STRUCTURE_MTYPE, 2, {
554             SET_ELEMENT_TYPE(SIMPLE_STYPE),
555                 TPM2B_NAME_MARSHAL_REF,
556                 (UINT16)(offsetof(TPMS_CREATION_INFO, objectName)),
557             SET_ELEMENT_TYPE(SIMPLE_STYPE),
558                 TPM2B_DIGEST_MARSHAL_REF,
559                 (UINT16)(offsetof(TPMS_CREATION_INFO, creationHash))}},
560     // TPMS_NV_CERTIFY_INFO_DATA
561     {STRUCTURE_MTYPE, 3, {
562             SET_ELEMENT_TYPE(SIMPLE_STYPE),
563                 TPM2B_NAME_MARSHAL_REF,
564                 (UINT16)(offsetof(TPMS_NV_CERTIFY_INFO, indexName)),
565             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
566                 UINT16_MARSHAL_REF,
567                 (UINT16)(offsetof(TPMS_NV_CERTIFY_INFO, offset)),
568             SET_ELEMENT_TYPE(SIMPLE_STYPE),
569                 TPM2B_MAX_NV_BUFFER_MARSHAL_REF,
570                 (UINT16)(offsetof(TPMS_NV_CERTIFY_INFO, nvContents))}},
571     // TPMS_NV_DIGEST_CERTIFY_INFO_DATA
572     {STRUCTURE_MTYPE, 2, {
573             SET_ELEMENT_TYPE(SIMPLE_STYPE),
574                 TPM2B_NAME_MARSHAL_REF,
575                 (UINT16)(offsetof(TPMS_NV_DIGEST_CERTIFY_INFO, indexName)),
576             SET_ELEMENT_TYPE(SIMPLE_STYPE),
577                 TPM2B_DIGEST_MARSHAL_REF,
578                 (UINT16)(offsetof(TPMS_NV_DIGEST_CERTIFY_INFO, nvDigest))}},
579     // TPMI_ST_ATTEST_DATA
580     {VALUES_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 1, 1,
581        {RANGE(TPM_ST_ATTEST_NV, TPM_ST_ATTEST_CREATION, UINT16),
582         TPM_ST_ATTEST_NV_DIGEST}},
583     // TPMU_ATTEST_DATA
584     {8, 0, (UINT16)(offsetof(TPMU_ATTEST_mst, marshalingTypes)),
585         {(UINT32)TPM_ST_ATTEST_CERTIFY,        (UINT32)TPM_ST_ATTEST_CREATION,
586          (UINT32)TPM_ST_ATTEST_QUOTE,          (UINT32)TPM_ST_ATTEST_COMMAND_AUDIT,
587          (UINT32)TPM_ST_ATTEST_SESSION_AUDIT,  (UINT32)TPM_ST_ATTEST_TIME,
588          (UINT32)TPM_ST_ATTEST_NV,             (UINT32)TPM_ST_ATTEST_NV_DIGEST},
589         {(UINT16)(TPMS_CERTIFY_INFO_MARSHAL_REF),
590          (UINT16)(TPMS_CREATION_INFO_MARSHAL_REF),
591          (UINT16)(TPMS_QUOTE_INFO_MARSHAL_REF),
592          (UINT16)(TPMS_COMMAND_AUDIT_INFO_MARSHAL_REF),
593          (UINT16)(TPMS_SESSION_AUDIT_INFO_MARSHAL_REF),
594          (UINT16)(TPMS_TIME_ATTEST_INFO_MARSHAL_REF),
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
595         (UINT16)(TPMS_NV_CERTIFY_INFO_MARSHAL_REF),
596         (UINT16)(TPMS_NV_DIGEST_CERTIFY_INFO_MARSHAL_REF)}
597     },
598     // TPMS_ATTEST_DATA
599     {STRUCTURE_MTYPE, 7, {
600             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
601                 TPM_GENERATED_MARSHAL_REF,
602                 (UINT16)(offsetof(TPMS_ATTEST, magic)),
603             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
604                 TPMI_ST_ATTEST_MARSHAL_REF,
605                 (UINT16)(offsetof(TPMS_ATTEST, type)),
606             SET_ELEMENT_TYPE(SIMPLE_STYPE),
607                 TPM2B_NAME_MARSHAL_REF,
608                 (UINT16)(offsetof(TPMS_ATTEST, qualifiedSigner)),
609             SET_ELEMENT_TYPE(SIMPLE_STYPE),
610                 TPM2B_DATA_MARSHAL_REF,
611                 (UINT16)(offsetof(TPMS_ATTEST, extraData)),
612             SET_ELEMENT_TYPE(SIMPLE_STYPE),
613                 TPMS_CLOCK_INFO_MARSHAL_REF,
614                 (UINT16)(offsetof(TPMS_ATTEST, clockInfo)),
615             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(EIGHT_BYTES),
616                 UINT64_MARSHAL_REF,
617                 (UINT16)(offsetof(TPMS_ATTEST, firmwareVersion)),
618             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(1),
619                 TPMU_ATTEST_MARSHAL_REF,
620                 (UINT16)(offsetof(TPMS_ATTEST, attested))}},
621     // TPM2B_ATTEST_DATA
622     {TPM2B_MTYPE, Type28_MARSHAL_REF},
623     // TPMS_AUTH_COMMAND_DATA
624     {STRUCTURE_MTYPE, 4, {
625             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
626                 TPMI_SH_AUTH_SESSION_MARSHAL_REF|NULL_FLAG,
627                 (UINT16)(offsetof(TPMS_AUTH_COMMAND, sessionHandle)),
628             SET_ELEMENT_TYPE(SIMPLE_STYPE),
629                 TPM2B_NONCE_MARSHAL_REF,
630                 (UINT16)(offsetof(TPMS_AUTH_COMMAND, nonce)),
631             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(ONE_BYTES),
632                 TPMA_SESSION_MARSHAL_REF,
633                 (UINT16)(offsetof(TPMS_AUTH_COMMAND, sessionAttributes)),
634             SET_ELEMENT_TYPE(SIMPLE_STYPE),
635                 TPM2B_AUTH_MARSHAL_REF,
636                 (UINT16)(offsetof(TPMS_AUTH_COMMAND, hmac))}},
637     // TPMS_AUTH_RESPONSE_DATA
638     {STRUCTURE_MTYPE, 3, {
639             SET_ELEMENT_TYPE(SIMPLE_STYPE),
640                 TPM2B_NONCE_MARSHAL_REF,
641                 (UINT16)(offsetof(TPMS_AUTH_RESPONSE, nonce)),
642             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(ONE_BYTES),
643                 TPMA_SESSION_MARSHAL_REF,
644                 (UINT16)(offsetof(TPMS_AUTH_RESPONSE, sessionAttributes)),
645             SET_ELEMENT_TYPE(SIMPLE_STYPE),
646                 TPM2B_AUTH_MARSHAL_REF,
647                 (UINT16)(offsetof(TPMS_AUTH_RESPONSE, hmac))}},
648     // TPMI_TDES_KEY_BITS_DATA
649     {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 1,
650         {128*TDES_128}},
651     // TPMI_AES_KEY_BITS_DATA
652     {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 3,
653         {192*AES_192, 128*AES_128, 256*AES_256}},
654     // TPMI_SM4_KEY_BITS_DATA
655     {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 1,
656         {128*SM4_128}},
657     // TPMI_CAMELLIA_KEY_BITS_DATA
658     {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 3,
659         {192*CAMELLIA_192, 128*CAMELLIA_128, 256*CAMELLIA_256}},
660     // TPMU_SYM_KEY_BITS_DATA
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
661    {6, 0, (UINT16)(offsetof(TPMU_SYM_KEY_BITS_mst, marshalingTypes)),
662        {(UINT32)TPM_ALG_TDES,     (UINT32)TPM_ALG_AES,      (UINT32)TPM_ALG_SM4,
663         (UINT32)TPM_ALG_CAMELLIA, (UINT32)TPM_ALG_XOR,      (UINT32)TPM_ALG_NULL},
664        {(UINT16)(TPMI_TDES_KEY_BITS_MARSHAL_REF),
665         (UINT16)(TPMI_AES_KEY_BITS_MARSHAL_REF),
666         (UINT16)(TPMI_SM4_KEY_BITS_MARSHAL_REF),
667         (UINT16)(TPMI_CAMELLIA_KEY_BITS_MARSHAL_REF),
668         (UINT16)(TPMI_ALG_HASH_MARSHAL_REF),
669         (UINT16)(UINT0_MARSHAL_REF)}
670    },
671    // TPMU_SYM_MODE_DATA
672    {6, 0, (UINT16)(offsetof(TPMU_SYM_MODE_mst, marshalingTypes)),
673        {(UINT32)TPM_ALG_TDES,     (UINT32)TPM_ALG_AES,      (UINT32)TPM_ALG_SM4,
674         (UINT32)TPM_ALG_CAMELLIA, (UINT32)TPM_ALG_XOR,      (UINT32)TPM_ALG_NULL},
675        {(UINT16)(TPMI_ALG_SYM_MODE_MARSHAL_REF|NULL_FLAG),
676         (UINT16)(TPMI_ALG_SYM_MODE_MARSHAL_REF|NULL_FLAG),
677         (UINT16)(TPMI_ALG_SYM_MODE_MARSHAL_REF|NULL_FLAG),
678         (UINT16)(TPMI_ALG_SYM_MODE_MARSHAL_REF|NULL_FLAG),
679         (UINT16)(UINT0_MARSHAL_REF),
680         (UINT16)(UINT0_MARSHAL_REF)}
681    },
682    // TPMT_SYM_DEF_DATA
683    {STRUCTURE_MTYPE, 3, {
684            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
685                TPMI_ALG_SYM_MARSHAL_REF,
686                (UINT16)(offsetof(TPMT_SYM_DEF, algorithm)),
687            SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
688                TPMU_SYM_KEY_BITS_MARSHAL_REF,
689                (UINT16)(offsetof(TPMT_SYM_DEF, keyBits)),
690            SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
691                TPMU_SYM_MODE_MARSHAL_REF,
692                (UINT16)(offsetof(TPMT_SYM_DEF, mode))}},
693    // TPMT_SYM_DEF_OBJECT_DATA
694    {STRUCTURE_MTYPE, 3, {
695            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
696                TPMI_ALG_SYM_OBJECT_MARSHAL_REF,
697                (UINT16)(offsetof(TPMT_SYM_DEF_OBJECT, algorithm)),
698            SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
699                TPMU_SYM_KEY_BITS_MARSHAL_REF,
700                (UINT16)(offsetof(TPMT_SYM_DEF_OBJECT, keyBits)),
701            SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
702                TPMU_SYM_MODE_MARSHAL_REF,
703                (UINT16)(offsetof(TPMT_SYM_DEF_OBJECT, mode))}},
704    // TPM2B_SYM_KEY_DATA
705    {TPM2B_MTYPE, Type29_MARSHAL_REF},
706    // TPMS_SYMCIPHER_PARMS_DATA
707    {STRUCTURE_MTYPE, 1, {
708            SET_ELEMENT_TYPE(SIMPLE_STYPE),
709                TPMT_SYM_DEF_OBJECT_MARSHAL_REF,
710                (UINT16)(offsetof(TPMS_SYMCIPHER_PARMS, sym))}},
711    // TPM2B_LABEL_DATA
712    {TPM2B_MTYPE, Type30_MARSHAL_REF},
713    // TPMS_DERIVE_DATA
714    {STRUCTURE_MTYPE, 2, {
715            SET_ELEMENT_TYPE(SIMPLE_STYPE),
716                TPM2B_LABEL_MARSHAL_REF,
717                (UINT16)(offsetof(TPMS_DERIVE, label)),
718            SET_ELEMENT_TYPE(SIMPLE_STYPE),
719                TPM2B_LABEL_MARSHAL_REF,
720                (UINT16)(offsetof(TPMS_DERIVE, context))}},
721    // TPM2B_DERIVE_DATA
722    {TPM2B_MTYPE, Type31_MARSHAL_REF},
723    // TPM2B_SENSITIVE_DATA_DATA
724    {TPM2B_MTYPE, Type32_MARSHAL_REF},
725    // TPMS_SENSITIVE_CREATE_DATA
726    {STRUCTURE_MTYPE, 2, {
```

 Copyright © TCG 2006-2020

```
727             SET_ELEMENT_TYPE(SIMPLE_STYPE),
728                 TPM2B_AUTH_MARSHAL_REF,
729                 (UINT16)(offsetof(TPMS_SENSITIVE_CREATE, userAuth)),
730             SET_ELEMENT_TYPE(SIMPLE_STYPE),
731                 TPM2B_SENSITIVE_DATA_MARSHAL_REF,
732                 (UINT16)(offsetof(TPMS_SENSITIVE_CREATE, data))}},
733     // TPM2B_SENSITIVE_CREATE_DATA
734     {TPM2BS_MTYPE,
735         (UINT8)(offsetof(TPM2B_SENSITIVE_CREATE, sensitive))|SIZE_EQUAL,
736         UINT16_MARSHAL_REF,
737         TPMS_SENSITIVE_CREATE_MARSHAL_REF},
738     // TPMS_SCHEME_HASH_DATA
739     {STRUCTURE_MTYPE, 1, {
740             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
741                 TPMI_ALG_HASH_MARSHAL_REF,
742                 (UINT16)(offsetof(TPMS_SCHEME_HASH, hashAlg))}},
743     // TPMS_SCHEME_ECDAA_DATA
744     {STRUCTURE_MTYPE, 2, {
745             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
746                 TPMI_ALG_HASH_MARSHAL_REF,
747                 (UINT16)(offsetof(TPMS_SCHEME_ECDAA, hashAlg)),
748             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
749                 UINT16_MARSHAL_REF,
750                 (UINT16)(offsetof(TPMS_SCHEME_ECDAA, count))}},
751     // TPMI_ALG_KEYEDHASH_SCHEME_DATA
752     {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_VALUE,
753         {TPM_ALG_NULL,
754         RANGE(5, 10, UINT16),
755         ((ALG_HMAC << 0)|(ALG_XOR << 5))}},
756     // TPMS_SCHEME_XOR_DATA
757     {STRUCTURE_MTYPE, 2, {
758             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
759                 TPMI_ALG_HASH_MARSHAL_REF,
760                 (UINT16)(offsetof(TPMS_SCHEME_XOR, hashAlg)),
761             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
762                 TPMI_ALG_KDF_MARSHAL_REF|NULL_FLAG,
763                 (UINT16)(offsetof(TPMS_SCHEME_XOR, kdf))}},
764     // TPMU_SCHEME_KEYEDHASH_DATA
765     {3, 0, (UINT16)(offsetof(TPMU_SCHEME_KEYEDHASH_mst, marshalingTypes)),
766         {(UINT32)TPM_ALG_HMAC, (UINT32)TPM_ALG_XOR, (UINT32)TPM_ALG_NULL},
767         {(UINT16)(TPMS_SCHEME_HMAC_MARSHAL_REF),
768          (UINT16)(TPMS_SCHEME_XOR_MARSHAL_REF),
769          (UINT16)(UINT0_MARSHAL_REF)}
770     },
771     // TPMT_KEYEDHASH_SCHEME_DATA
772     {STRUCTURE_MTYPE, 2, {
773             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
774                 TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_REF,
775                 (UINT16)(offsetof(TPMT_KEYEDHASH_SCHEME, scheme)),
776             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
777                 TPMU_SCHEME_KEYEDHASH_MARSHAL_REF,
778                 (UINT16)(offsetof(TPMT_KEYEDHASH_SCHEME, details))}},
779     // TPMU_SIG_SCHEME_DATA
780     {8, 0, (UINT16)(offsetof(TPMU_SIG_SCHEME_mst, marshalingTypes)),
781         {(UINT32)TPM_ALG_ECDAA,      (UINT32)TPM_ALG_RSASSA,
782          (UINT32)TPM_ALG_RSAPSS,     (UINT32)TPM_ALG_ECDSA,
783          (UINT32)TPM_ALG_SM2,        (UINT32)TPM_ALG_ECSCHNORR,
784          (UINT32)TPM_ALG_HMAC,       (UINT32)TPM_ALG_NULL},
785         {(UINT16)(TPMS_SIG_SCHEME_ECDAA_MARSHAL_REF),
786          (UINT16)(TPMS_SIG_SCHEME_RSASSA_MARSHAL_REF),
787          (UINT16)(TPMS_SIG_SCHEME_RSAPSS_MARSHAL_REF),
788          (UINT16)(TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF),
789          (UINT16)(TPMS_SIG_SCHEME_SM2_MARSHAL_REF),
790          (UINT16)(TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_REF),
791          (UINT16)(TPMS_SCHEME_HMAC_MARSHAL_REF),
792          (UINT16)(UINT0_MARSHAL_REF)}
```

```
793    },
794    // TPMT_SIG_SCHEME_DATA
795    {STRUCTURE_MTYPE, 2, {
796            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
797                TPMI_ALG_SIG_SCHEME_MARSHAL_REF,
798                (UINT16)(offsetof(TPMT_SIG_SCHEME, scheme)),
799            SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
800                TPMU_SIG_SCHEME_MARSHAL_REF,
801                (UINT16)(offsetof(TPMT_SIG_SCHEME, details))}},
802    // TPMU_KDF_SCHEME_DATA
803    {5, 0, (UINT16)(offsetof(TPMU_KDF_SCHEME_mst, marshalingTypes)),
804        {(UINT32)TPM_ALG_MGF1,           (UINT32)TPM_ALG_KDF1_SP800_56A,
805         (UINT32)TPM_ALG_KDF2,           (UINT32)TPM_ALG_KDF1_SP800_108,
806         (UINT32)TPM_ALG_NULL},
807        {(UINT16)(TPMS_SCHEME_MGF1_MARSHAL_REF),
808         (UINT16)(TPMS_SCHEME_KDF1_SP800_56A_MARSHAL_REF),
809         (UINT16)(TPMS_SCHEME_KDF2_MARSHAL_REF),
810         (UINT16)(TPMS_SCHEME_KDF1_SP800_108_MARSHAL_REF),
811         (UINT16)(UINT0_MARSHAL_REF)}
812    },
813    // TPMT_KDF_SCHEME_DATA
814    {STRUCTURE_MTYPE, 2, {
815            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
816                TPMI_ALG_KDF_MARSHAL_REF,
817                (UINT16)(offsetof(TPMT_KDF_SCHEME, scheme)),
818            SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
819                TPMU_KDF_SCHEME_MARSHAL_REF,
820                (UINT16)(offsetof(TPMT_KDF_SCHEME, details))}},
821    // TPMI_ALG_ASYM_SCHEME_DATA
822    {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_VALUE,
823        {TPM_ALG_NULL,
824        RANGE(20, 29, UINT16),
825        ((ALG_RSASSA << 0)    | (ALG_RSAES << 1)     | (ALG_RSAPSS << 2)    |
826         (ALG_OAEP << 3)      | (ALG_ECDSA << 4)     | (ALG_ECDH << 5)      |
827         (ALG_ECDAA << 6)     | (ALG_SM2 << 7)       | (ALG_ECSCHNORR << 8) |
828         (ALG_ECMQV << 9))}},
829    // TPMU_ASYM_SCHEME_DATA
830    {11, 0, (UINT16)(offsetof(TPMU_ASYM_SCHEME_mst, marshalingTypes)),
831        {(UINT32)TPM_ALG_ECDH,      (UINT32)TPM_ALG_ECMQV,
832         (UINT32)TPM_ALG_ECDAA,     (UINT32)TPM_ALG_RSASSA,
833         (UINT32)TPM_ALG_RSAPSS,    (UINT32)TPM_ALG_ECDSA,
834         (UINT32)TPM_ALG_SM2,       (UINT32)TPM_ALG_ECSCHNORR,
835         (UINT32)TPM_ALG_RSAES,     (UINT32)TPM_ALG_OAEP,
836         (UINT32)TPM_ALG_NULL},
837        {(UINT16)(TPMS_KEY_SCHEME_ECDH_MARSHAL_REF),
838         (UINT16)(TPMS_KEY_SCHEME_ECMQV_MARSHAL_REF),
839         (UINT16)(TPMS_SIG_SCHEME_ECDAA_MARSHAL_REF),
840         (UINT16)(TPMS_SIG_SCHEME_RSASSA_MARSHAL_REF),
841         (UINT16)(TPMS_SIG_SCHEME_RSAPSS_MARSHAL_REF),
842         (UINT16)(TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF),
843         (UINT16)(TPMS_SIG_SCHEME_SM2_MARSHAL_REF),
844         (UINT16)(TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_REF),
845         (UINT16)(TPMS_ENC_SCHEME_RSAES_MARSHAL_REF),
846         (UINT16)(TPMS_ENC_SCHEME_OAEP_MARSHAL_REF),
847         (UINT16)(UINT0_MARSHAL_REF)}
848    },
849    // TPMI_ALG_RSA_SCHEME_DATA
850    {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_VALUE,
851        {TPM_ALG_NULL,
852        RANGE(20, 23, UINT16),
853        ((ALG_RSASSA << 0)|(ALG_RSAES << 1)|(ALG_RSAPSS << 2)|(ALG_OAEP << 3))}},
854    // TPMT_RSA_SCHEME_DATA
855    {STRUCTURE_MTYPE, 2, {
856            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
857                TPMI_ALG_RSA_SCHEME_MARSHAL_REF,
858                (UINT16)(offsetof(TPMT_RSA_SCHEME, scheme)),
```

```
859             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
860                 TPMU_ASYM_SCHEME_MARSHAL_REF,
861                 (UINT16)(offsetof(TPMT_RSA_SCHEME, details))}},
862     // TPMI_ALG_RSA_DECRYPT_DATA
863     {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_VALUE,
864         {TPM_ALG_NULL,
865         RANGE(21, 23, UINT16),
866         ((ALG_RSAES << 0)|(ALG_OAEP << 2))}},
867     // TPMT_RSA_DECRYPT_DATA
868     {STRUCTURE_MTYPE, 2, {
869             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
870                 TPMI_ALG_RSA_DECRYPT_MARSHAL_REF,
871                 (UINT16)(offsetof(TPMT_RSA_DECRYPT, scheme)),
872             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
873                 TPMU_ASYM_SCHEME_MARSHAL_REF,
874                 (UINT16)(offsetof(TPMT_RSA_DECRYPT, details))}},
875     // TPM2B_PUBLIC_KEY_RSA_DATA
876     {TPM2B_MTYPE, Type33_MARSHAL_REF},
877     // TPMI_RSA_KEY_BITS_DATA
878     {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 3,
879         {3072*RSA_3072, 1024*RSA_1024, 2048*RSA_2048}},
880     // TPM2B_PRIVATE_KEY_RSA_DATA
881     {TPM2B_MTYPE, Type34_MARSHAL_REF},
882     // TPM2B_ECC_PARAMETER_DATA
883     {TPM2B_MTYPE, Type35_MARSHAL_REF},
884     // TPMS_ECC_POINT_DATA
885     {STRUCTURE_MTYPE, 2, {
886             SET_ELEMENT_TYPE(SIMPLE_STYPE),
887                 TPM2B_ECC_PARAMETER_MARSHAL_REF,
888                 (UINT16)(offsetof(TPMS_ECC_POINT, x)),
889             SET_ELEMENT_TYPE(SIMPLE_STYPE),
890                 TPM2B_ECC_PARAMETER_MARSHAL_REF,
891                 (UINT16)(offsetof(TPMS_ECC_POINT, y))}},
892     // TPM2B_ECC_POINT_DATA
893     {TPM2BS_MTYPE,
894         (UINT8)(offsetof(TPM2B_ECC_POINT, point))|SIZE_EQUAL,
895         UINT16_MARSHAL_REF,
896         TPMS_ECC_POINT_MARSHAL_REF},
897     // TPMI_ALG_ECC_SCHEME_DATA
898     {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SCHEME,
899         {TPM_ALG_NULL,
900         RANGE(24, 29, UINT16),
901         ((ALG_ECDSA << 0)      | (ALG_ECDH << 1)       | (ALG_ECDAA << 2)      |
902          (ALG_SM2 << 3)        | (ALG_ECSCHNORR << 4) | (ALG_ECMQV << 5))}},
903     // TPMI_ECC_CURVE_DATA
904     {MIN_MAX_MTYPE, TWO_BYTES|HAS_BITS, (UINT8)TPM_RC_CURVE,
905         {RANGE(1, 32, UINT16),
906         ((ECC_NIST_P192 << 0) | (ECC_NIST_P224 << 1) | (ECC_NIST_P256 << 2) |
907          (ECC_NIST_P384 << 3) | (ECC_NIST_P521 << 4) | (ECC_BN_P256 << 15)  |
908          (ECC_BN_P638 << 16)  | (ECC_SM2_P256 << 31))}},
909     // TPMT_ECC_SCHEME_DATA
910     {STRUCTURE_MTYPE, 2, {
911             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
912                 TPMI_ALG_ECC_SCHEME_MARSHAL_REF,
913                 (UINT16)(offsetof(TPMT_ECC_SCHEME, scheme)),
914             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
915                 TPMU_ASYM_SCHEME_MARSHAL_REF,
916                 (UINT16)(offsetof(TPMT_ECC_SCHEME, details))}},
917     // TPMS_ALGORITHM_DETAIL_ECC_DATA
918     {STRUCTURE_MTYPE, 11, {
919             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
920                 TPM_ECC_CURVE_MARSHAL_REF,
921                 (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, curveID)),
922             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
923                 UINT16_MARSHAL_REF,
924                 (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, keySize)),
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
925         SET_ELEMENT_TYPE(SIMPLE_STYPE),
926             TPMT_KDF_SCHEME_MARSHAL_REF|NULL_FLAG,
927             (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, kdf)),
928         SET_ELEMENT_TYPE(SIMPLE_STYPE),
929             TPMT_ECC_SCHEME_MARSHAL_REF|NULL_FLAG,
930             (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, sign)),
931         SET_ELEMENT_TYPE(SIMPLE_STYPE),
932             TPM2B_ECC_PARAMETER_MARSHAL_REF,
933             (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, p)),
934         SET_ELEMENT_TYPE(SIMPLE_STYPE),
935             TPM2B_ECC_PARAMETER_MARSHAL_REF,
936             (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, a)),
937         SET_ELEMENT_TYPE(SIMPLE_STYPE),
938             TPM2B_ECC_PARAMETER_MARSHAL_REF,
939             (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, b)),
940         SET_ELEMENT_TYPE(SIMPLE_STYPE),
941             TPM2B_ECC_PARAMETER_MARSHAL_REF,
942             (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, gX)),
943         SET_ELEMENT_TYPE(SIMPLE_STYPE),
944             TPM2B_ECC_PARAMETER_MARSHAL_REF,
945             (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, gY)),
946         SET_ELEMENT_TYPE(SIMPLE_STYPE),
947             TPM2B_ECC_PARAMETER_MARSHAL_REF,
948             (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, n)),
949         SET_ELEMENT_TYPE(SIMPLE_STYPE),
950             TPM2B_ECC_PARAMETER_MARSHAL_REF,
951             (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, h))}},
952 // TPMS_SIGNATURE_RSA_DATA
953 {STRUCTURE_MTYPE, 2, {
954         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
955             TPMI_ALG_HASH_MARSHAL_REF,
956             (UINT16)(offsetof(TPMS_SIGNATURE_RSA, hash)),
957         SET_ELEMENT_TYPE(SIMPLE_STYPE),
958             TPM2B_PUBLIC_KEY_RSA_MARSHAL_REF,
959             (UINT16)(offsetof(TPMS_SIGNATURE_RSA, sig))}},
960 // TPMS_SIGNATURE_ECC_DATA
961 {STRUCTURE_MTYPE, 3, {
962         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
963             TPMI_ALG_HASH_MARSHAL_REF,
964             (UINT16)(offsetof(TPMS_SIGNATURE_ECC, hash)),
965         SET_ELEMENT_TYPE(SIMPLE_STYPE),
966             TPM2B_ECC_PARAMETER_MARSHAL_REF,
967             (UINT16)(offsetof(TPMS_SIGNATURE_ECC, signatureR)),
968         SET_ELEMENT_TYPE(SIMPLE_STYPE),
969             TPM2B_ECC_PARAMETER_MARSHAL_REF,
970             (UINT16)(offsetof(TPMS_SIGNATURE_ECC, signatureS))}},
971 // TPMU_SIGNATURE_DATA
972 {8, 0, (UINT16)(offsetof(TPMU_SIGNATURE_mst, marshalingTypes)),
973     {(UINT32)TPM_ALG_ECDAA,     (UINT32)TPM_ALG_RSASSA,
974      (UINT32)TPM_ALG_RSAPSS,    (UINT32)TPM_ALG_ECDSA,
975      (UINT32)TPM_ALG_SM2,       (UINT32)TPM_ALG_ECSCHNORR,
976      (UINT32)TPM_ALG_HMAC,      (UINT32)TPM_ALG_NULL},
977     {(UINT16)(TPMS_SIGNATURE_ECDAA_MARSHAL_REF),
978      (UINT16)(TPMS_SIGNATURE_RSASSA_MARSHAL_REF),
979      (UINT16)(TPMS_SIGNATURE_RSAPSS_MARSHAL_REF),
980      (UINT16)(TPMS_SIGNATURE_ECDSA_MARSHAL_REF),
981      (UINT16)(TPMS_SIGNATURE_SM2_MARSHAL_REF),
982      (UINT16)(TPMS_SIGNATURE_ECSCHNORR_MARSHAL_REF),
983      (UINT16)(TPMT_HA_MARSHAL_REF),
984      (UINT16)(UINT0_MARSHAL_REF)}
985 },
986 // TPMT_SIGNATURE_DATA
987 {STRUCTURE_MTYPE, 2, {
988         SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
989             TPMI_ALG_SIG_SCHEME_MARSHAL_REF,
990             (UINT16)(offsetof(TPMT_SIGNATURE, sigAlg)),
```

 Copyright © TCG 2006-2020

```
991             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
992                 TPMU_SIGNATURE_MARSHAL_REF,
993                 (UINT16)(offsetof(TPMT_SIGNATURE, signature))}},
994     // TPMU_ENCRYPTED_SECRET_DATA
995     {4, IS_ARRAY_UNION, (UINT16)(offsetof(TPMU_ENCRYPTED_SECRET_mst, marshalingTypes)),
996         {(UINT32)TPM_ALG_ECC,        (UINT32)TPM_ALG_RSA,
997          (UINT32)TPM_ALG_SYMCIPHER, (UINT32)TPM_ALG_KEYEDHASH},
998         {(UINT16)(sizeof(TPMS_ECC_POINT)), (UINT16)(MAX_RSA_KEY_BYTES),
999          (UINT16)(sizeof(TPM2B_DIGEST)),    (UINT16)(sizeof(TPM2B_DIGEST))}
1000    },
1001    // TPM2B_ENCRYPTED_SECRET_DATA
1002    {TPM2B_MTYPE, Type36_MARSHAL_REF},
1003    // TPMI_ALG_PUBLIC_DATA
1004    {MIN_MAX_MTYPE, TWO_BYTES|HAS_BITS, (UINT8)TPM_RC_TYPE,
1005        {RANGE(1, 37, UINT16),
1006        ((ALG_RSA << 0)|(ALG_KEYEDHASH << 7)),
1007        ((ALG_ECC << 2)|(ALG_SYMCIPHER << 4))}},
1008    // TPMU_PUBLIC_ID_DATA
1009    {4, 0, (UINT16)(offsetof(TPMU_PUBLIC_ID_mst, marshalingTypes)),
1010        {(UINT32)TPM_ALG_KEYEDHASH, (UINT32)TPM_ALG_SYMCIPHER,
1011         (UINT32)TPM_ALG_RSA,       (UINT32)TPM_ALG_ECC},
1012        {(UINT16)(TPM2B_DIGEST_MARSHAL_REF),
1013         (UINT16)(TPM2B_DIGEST_MARSHAL_REF),
1014         (UINT16)(TPM2B_PUBLIC_KEY_RSA_MARSHAL_REF),
1015         (UINT16)(TPMS_ECC_POINT_MARSHAL_REF)}
1016    },
1017    // TPMS_KEYEDHASH_PARMS_DATA
1018    {STRUCTURE_MTYPE, 1, {
1019            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1020                TPMT_KEYEDHASH_SCHEME_MARSHAL_REF|NULL_FLAG,
1021                (UINT16)(offsetof(TPMS_KEYEDHASH_PARMS, scheme))}},
1022    // TPMS_RSA_PARMS_DATA
1023    {STRUCTURE_MTYPE, 4, {
1024            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1025                TPMT_SYM_DEF_OBJECT_MARSHAL_REF|NULL_FLAG,
1026                (UINT16)(offsetof(TPMS_RSA_PARMS, symmetric)),
1027            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1028                TPMT_RSA_SCHEME_MARSHAL_REF|NULL_FLAG,
1029                (UINT16)(offsetof(TPMS_RSA_PARMS, scheme)),
1030            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1031                TPMI_RSA_KEY_BITS_MARSHAL_REF,
1032                (UINT16)(offsetof(TPMS_RSA_PARMS, keyBits)),
1033            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1034                UINT32_MARSHAL_REF,
1035                (UINT16)(offsetof(TPMS_RSA_PARMS, exponent))}},
1036    // TPMS_ECC_PARMS_DATA
1037    {STRUCTURE_MTYPE, 4, {
1038            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1039                TPMT_SYM_DEF_OBJECT_MARSHAL_REF|NULL_FLAG,
1040                (UINT16)(offsetof(TPMS_ECC_PARMS, symmetric)),
1041            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1042                TPMT_ECC_SCHEME_MARSHAL_REF|NULL_FLAG,
1043                (UINT16)(offsetof(TPMS_ECC_PARMS, scheme)),
1044            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1045                TPMI_ECC_CURVE_MARSHAL_REF,
1046                (UINT16)(offsetof(TPMS_ECC_PARMS, curveID)),
1047            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1048                TPMT_KDF_SCHEME_MARSHAL_REF|NULL_FLAG,
1049                (UINT16)(offsetof(TPMS_ECC_PARMS, kdf))}},
1050    // TPMU_PUBLIC_PARMS_DATA
1051    {4, 0, (UINT16)(offsetof(TPMU_PUBLIC_PARMS_mst, marshalingTypes)),
1052        {(UINT32)TPM_ALG_KEYEDHASH, (UINT32)TPM_ALG_SYMCIPHER,
1053         (UINT32)TPM_ALG_RSA,       (UINT32)TPM_ALG_ECC},
1054        {(UINT16)(TPMS_KEYEDHASH_PARMS_MARSHAL_REF),
1055         (UINT16)(TPMS_SYMCIPHER_PARMS_MARSHAL_REF),
1056         (UINT16)(TPMS_RSA_PARMS_MARSHAL_REF),
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1057        (UINT16)(TPMS_ECC_PARMS_MARSHAL_REF)}
1058    },
1059    // TPMT_PUBLIC_PARMS_DATA
1060    {STRUCTURE_MTYPE, 2, {
1061            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1062                TPMI_ALG_PUBLIC_MARSHAL_REF,
1063                (UINT16)(offsetof(TPMT_PUBLIC_PARMS, type)),
1064            SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
1065                TPMU_PUBLIC_PARMS_MARSHAL_REF,
1066                (UINT16)(offsetof(TPMT_PUBLIC_PARMS, parameters))}},
1067    // TPMT_PUBLIC_DATA
1068    {STRUCTURE_MTYPE, 6, {
1069            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1070                TPMI_ALG_PUBLIC_MARSHAL_REF,
1071                (UINT16)(offsetof(TPMT_PUBLIC, type)),
1072            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
1073                TPMI_ALG_HASH_MARSHAL_REF,
1074                (UINT16)(offsetof(TPMT_PUBLIC, nameAlg)),
1075            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1076                TPMA_OBJECT_MARSHAL_REF,
1077                (UINT16)(offsetof(TPMT_PUBLIC, objectAttributes)),
1078            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1079                TPM2B_DIGEST_MARSHAL_REF,
1080                (UINT16)(offsetof(TPMT_PUBLIC, authPolicy)),
1081            SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
1082                TPMU_PUBLIC_PARMS_MARSHAL_REF,
1083                (UINT16)(offsetof(TPMT_PUBLIC, parameters)),
1084            SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
1085                TPMU_PUBLIC_ID_MARSHAL_REF,
1086                (UINT16)(offsetof(TPMT_PUBLIC, unique))}},
1087    // TPM2B_PUBLIC_DATA
1088    {TPM2BS_MTYPE,
1089        (UINT8)(offsetof(TPM2B_PUBLIC, publicArea))|SIZE_EQUAL|ELEMENT_PROPAGATE,
1090        UINT16_MARSHAL_REF,
1091        TPMT_PUBLIC_MARSHAL_REF},
1092    // TPM2B_TEMPLATE_DATA
1093    {TPM2B_MTYPE, Type37_MARSHAL_REF},
1094    // TPM2B_PRIVATE_VENDOR_SPECIFIC_DATA
1095    {TPM2B_MTYPE, Type38_MARSHAL_REF},
1096    // TPMU_SENSITIVE_COMPOSITE_DATA
1097    {4, 0, (UINT16)(offsetof(TPMU_SENSITIVE_COMPOSITE_mst, marshalingTypes)),
1098        {(UINT32)TPM_ALG_RSA,       (UINT32)TPM_ALG_ECC,
1099         (UINT32)TPM_ALG_KEYEDHASH, (UINT32)TPM_ALG_SYMCIPHER},
1100        {(UINT16)(TPM2B_PRIVATE_KEY_RSA_MARSHAL_REF),
1101         (UINT16)(TPM2B_ECC_PARAMETER_MARSHAL_REF),
1102         (UINT16)(TPM2B_SENSITIVE_DATA_MARSHAL_REF),
1103         (UINT16)(TPM2B_SYM_KEY_MARSHAL_REF)}
1104    },
1105    // TPMT_SENSITIVE_DATA
1106    {STRUCTURE_MTYPE, 4, {
1107            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1108                TPMI_ALG_PUBLIC_MARSHAL_REF,
1109                (UINT16)(offsetof(TPMT_SENSITIVE, sensitiveType)),
1110            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1111                TPM2B_AUTH_MARSHAL_REF,
1112                (UINT16)(offsetof(TPMT_SENSITIVE, authValue)),
1113            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1114                TPM2B_DIGEST_MARSHAL_REF,
1115                (UINT16)(offsetof(TPMT_SENSITIVE, seedValue)),
1116            SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
1117                TPMU_SENSITIVE_COMPOSITE_MARSHAL_REF,
1118                (UINT16)(offsetof(TPMT_SENSITIVE, sensitive))}},
1119    // TPM2B_SENSITIVE_DATA
1120    {TPM2BS_MTYPE,
1121        (UINT8)(offsetof(TPM2B_SENSITIVE, sensitiveArea)),
1122        UINT16_MARSHAL_REF,
```

 Copyright © TCG 2006-2020

```
1123        TPMT_SENSITIVE_MARSHAL_REF},
1124    // TPM2B_PRIVATE_DATA
1125    {TPM2B_MTYPE, Type39_MARSHAL_REF},
1126    // TPM2B_ID_OBJECT_DATA
1127    {TPM2B_MTYPE, Type40_MARSHAL_REF},
1128    // TPMS_NV_PIN_COUNTER_PARAMETERS_DATA
1129    {STRUCTURE_MTYPE, 2, {
1130            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1131                UINT32_MARSHAL_REF,
1132                (UINT16)(offsetof(TPMS_NV_PIN_COUNTER_PARAMETERS, pinCount)),
1133            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1134                UINT32_MARSHAL_REF,
1135                (UINT16)(offsetof(TPMS_NV_PIN_COUNTER_PARAMETERS, pinLimit))}},
1136    // TPMA_NV_DATA
1137    {ATTRIBUTES_MTYPE, FOUR_BYTES, 0x01F00300},
1138    // TPMS_NV_PUBLIC_DATA
1139    {STRUCTURE_MTYPE, 5, {
1140            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1141                TPMI_RH_NV_INDEX_MARSHAL_REF,
1142                (UINT16)(offsetof(TPMS_NV_PUBLIC, nvIndex)),
1143            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1144                TPMI_ALG_HASH_MARSHAL_REF,
1145                (UINT16)(offsetof(TPMS_NV_PUBLIC, nameAlg)),
1146            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1147                TPMA_NV_MARSHAL_REF,
1148                (UINT16)(offsetof(TPMS_NV_PUBLIC, attributes)),
1149            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1150                TPM2B_DIGEST_MARSHAL_REF,
1151                (UINT16)(offsetof(TPMS_NV_PUBLIC, authPolicy)),
1152            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1153                Type41_MARSHAL_REF,
1154                (UINT16)(offsetof(TPMS_NV_PUBLIC, dataSize))}},
1155    // TPM2B_NV_PUBLIC_DATA
1156    {TPM2BS_MTYPE,
1157        (UINT8)(offsetof(TPM2B_NV_PUBLIC, nvPublic))|SIZE_EQUAL,
1158        UINT16_MARSHAL_REF,
1159        TPMS_NV_PUBLIC_MARSHAL_REF},
1160    // TPM2B_CONTEXT_SENSITIVE_DATA
1161    {TPM2B_MTYPE, Type42_MARSHAL_REF},
1162    // TPMS_CONTEXT_DATA_DATA
1163    {STRUCTURE_MTYPE, 2, {
1164            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1165                TPM2B_DIGEST_MARSHAL_REF,
1166                (UINT16)(offsetof(TPMS_CONTEXT_DATA, integrity)),
1167            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1168                TPM2B_CONTEXT_SENSITIVE_MARSHAL_REF,
1169                (UINT16)(offsetof(TPMS_CONTEXT_DATA, encrypted))}},
1170    // TPM2B_CONTEXT_DATA_DATA
1171    {TPM2B_MTYPE, Type43_MARSHAL_REF},
1172    // TPMS_CONTEXT_DATA
1173    {STRUCTURE_MTYPE, 4, {
1174            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(EIGHT_BYTES),
1175                UINT64_MARSHAL_REF,
1176                (UINT16)(offsetof(TPMS_CONTEXT, sequence)),
1177            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1178                TPMI_DH_SAVED_MARSHAL_REF,
1179                (UINT16)(offsetof(TPMS_CONTEXT, savedHandle)),
1180            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1181                TPMI_RH_HIERARCHY_MARSHAL_REF|NULL_FLAG,
1182                (UINT16)(offsetof(TPMS_CONTEXT, hierarchy)),
1183            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1184                TPM2B_CONTEXT_DATA_MARSHAL_REF,
1185                (UINT16)(offsetof(TPMS_CONTEXT, contextBlob))}},
1186    // TPMS_CREATION_DATA_DATA
1187    {STRUCTURE_MTYPE, 7, {
1188            SET_ELEMENT_TYPE(SIMPLE_STYPE),
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1189                TPML_PCR_SELECTION_MARSHAL_REF,
1190                (UINT16)(offsetof(TPMS_CREATION_DATA, pcrSelect)),
1191            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1192                TPM2B_DIGEST_MARSHAL_REF,
1193                (UINT16)(offsetof(TPMS_CREATION_DATA, pcrDigest)),
1194            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(ONE_BYTES),
1195                TPMA_LOCALITY_MARSHAL_REF,
1196                (UINT16)(offsetof(TPMS_CREATION_DATA, locality)),
1197            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1198                TPM_ALG_ID_MARSHAL_REF,
1199                (UINT16)(offsetof(TPMS_CREATION_DATA, parentNameAlg)),
1200            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1201                TPM2B_NAME_MARSHAL_REF,
1202                (UINT16)(offsetof(TPMS_CREATION_DATA, parentName)),
1203            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1204                TPM2B_NAME_MARSHAL_REF,
1205                (UINT16)(offsetof(TPMS_CREATION_DATA, parentQualifiedName)),
1206            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1207                TPM2B_DATA_MARSHAL_REF,
1208                (UINT16)(offsetof(TPMS_CREATION_DATA, outsideInfo))}},
1209    // TPM2B_CREATION_DATA_DATA
1210    {TPM2BS_MTYPE,
1211        (UINT8)(offsetof(TPM2B_CREATION_DATA, creationData))|SIZE_EQUAL,
1212        UINT16_MARSHAL_REF,
1213        TPMS_CREATION_DATA_MARSHAL_REF},
1214    // TPM_AT_DATA
1215    {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 4,
1216        {TPM_AT_ANY, TPM_AT_ERROR, TPM_AT_PV1, TPM_AT_VEND}},
1217    // TPMS_AC_OUTPUT_DATA
1218    {STRUCTURE_MTYPE, 2, {
1219            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1220                TPM_AT_MARSHAL_REF,
1221                (UINT16)(offsetof(TPMS_AC_OUTPUT, tag)),
1222            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1223                UINT32_MARSHAL_REF,
1224                (UINT16)(offsetof(TPMS_AC_OUTPUT, data))}},
1225    // TPML_AC_CAPABILITIES_DATA
1226    {LIST_MTYPE,
1227        (UINT8)(offsetof(TPML_AC_CAPABILITIES, acCapabilities)),
1228        Type44_MARSHAL_REF,
1229        TPMS_AC_OUTPUT_ARRAY_MARSHAL_INDEX},
1230    // Type00_DATA
1231    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1232        {RANGE(0, sizeof(TPMU_HA), UINT16)}},
1233    // Type01_DATA
1234    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1235        {RANGE(0, sizeof(TPMT_HA), UINT16)}},
1236    // Type02_DATA
1237    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1238        {RANGE(0, 1024, UINT16)}},
1239    // Type03_DATA
1240    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1241        {RANGE(0, MAX_DIGEST_BUFFER, UINT16)}},
1242    // Type04_DATA
1243    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1244        {RANGE(0, MAX_NV_BUFFER_SIZE, UINT16)}},
1245    // Type05_DATA
1246    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1247        {RANGE(0, sizeof(UINT64), UINT16)}},
1248    // Type06_DATA
1249    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1250        {RANGE(0, MAX_SYM_BLOCK_SIZE, UINT16)}},
1251    // Type07_DATA
1252    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1253        {RANGE(0, sizeof(TPMU_NAME), UINT16)}},
1254    // Type08_DATA
```

```
1255    {MIN_MAX_MTYPE, ONE_BYTES, (UINT8)TPM_RC_VALUE,
1256        {RANGE(PCR_SELECT_MIN, PCR_SELECT_MAX, UINT8)}},
1257    // Type10_DATA
1258    {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_TAG, 1,
1259        {TPM_ST_CREATION}},
1260    // Type11_DATA
1261    {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_TAG, 1,
1262        {TPM_ST_VERIFIED}},
1263    // Type12_DATA
1264    {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_TAG, 2,
1265        {TPM_ST_AUTH_SECRET, TPM_ST_AUTH_SIGNED}},
1266    // Type13_DATA
1267    {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_TAG, 1,
1268        {TPM_ST_HASHCHECK}},
1269    // Type15_DATA
1270    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1271        {RANGE(0, MAX_CAP_CC, UINT32)}},
1272    // Type17_DATA
1273    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1274        {RANGE(0, MAX_ALG_LIST_SIZE, UINT32)}},
1275    // Type18_DATA
1276    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1277        {RANGE(0, MAX_CAP_HANDLES, UINT32)}},
1278    // Type19_DATA
1279    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1280        {RANGE(2, 8, UINT32)}},
1281    // Type20_DATA
1282    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1283        {RANGE(0, HASH_COUNT, UINT32)}},
1284    // Type22_DATA
1285    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1286        {RANGE(0, MAX_CAP_ALGS, UINT32)}},
1287    // Type23_DATA
1288    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1289        {RANGE(0, MAX_TPM_PROPERTIES, UINT32)}},
1290    // Type24_DATA
1291    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1292        {RANGE(0, MAX_PCR_PROPERTIES, UINT32)}},
1293    // Type25_DATA
1294    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1295        {RANGE(0, MAX_ECC_CURVES, UINT32)}},
1296    // Type26_DATA
1297    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1298        {RANGE(0, MAX_TAGGED_POLICIES, UINT32)}},
1299    // Type27_DATA
1300    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1301        {RANGE(0, MAX_ACT_DATA, UINT32)}},
1302    // Type28_DATA
1303    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1304        {RANGE(0, sizeof(TPMS_ATTEST), UINT16)}},
1305    // Type29_DATA
1306    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1307        {RANGE(0, MAX_SYM_KEY_BYTES, UINT16)}},
1308    // Type30_DATA
1309    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1310        {RANGE(0, LABEL_MAX_BUFFER, UINT16)}},
1311    // Type31_DATA
1312    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1313        {RANGE(0, sizeof(TPMS_DERIVE), UINT16)}},
1314    // Type32_DATA
1315    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1316        {RANGE(0, sizeof(TPMU_SENSITIVE_CREATE), UINT16)}},
1317    // Type33_DATA
1318    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1319        {RANGE(0, MAX_RSA_KEY_BYTES, UINT16)}},
1320    // Type34_DATA
```

```
1321    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1322        {RANGE(0, RSA_PRIVATE_SIZE, UINT16)}},
1323    // Type35_DATA
1324    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1325        {RANGE(0, MAX_ECC_KEY_BYTES, UINT16)}},
1326    // Type36_DATA
1327    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1328        {RANGE(0, sizeof(TPMU_ENCRYPTED_SECRET), UINT16)}},
1329    // Type37_DATA
1330    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1331        {RANGE(0, sizeof(TPMT_PUBLIC), UINT16)}},
1332    // Type38_DATA
1333    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1334        {RANGE(0, PRIVATE_VENDOR_SPECIFIC_BYTES, UINT16)}},
1335    // Type39_DATA
1336    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1337        {RANGE(0, sizeof(_PRIVATE), UINT16)}},
1338    // Type40_DATA
1339    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1340        {RANGE(0, sizeof(TPMS_ID_OBJECT), UINT16)}},
1341    // Type41_DATA
1342    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1343        {RANGE(0, MAX_NV_INDEX_SIZE, UINT16)}},
1344    // Type42_DATA
1345    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1346        {RANGE(0, MAX_CONTEXT_SIZE, UINT16)}},
1347    // Type43_DATA
1348    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1349        {RANGE(0, sizeof(TPMS_CONTEXT_DATA), UINT16)}},
1350    // Type44_DATA
1351    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1352        {RANGE(0, MAX_AC_CAPABILITIES, UINT32)}}
1353    };
1354    #endif // TABLE_DRIVEN_MARSHAL
```

9.11 MathOnByteBuffers.c

9.11.1 Introduction

This file contains implementation of the math functions that are performed with canonical integers in byte buffers. The canonical integer is big-endian bytes.

```
1   #include "Tpm.h"
```

9.11.2 Functions

9.11.2.1 UnsignedCmpB

This function compare two unsigned values. The values are byte-aligned, big-endian numbers (e.g, a hash).

Return Value	Meaning
1	if (a > b)
0	if (a = b)
-1	if (a < b)

```
2   LIB_EXPORT int
3   UnsignedCompareB(
4       UINT32           aSize,         // IN: size of a
5       const BYTE      *a,             // IN: a
6       UINT32           bSize,         // IN: size of b
7       const BYTE      *b              // IN: b
8       )
9   {
10      UINT32              i;
11      if(aSize > bSize)
12          return 1;
13      else if(aSize < bSize)
14          return -1;
15      else
16      {
17          for(i = 0; i < aSize; i++)
18          {
19              if(a[i] != b[i])
20                  return (a[i] > b[i]) ? 1 : -1;
21          }
22      }
23      return 0;
24  }
```

9.11.2.2 SignedCompareB()

Compare two signed integers:

Return Value	Meaning
1	if a > b
0	if a = b
-1	if a < b

```
25  int
26  SignedCompareB(
27      const UINT32     aSize,          // IN: size of a
28      const BYTE      *a,              // IN: a buffer
29      const UINT32     bSize,          // IN: size of b
30      const BYTE      *b               // IN: b buffer
31      )
32  {
33      int       signA, signB;       // sign of a and b
34
35      // For positive or 0, sign_a is 1
36      // for negative, sign_a is 0
37      signA = ((a[0] & 0x80) == 0) ? 1 : 0;
38
39      // For positive or 0, sign_b is 1
40      // for negative, sign_b is 0
41      signB = ((b[0] & 0x80) == 0) ? 1 : 0;
42
43      if(signA != signB)
44      {
45          return signA - signB;
46      }
47      if(signA == 1)
48          // do unsigned compare function
49          return UnsignedCompareB(aSize, a, bSize, b);
50      else
51          // do unsigned compare the other way
52          return 0 - UnsignedCompareB(aSize, a, bSize, b);
53  }
```

9.11.2.3 ModExpB

This function is used to do modular exponentiation in support of RSA. The most typical uses are: *c* = *m*^*e* mod *n* (RSA encrypt) and *m* = *c*^*d* mod *n* (RSA decrypt). When doing decryption, the *e* parameter of the function will contain the private exponent *d* instead of the public exponent *e*.

If the results will not fit in the provided buffer, an error is returned (CRYPT_ERROR_UNDERFLOW). If the results is smaller than the buffer, the results is de-normalized.

This version is intended for use with RSA and requires that *m* be less than *n*.

Error Returns	Meaning
TPM_RC_SIZE	number to exponentiate is larger than the modulus
TPM_RC_NO_RESULT	result will not fit into the provided buffer

```
54  TPM_RC
55  ModExpB(
56      UINT32           cSize,          // IN: the size of the output buffer. It will
57                                       //     need to be the same size as the modulus
58      BYTE            *c,              // OUT: the buffer to receive the results
59                                       //     (c->size must be set to the maximum size
60                                       //     for the returned value)
61      const UINT32     mSize,
62      const BYTE      *m,              // IN: number to exponentiate
```

 Copyright © TCG 2006-2020

```
63      const UINT32     eSize,
64      const BYTE      *e,              // IN: power
65      const UINT32     nSize,
66      const BYTE      *n               // IN: modulus
67      )
68  {
69      BN_MAX(bnC);
70      BN_MAX(bnM);
71      BN_MAX(bnE);
72      BN_MAX(bnN);
73      NUMBYTES         tSize = (NUMBYTES)nSize;
74      TPM_RC           retVal = TPM_RC_SUCCESS;
75
76      // Convert input parameters
77      BnFromBytes(bnM, m, (NUMBYTES)mSize);
78      BnFromBytes(bnE, e, (NUMBYTES)eSize);
79      BnFromBytes(bnN, n, (NUMBYTES)nSize);
80
81      // Make sure that the output is big enough to hold the result
82      // and that 'm' is less than 'n' (the modulus)
83      if(cSize < nSize)
84          ERROR_RETURN(TPM_RC_NO_RESULT);
85      if(BnUnsignedCmp(bnM, bnN) >= 0)
86          ERROR_RETURN(TPM_RC_SIZE);
87      BnModExp(bnC, bnM, bnE, bnN);
88      BnToBytes(bnC, c, &tSize);
89  Exit:
90      return retVal;
91  }
```

9.11.2.4 DivideB()

Divide an integer (*n*) by an integer (*d*) producing a quotient (*q*) and a remainder (*r*). If *q* or *r* is not needed, then the pointer to them may be set to NULL.

Error Returns	Meaning
TPM_RC_NO_RESULT	*q* or *r* is too small to receive the result

```
92  LIB_EXPORT TPM_RC
93  DivideB(
94      const TPM2B     *n,              // IN: numerator
95      const TPM2B     *d,              // IN: denominator
96      TPM2B           *q,              // OUT: quotient
97      TPM2B           *r               // OUT: remainder
98      )
99  {
100     BN_MAX_INITIALIZED(bnN, n);
101     BN_MAX_INITIALIZED(bnD, d);
102     BN_MAX(bnQ);
103     BN_MAX(bnR);
104 //
105     // Do divide with converted values
106     BnDiv(bnQ, bnR, bnN, bnD);
107
108     // Convert the BIGNUM result back to 2B format using the size of the original
109     // number
110     if(q != NULL)
111         if(!BnTo2B(bnQ, q, q->size))
112             return TPM_RC_NO_RESULT;
113     if(r != NULL)
114         if(!BnTo2B(bnR, r, r->size))
115             return TPM_RC_NO_RESULT;
116     return TPM_RC_SUCCESS;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
117     }
```

9.11.2.5 AdjustNumberB()

Remove/add leading zeros from a number in a TPM2B. Will try to make the number by adding or removing leading zeros. If the number is larger than the requested size, it will make the number as small as possible. Setting *requestedSize* to zero is equivalent to requesting that the number be normalized.

```
118     UINT16
119     AdjustNumberB(
120         TPM2B            *num,
121         UINT16            requestedSize
122         )
123     {
124         BYTE             *from;
125         UINT16            i;
126         // See if number is already the requested size
127         if(num->size == requestedSize)
128             return requestedSize;
129         from = num->buffer;
130         if (num->size > requestedSize)
131         {
132         // This is a request to shift the number to the left (remove leading zeros)
133             // Find the first non-zero byte. Don't look past the point where removing
134             // more zeros would make the number smaller than requested, and don't throw
135             // away any significant digits.
136             for(i = num->size; *from == 0 && i > requestedSize; from++, i--);
137             if(i < num->size)
138             {
139                 num->size = i;
140                 MemoryCopy(num->buffer, from, i);
141             }
142         }
143         // This is a request to shift the number to the right (add leading zeros)
144         else
145         {
146             MemoryCopy(&num->buffer[requestedSize - num->size], num->buffer, num->size);
147             MemorySet(num->buffer, 0, requestedSize- num->size);
148             num->size = requestedSize;
149         }
150         return num->size;
151     }
```

9.11.2.6 ShiftLeft()

This function shifts a byte buffer (a TPM2B) one byte to the left. That is, the most significant bit of the most significant byte is lost.

```
152     TPM2B *
153     ShiftLeft(
154         TPM2B        *value          // IN/OUT: value to shift and shifted value out
155     )
156     {
157         UINT16        count = value->size;
158         BYTE         *buffer = value->buffer;
159         if(count > 0)
160         {
161             for(count -= 1; count > 0; buffer++, count--)
162             {
163                 buffer[0] = (buffer[0] << 1) + ((buffer[1] & 0x80) ? 1 : 0);
164             }
165             *buffer <<= 1;
```

```
166     }
167     return value;
168 }
```

9.12 Memory.c

9.12.1 Description

This file contains a set of miscellaneous memory manipulation routines. Many of the functions have the same semantics as functions defined in string.h. Those functions are not used directly in the TPM because they are not *safe*

This version uses string.h after adding guards. This is because the math libraries invariably use those functions so it is not practical to prevent those library functions from being pulled into the build.

9.12.2 Includes and Data Definitions

```
1   #include "Tpm.h"
2   #include "Memory_fp.h"
```

9.12.3 Functions

9.12.3.1 MemoryCopy()

This is an alias for memmove. This is used in place of memcpy because some of the moves may overlap and rather than try to make sure that memmove is used when necessary, it is always used.

```
3   void
4   MemoryCopy(
5       void          *dest,
6       const void    *src,
7       int            sSize
8       )
9   {
10      if(dest != src)
11          memmove(dest, src, sSize);
12  }
```

9.12.3.2 MemoryEqual()

This function indicates if two buffers have the same values in the indicated number of bytes.

Return Value	Meaning
TRUE(1)	all octets are the same
FALSE(0)	all octets are not the same

```
13  BOOL
14  MemoryEqual(
15      const void      *buffer1,       // IN: compare buffer1
16      const void      *buffer2,       // IN: compare buffer2
17      unsigned int     size           // IN: size of bytes being compared
18      )
19  {
20      BYTE          equal = 0;
21      const BYTE   *b1 = (BYTE *)buffer1;
22      const BYTE   *b2 = (BYTE *)buffer2;
23  //
24      // Compare all bytes so that there is no leakage of information
25      // due to timing differences.
26      for(; size > 0; size--)
```

```
27          equal |= (*b1++ ^ *b2++);
28      return (equal == 0);
29  }
```

9.12.3.3 MemoryCopy2B()

This function copies a TPM2B. This can be used when the TPM2B types are the same or different.

This function returns the number of octets in the data buffer of the TPM2B.

```
30  LIB_EXPORT INT16
31  MemoryCopy2B(
32      TPM2B           *dest,          // OUT: receiving TPM2B
33      const TPM2B     *source,        // IN: source TPM2B
34      unsigned int     dSize          // IN: size of the receiving buffer
35      )
36  {
37      pAssert(dest != NULL);
38      if(source == NULL)
39          dest->size = 0;
40      else
41      {
42          pAssert(source->size <= dSize);
43          MemoryCopy(dest->buffer, source->buffer, source->size);
44          dest->size = source->size;
45      }
46      return dest->size;
47  }
```

9.12.3.4 MemoryConcat2B()

This function will concatenate the buffer contents of a TPM2B to an the buffer contents of another TPM2B and adjust the size accordingly (*a* := (*a* | *b*)).

```
48  void
49  MemoryConcat2B(
50      TPM2B           *aInOut,        // IN/OUT: destination 2B
51      TPM2B           *bIn,           // IN: second 2B
52      unsigned int     aMaxSize       // IN: The size of aInOut.buffer (max values for
53                                      //     aInOut.size)
54      )
55  {
56      pAssert(bIn->size <= aMaxSize - aInOut->size);
57      MemoryCopy(&aInOut->buffer[aInOut->size], &bIn->buffer, bIn->size);
58      aInOut->size = aInOut->size + bIn->size;
59      return;
60  }
```

9.12.3.5 MemoryEqual2B()

This function will compare two TPM2B structures. To be equal, they need to be the same size and the buffer contexts need to be the same in all octets.

Return Value	Meaning
TRUE(1)	size and buffer contents are the same
FALSE(0)	size or buffer contents are not the same

```
61  BOOL
62  MemoryEqual2B(
```

```
63      const TPM2B     *aIn,           // IN: compare value
64      const TPM2B     *bIn            // IN: compare value
65      )
66  {
67      if(aIn->size != bIn->size)
68          return FALSE;
69      return MemoryEqual(aIn->buffer, bIn->buffer, aIn->size);
70  }
```

9.12.3.6 MemorySet()

This function will set all the octets in the specified memory range to the specified octet value.

NOTE: A previous version had an additional parameter (*dSize*) that was intended to make sure that the destination would not be overrun. The problem is that, in use, all that was happening was that the value of size was used for *dSize* so there was no benefit in the extra parameter.

```
71  void
72  MemorySet(
73      void            *dest,
74      int              value,
75      size_t           size
76      )
77  {
78      memset(dest, value, size);
79  }
```

9.12.3.7 MemoryPad2B()

Function to pad a TPM2B with zeros and adjust the size.

```
80  void
81  MemoryPad2B(
82      TPM2B           *b,
83      UINT16           newSize
84      )
85  {
86      MemorySet(&b->buffer[b->size], 0, newSize - b->size);
87      b->size = newSize;
88  }
```

9.12.3.8 Uint16ToByteArray()

Function to write an integer to a byte array

```
89  void
90  Uint16ToByteArray(
91      UINT16              i,
92      BYTE                *a
93      )
94  {
95      a[1] = (BYTE)(i); i >>= 8;
96      a[0] = (BYTE)(i);
97  }
```

9.12.3.9 Uint32ToByteArray()

Function to write an integer to a byte array

```
98  void
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
 99     Uint32ToByteArray(
100         UINT32              i,
101         BYTE                *a
102         )
103     {
104         a[3] = (BYTE)(i); i >>= 8;
105         a[2] = (BYTE)(i); i >>= 8;
106         a[1] = (BYTE)(i); i >>= 8;
107         a[0] = (BYTE)(i);
108     }
```

9.12.3.10 Uint64ToByteArray()

Function to write an integer to a byte array

```
109     void
110     Uint64ToByteArray(
111         UINT64               i,
112         BYTE                *a
113         )
114     {
115         a[7] = (BYTE)(i); i >>= 8;
116         a[6] = (BYTE)(i); i >>= 8;
117         a[5] = (BYTE)(i); i >>= 8;
118         a[4] = (BYTE)(i); i >>= 8;
119         a[3] = (BYTE)(i); i >>= 8;
120         a[2] = (BYTE)(i); i >>= 8;
121         a[1] = (BYTE)(i); i >>= 8;
122         a[0] = (BYTE)(i);
123     }
```

9.12.3.11 ByteArrayToUint8()

Function to write a UINT8 to a byte array. This is included for completeness and to allow certain macro expansions

```
124     UINT8
125     ByteArrayToUint8(
126         BYTE                *a
127     )
128     {
129         return          *a;
130     }
```

9.12.3.12 ByteArrayToUint16()

Function to write an integer to a byte array

```
131     UINT16
132     ByteArrayToUint16(
133         BYTE                *a
134     )
135     {
136         return ((UINT16)a[0] << 8) + a[1];
137     }
```

9.12.3.13 ByteArrayToUint32()

Function to write an integer to a byte array

```
138 UINT32
139 ByteArrayToUint32(
140     BYTE                 *a
141 )
142 {
143     return (UINT32)((((((UINT32)a[0] << 8) + a[1]) << 8) + (UINT32)a[2]) << 8) + a[3];
144 }
```

9.12.3.14 ByteArrayToUint64()

Function to write an integer to a byte array

```
145 UINT64
146 ByteArrayToUint64(
147     BYTE                 *a
148     )
149 {
150     return (((UINT64)BYTE_ARRAY_TO_UINT32(a)) << 32) + BYTE_ARRAY_TO_UINT32(&a[4]);
151 }
```

9.13 Power.c

9.13.1 Description

This file contains functions that receive the simulated power state transitions of the TPM.

9.13.2 Includes and Data Definitions

```
1   #define POWER_C
2   #include "Tpm.h"
```

9.13.3 Functions

9.13.3.1 TPMInit()

This function is used to process a power on event.

```
3   void
4   TPMInit(
5       void
6       )
7   {
8       // Set state as not initialized. This means that Startup is required
9       g_initialized = FALSE;
10      return;
11  }
```

9.13.3.2 TPMRegisterStartup()

This function registers the fact that the TPM has been initialized (a TPM2_Startup() has completed successfully).

```
12  BOOL
13  TPMRegisterStartup(
14      void
15      )
16  {
17      g_initialized = TRUE;
18      return TRUE;
19  }
```

9.13.3.3 TPMIsStarted()

Indicates if the TPM has been initialized (a TPM2_Startup() has completed successfully after a _TPM_Init()).

Return Value	Meaning
TRUE(1)	TPM has been initialized
FALSE(0)	TPM has not been initialized

```
20  BOOL
21  TPMIsStarted(
22      void
23      )
24  {
```

 Copyright © TCG 2006-2020

```
25      return g_initialized;
26  }
```

9.14 PropertyCap.c

9.14.1 Description

This file contains the functions that are used for accessing the TPM_CAP_TPM_PROPERTY values.

9.14.2 Includes

```
1    #include "Tpm.h"
```

9.14.3 Functions

9.14.3.1 TPMPropertyIsDefined()

This function accepts a property selection and, if so, sets *value* to the value of the property.

All the fixed values are vendor dependent or determined by a platform-specific specification. The values in the table below are examples and should be changed by the vendor.

Return Value	Meaning
TRUE(1)	referenced property exists and *value* set
FALSE(0)	referenced property does not exist

```
2    static BOOL
3    TPMPropertyIsDefined(
4        TPM_PT              property,       // IN: property
5        UINT32             *value           // OUT: property value
6        )
7    {
8        switch(property)
9        {
10           case TPM_PT_FAMILY_INDICATOR:
11               // from the title page of the specification
12               // For this specification, the value is "2.0".
13               *value = TPM_SPEC_FAMILY;
14               break;
15           case TPM_PT_LEVEL:
16               // from the title page of the specification
17               *value = TPM_SPEC_LEVEL;
18               break;
19           case TPM_PT_REVISION:
20               // from the title page of the specification
21               *value = TPM_SPEC_VERSION;
22               break;
23           case TPM_PT_DAY_OF_YEAR:
24               // computed from the date value on the title page of the specification
25               *value = TPM_SPEC_DAY_OF_YEAR;
26               break;
27           case TPM_PT_YEAR:
28               // from the title page of the specification
29               *value = TPM_SPEC_YEAR;
30               break;
31           case TPM_PT_MANUFACTURER:
32               // vendor ID unique to each TPM manufacturer
33               *value = BYTE_ARRAY_TO_UINT32(MANUFACTURER);
34               break;
35           case TPM_PT_VENDOR_STRING_1:
36               // first four characters of the vendor ID string
37               *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_1);
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
 38                 break;
 39             case TPM_PT_VENDOR_STRING_2:
 40                 // second four characters of the vendor ID string
 41     #ifdef VENDOR_STRING_2
 42                 *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_2);
 43     #else
 44                 *value = 0;
 45     #endif
 46                 break;
 47             case TPM_PT_VENDOR_STRING_3:
 48                 // third four characters of the vendor ID string
 49     #ifdef VENDOR_STRING_3
 50                 *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_3);
 51     #else
 52                 *value = 0;
 53     #endif
 54                 break;
 55             case TPM_PT_VENDOR_STRING_4:
 56                 // fourth four characters of the vendor ID string
 57     #ifdef VENDOR_STRING_4
 58                 *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_4);
 59     #else
 60                 *value = 0;
 61     #endif
 62                 break;
 63             case TPM_PT_VENDOR_TPM_TYPE:
 64                 // vendor-defined value indicating the TPM model
 65                 *value = 1;
 66                 break;
 67             case TPM_PT_FIRMWARE_VERSION_1:
 68                 // more significant 32-bits of a vendor-specific value
 69                 *value = gp.firmwareV1;
 70                 break;
 71             case TPM_PT_FIRMWARE_VERSION_2:
 72                 // less significant 32-bits of a vendor-specific value
 73                 *value = gp.firmwareV2;
 74                 break;
 75             case TPM_PT_INPUT_BUFFER:
 76                 // maximum size of TPM2B_MAX_BUFFER
 77                 *value = MAX_DIGEST_BUFFER;
 78                 break;
 79             case TPM_PT_HR_TRANSIENT_MIN:
 80                 // minimum number of transient objects that can be held in TPM
 81                 // RAM
 82                 *value = MAX_LOADED_OBJECTS;
 83                 break;
 84             case TPM_PT_HR_PERSISTENT_MIN:
 85                 // minimum number of persistent objects that can be held in
 86                 // TPM NV memory
 87                 // In this implementation, there is no minimum number of
 88                 // persistent objects.
 89                 *value = MIN_EVICT_OBJECTS;
 90                 break;
 91             case TPM_PT_HR_LOADED_MIN:
 92                 // minimum number of authorization sessions that can be held in
 93                 // TPM RAM
 94                 *value = MAX_LOADED_SESSIONS;
 95                 break;
 96             case TPM_PT_ACTIVE_SESSIONS_MAX:
 97                 // number of authorization sessions that may be active at a time
 98                 *value = MAX_ACTIVE_SESSIONS;
 99                 break;
100             case TPM_PT_PCR_COUNT:
101                 // number of PCR implemented
102                 *value = IMPLEMENTATION_PCR;
103                 break;
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
104         case TPM_PT_PCR_SELECT_MIN:
105             // minimum number of bytes in a TPMS_PCR_SELECT.sizeOfSelect
106             *value = PCR_SELECT_MIN;
107             break;
108         case TPM_PT_CONTEXT_GAP_MAX:
109             // maximum allowed difference (unsigned) between the contextID
110             // values of two saved session contexts
111             *value = ((UINT32)1 << (sizeof(CONTEXT_SLOT) * 8)) - 1;
112             break;
113         case TPM_PT_NV_COUNTERS_MAX:
114             // maximum number of NV indexes that are allowed to have the
115             // TPMA_NV_COUNTER attribute SET
116             // In this implementation, there is no limitation on the number
117             // of counters, except for the size of the NV Index memory.
118             *value = 0;
119             break;
120         case TPM_PT_NV_INDEX_MAX:
121             // maximum size of an NV index data area
122             *value = MAX_NV_INDEX_SIZE;
123             break;
124         case TPM_PT_MEMORY:
125             // a TPMA_MEMORY indicating the memory management method for the TPM
126         {
127             union
128             {
129                 TPMA_MEMORY     att;
130                 UINT32          u32;
131             } attributes = { TPMA_ZERO_INITIALIZER() };
132             SET_ATTRIBUTE(attributes.att, TPMA_MEMORY, sharedNV);
133             SET_ATTRIBUTE(attributes.att, TPMA_MEMORY, objectCopiedToRam);
134
135             // Note: For a LSb0 machine, the bits in a bit field are in the correct
136             // order even if the machine is MSB0. For a MSb0 machine, a TPMA will
137             // be an integer manipulated by masking (USE_BIT_FIELD_STRUCTURES will
138             // be NO) so the bits are manipulate correctly.
139             *value = attributes.u32;
140             break;
141         }
142         case TPM_PT_CLOCK_UPDATE:
143             // interval, in seconds, between updates to the copy of
144             // TPMS_TIME_INFO .clock in NV
145             *value = (1 << NV_CLOCK_UPDATE_INTERVAL);
146             break;
147         case TPM_PT_CONTEXT_HASH:
148             // algorithm used for the integrity hash on saved contexts and
149             // for digesting the fuData of TPM2_FirmwareRead()
150             *value = CONTEXT_INTEGRITY_HASH_ALG;
151             break;
152         case TPM_PT_CONTEXT_SYM:
153             // algorithm used for encryption of saved contexts
154             *value = CONTEXT_ENCRYPT_ALG;
155             break;
156         case TPM_PT_CONTEXT_SYM_SIZE:
157             // size of the key used for encryption of saved contexts
158             *value = CONTEXT_ENCRYPT_KEY_BITS;
159             break;
160         case TPM_PT_ORDERLY_COUNT:
161             // maximum difference between the volatile and non-volatile
162             // versions of TPMA_NV_COUNTER that have TPMA_NV_ORDERLY SET
163             *value = MAX_ORDERLY_COUNT;
164             break;
165         case TPM_PT_MAX_COMMAND_SIZE:
166             // maximum value for 'commandSize'
167             *value = MAX_COMMAND_SIZE;
168             break;
169         case TPM_PT_MAX_RESPONSE_SIZE:
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
170                 // maximum value for 'responseSize'
171                 *value = MAX_RESPONSE_SIZE;
172                 break;
173             case TPM_PT_MAX_DIGEST:
174                 // maximum size of a digest that can be produced by the TPM
175                 *value = sizeof(TPMU_HA);
176                 break;
177             case TPM_PT_MAX_OBJECT_CONTEXT:
178     // Header has 'sequence', 'handle' and 'hierarchy'
179     #define SIZE_OF_CONTEXT_HEADER                          \
180         sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) + sizeof(TPMI_RH_HIERARCHY)
181     #define SIZE_OF_CONTEXT_INTEGRITY (sizeof(UINT16) + CONTEXT_INTEGRITY_HASH_SIZE)
182     #define SIZE_OF_FINGERPRINT     sizeof(UINT64)
183     #define SIZE_OF_CONTEXT_BLOB_OVERHEAD                       \
184             (sizeof(UINT16)  + SIZE_OF_CONTEXT_INTEGRITY + SIZE_OF_FINGERPRINT)
185     #define SIZE_OF_CONTEXT_OVERHEAD                            \
186             (SIZE_OF_CONTEXT_HEADER + SIZE_OF_CONTEXT_BLOB_OVERHEAD)
187     #if 0
188                 // maximum size of a TPMS_CONTEXT that will be returned by
189                 // TPM2_ContextSave for object context
190                 *value = 0;
191                 // adding sequence, saved handle and hierarchy
192                 *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
193                     sizeof(TPMI_RH_HIERARCHY);
194                 // add size field in TPM2B_CONTEXT
195                 *value += sizeof(UINT16);
196                 // add integrity hash size
197                 *value += sizeof(UINT16) +
198                     CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
199                 // Add fingerprint size, which is the same as sequence size
200                 *value += sizeof(UINT64);
201                 // Add OBJECT structure size
202                 *value += sizeof(OBJECT);
203     #else
204                 // the maximum size of a TPMS_CONTEXT that will be returned by
205                 // TPM2_ContextSave for object context
206                 *value = SIZE_OF_CONTEXT_OVERHEAD + sizeof(OBJECT);
207     #endif
208                 break;
209             case TPM_PT_MAX_SESSION_CONTEXT:
210     #if 0
211
212                 // the maximum size of a TPMS_CONTEXT that will be returned by
213                 // TPM2_ContextSave for object context
214                 *value = 0;
215                 // adding sequence, saved handle and hierarchy
216                 *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
217                     sizeof(TPMI_RH_HIERARCHY);
218                 // Add size field in TPM2B_CONTEXT
219                 *value += sizeof(UINT16);
220     // Add integrity hash size
221                 *value += sizeof(UINT16) +
222                     CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
223           // Add fingerprint size, which is the same as sequence size
224                 *value += sizeof(UINT64);
225                 // Add SESSION structure size
226                 *value += sizeof(SESSION);
227     #else
228                  // the maximum size of a TPMS_CONTEXT that will be returned by
229                 // TPM2_ContextSave for object context
230                 *value = SIZE_OF_CONTEXT_OVERHEAD + sizeof(SESSION);
231     #endif
232                 break;
233             case TPM_PT_PS_FAMILY_INDICATOR:
234                 // platform specific values for the TPM_PT_PS parameters from
235                 // the relevant platform-specific specification
```

```
236                 // In this reference implementation, all of these values are 0.
237                 *value = PLATFORM_FAMILY;
238                 break;
239             case TPM_PT_PS_LEVEL:
240                 // level of the platform-specific specification
241                 *value = PLATFORM_LEVEL;
242                 break;
243             case TPM_PT_PS_REVISION:
244                 // specification Revision times 100 for the platform-specific
245                 // specification
246                 *value = PLATFORM_VERSION;
247                 break;
248             case TPM_PT_PS_DAY_OF_YEAR:
249                 // platform-specific specification day of year using TCG calendar
250                 *value = PLATFORM_DAY_OF_YEAR;
251                 break;
252             case TPM_PT_PS_YEAR:
253                 // platform-specific specification year using the CE
254                 *value = PLATFORM_YEAR;
255                 break;
256             case TPM_PT_SPLIT_MAX:
257                 // number of split signing operations supported by the TPM
258                 *value = 0;
259     #if ALG_ECC
260                 *value = sizeof(gr.commitArray) * 8;
261     #endif
262                 break;
263             case TPM_PT_TOTAL_COMMANDS:
264                 // total number of commands implemented in the TPM
265                 // Since the reference implementation does not have any
266                 // vendor-defined commands, this will be the same as the
267                 // number of library commands.
268             {
269     #if COMPRESSED_LISTS
270                 (*value) = COMMAND_COUNT;
271     #else
272                 COMMAND_INDEX       commandIndex;
273                 *value = 0;
274
275                 // scan all implemented commands
276                 for(commandIndex = GetClosestCommandIndex(0);
277                 commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
278                     commandIndex = GetNextCommandIndex(commandIndex))
279                 {
280                     (*value)++;     // count of all implemented
281                 }
282     #endif
283                 break;
284             }
285             case TPM_PT_LIBRARY_COMMANDS:
286                 // number of commands from the TPM library that are implemented
287             {
288     #if COMPRESSED_LISTS
289                 *value = LIBRARY_COMMAND_ARRAY_SIZE;
290     #else
291                 COMMAND_INDEX       commandIndex;
292                 *value = 0;
293
294                 // scan all implemented commands
295                 for(commandIndex = GetClosestCommandIndex(0);
296                 commandIndex < LIBRARY_COMMAND_ARRAY_SIZE;
297                     commandIndex = GetNextCommandIndex(commandIndex))
298                 {
299                     (*value)++;
300                 }
301     #endif
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
302                 break;
303             }
304         case TPM_PT_VENDOR_COMMANDS:
305             // number of vendor commands that are implemented
306             *value = VENDOR_COMMAND_ARRAY_SIZE;
307             break;
308         case TPM_PT_NV_BUFFER_MAX:
309             // Maximum data size in an NV write command
310             *value = MAX_NV_BUFFER_SIZE;
311             break;
312         case TPM_PT_MODES:
313 #if FIPS_COMPLIANT
314             *value = 1;
315 #else
316             *value = 0;
317 #endif
318             break;
319         case TPM_PT_MAX_CAP_BUFFER:
320             *value = MAX_CAP_BUFFER;
321             break;
322
323         // Start of variable commands
324         case TPM_PT_PERMANENT:
325             // TPMA_PERMANENT
326         {
327             union {
328                 TPMA_PERMANENT      attr;
329                 UINT32              u32;
330             } flags = { TPMA_ZERO_INITIALIZER() };
331             if(gp.ownerAuth.t.size != 0)
332                 SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, ownerAuthSet);
333             if(gp.endorsementAuth.t.size != 0)
334                 SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, endorsementAuthSet);
335             if(gp.lockoutAuth.t.size != 0)
336                 SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, lockoutAuthSet);
337             if(gp.disableClear)
338                 SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, disableClear);
339             if(gp.failedTries >= gp.maxTries)
340                 SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, inLockout);
341             // In this implementation, EPS is always generated by TPM
342             SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, tpmGeneratedEPS);
343
344             // Note: For a LSb0 machine, the bits in a bit field are in the correct
345             // order even if the machine is MSB0. For a MSb0 machine, a TPMA will
346             // be an integer manipulated by masking (USE_BIT_FIELD_STRUCTURES will
347             // be NO) so the bits are manipulate correctly.
348             *value = flags.u32;
349             break;
350         }
351         case TPM_PT_STARTUP_CLEAR:
352             // TPMA_STARTUP_CLEAR
353         {
354             union {
355                 TPMA_STARTUP_CLEAR  attr;
356                 UINT32              u32;
357             } flags = { TPMA_ZERO_INITIALIZER() };
358 //
359             if(g_phEnable)
360                 SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, phEnable);
361             if(gc.shEnable)
362                 SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, shEnable);
363             if(gc.ehEnable)
364                 SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, ehEnable);
365             if(gc.phEnableNV)
366                 SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, phEnableNV);
367             if(g_prevOrderlyState != SU_NONE_VALUE)
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
368                     SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, orderly);
369
370                 // Note: For a LSb0 machine, the bits in a bit field are in the correct
371                 // order even if the machine is MSB0. For a MSb0 machine, a TPMA will
372                 // be an integer manipulated by masking (USE_BIT_FIELD_STRUCTURES will
373                 // be NO) so the bits are manipulate correctly.
374                 *value = flags.u32;
375                 break;
376             }
377             case TPM_PT_HR_NV_INDEX:
378                 // number of NV indexes currently defined
379                 *value = NvCapGetIndexNumber();
380                 break;
381             case TPM_PT_HR_LOADED:
382                 // number of authorization sessions currently loaded into TPM
383                 // RAM
384                 *value = SessionCapGetLoadedNumber();
385                 break;
386             case TPM_PT_HR_LOADED_AVAIL:
387                 // number of additional authorization sessions, of any type,
388                 // that could be loaded into TPM RAM
389                 *value = SessionCapGetLoadedAvail();
390                 break;
391             case TPM_PT_HR_ACTIVE:
392                 // number of active authorization sessions currently being
393                 // tracked by the TPM
394                 *value = SessionCapGetActiveNumber();
395                 break;
396             case TPM_PT_HR_ACTIVE_AVAIL:
397                 // number of additional authorization sessions, of any type,
398                 // that could be created
399                 *value = SessionCapGetActiveAvail();
400                 break;
401             case TPM_PT_HR_TRANSIENT_AVAIL:
402                 // estimate of the number of additional transient objects that
403                 // could be loaded into TPM RAM
404                 *value = ObjectCapGetTransientAvail();
405                 break;
406             case TPM_PT_HR_PERSISTENT:
407                 // number of persistent objects currently loaded into TPM
408                 // NV memory
409                 *value = NvCapGetPersistentNumber();
410                 break;
411             case TPM_PT_HR_PERSISTENT_AVAIL:
412                 // number of additional persistent objects that could be loaded
413                 // into NV memory
414                 *value = NvCapGetPersistentAvail();
415                 break;
416             case TPM_PT_NV_COUNTERS:
417                 // number of defined NV indexes that have NV TPMA_NV_COUNTER
418                 // attribute SET
419                 *value = NvCapGetCounterNumber();
420                 break;
421             case TPM_PT_NV_COUNTERS_AVAIL:
422                 // number of additional NV indexes that can be defined with their
423                 // TPMA_NV_COUNTER attribute SET
424                 *value = NvCapGetCounterAvail();
425                 break;
426             case TPM_PT_ALGORITHM_SET:
427                 // region code for the TPM
428                 *value = gp.algorithmSet;
429                 break;
430             case TPM_PT_LOADED_CURVES:
431 #if ALG_ECC
432             // number of loaded ECC curves
433                 *value = ECC_CURVE_COUNT;
```

```
434 #else // ALG_ECC
435             *value = 0;
436 #endif // ALG_ECC
437             break;
438         case TPM_PT_LOCKOUT_COUNTER:
439             // current value of the lockout counter
440             *value = gp.failedTries;
441             break;
442         case TPM_PT_MAX_AUTH_FAIL:
443             // number of authorization failures before DA lockout is invoked
444             *value = gp.maxTries;
445             break;
446         case TPM_PT_LOCKOUT_INTERVAL:
447             // number of seconds before the value reported by
448             // TPM_PT_LOCKOUT_COUNTER is decremented
449             *value = gp.recoveryTime;
450             break;
451         case TPM_PT_LOCKOUT_RECOVERY:
452             // number of seconds after a lockoutAuth failure before use of
453             // lockoutAuth may be attempted again
454             *value = gp.lockoutRecovery;
455             break;
456         case TPM_PT_NV_WRITE_RECOVERY:
457             // number of milliseconds before the TPM will accept another command
458             // that will modify NV.
459             // This should make a call to the platform code that is doing rate
460             // limiting of NV. Rate limiting is not implemented in the reference
461             // code so no call is made.
462             *value = 0;
463             break;
464         case TPM_PT_AUDIT_COUNTER_0:
465             // high-order 32 bits of the command audit counter
466             *value = (UINT32)(gp.auditCounter >> 32);
467             break;
468         case TPM_PT_AUDIT_COUNTER_1:
469             // low-order 32 bits of the command audit counter
470             *value = (UINT32)(gp.auditCounter);
471             break;
472         default:
473             // property is not defined
474             return FALSE;
475             break;
476     }
477     return TRUE;
478 }
```

9.14.3.2 TPMCapGetProperties()

This function is used to get the TPM_PT values. The search of properties will start at *property* and continue until *propertyList* has as many values as will fit, or the last property has been reported, or the list has as many values as requested in *count*.

Return Value	Meaning
YES	more properties are available
NO	no more properties to be reported

```
479 TPMI_YES_NO
480 TPMCapGetProperties(
481     TPM_PT                     property,      // IN: the starting TPM property
482     UINT32                     count,         // IN: maximum number of returned
483                                               //     properties
484     TPML_TAGGED_TPM_PROPERTY   *propertyList  // OUT: property list
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
485         )
486     {
487         TPMI_YES_NO      more = NO;
488         UINT32           i;
489         UINT32           nextGroup;
490
491         // initialize output property list
492         propertyList->count = 0;
493
494         // maximum count of properties we may return is MAX_PCR_PROPERTIES
495         if(count > MAX_TPM_PROPERTIES) count = MAX_TPM_PROPERTIES;
496
497         // if property is less than PT_FIXED, start from PT_FIXED
498         if(property < PT_FIXED)
499             property = PT_FIXED;
500         // There is only the fixed and variable groups with the variable group coming
501         // last
502         if(property >= (PT_VAR + PT_GROUP))
503             return more;
504
505         // Don't read past the end of the selected group
506         nextGroup = ((property / PT_GROUP) * PT_GROUP) + PT_GROUP;
507
508         // Scan through the TPM properties of the requested group.
509         for(i = property; i < nextGroup; i++)
510         {
511             UINT32           value;
512             // if we have hit the end of the group, quit
513             if(i != property && ((i % PT_GROUP) == 0))
514                 break;
515             if(TPMPropertyIsDefined((TPM_PT)i, &value))
516             {
517                 if(propertyList->count < count)
518                 {
519                     // If the list is not full, add this property
520                     propertyList->tpmProperty[propertyList->count].property =
521                         (TPM_PT)i;
522                     propertyList->tpmProperty[propertyList->count].value = value;
523                     propertyList->count++;
524                 }
525                 else
526                 {
527                     // If the return list is full but there are more properties
528                     // available, set the indication and exit the loop.
529                     more = YES;
530                     break;
531                 }
532             }
533         }
534         return more;
535     }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

9.15 Response.c

9.15.1 Description

This file contains the common code for building a response header, including setting the size of the structure. *command* may be NULL if result is not TPM_RC_SUCCESS.

9.15.2 Includes and Defines

```
1   #include "Tpm.h"
```

9.15.3 BuildResponseHeader()

Adds the response header to the response. It will update command->*parameterSize* to indicate the total size of the response.

```
2   void
3   BuildResponseHeader(
4       COMMAND          *command,       // IN: main control structure
5       BYTE             *buffer,        // OUT: the output buffer
6       TPM_RC            result         // IN: the response code
7       )
8   {
9       TPM_ST              tag;
10      UINT32              size;
11
12      if(result != TPM_RC_SUCCESS)
13      {
14          tag = TPM_ST_NO_SESSIONS;
15          size = 10;
16      }
17      else
18      {
19          tag = command->tag;
20          // Compute the overall size of the response
21          size = STD_RESPONSE_HEADER + command->handleNum * sizeof(TPM_HANDLE);
22          size += command->parameterSize;
23          size += (command->tag == TPM_ST_SESSIONS) ?
24              command->authSize + sizeof(UINT32) : 0;
25      }
26      TPM_ST_Marshal(&tag, &buffer, NULL);
27      UINT32_Marshal(&size, &buffer, NULL);
28      TPM_RC_Marshal(&result, &buffer, NULL);
29      if(result == TPM_RC_SUCCESS)
30      {
31          if(command->handleNum > 0)
32              TPM_HANDLE_Marshal(&command->handles[0], &buffer, NULL);
33          if(tag == TPM_ST_SESSIONS)
34              UINT32_Marshal((UINT32 *)&command->parameterSize, &buffer, NULL);
35      }
36      command->parameterSize = size;
37  }
```

 Copyright © TCG 2006-2020

9.16 ResponseCodeProcessing.c

9.16.1 Description

This file contains the miscellaneous functions for processing response codes.

NOTE: Currently, there is only one.

9.16.2 Includes and Defines

```
1   #include "Tpm.h"
```

9.16.3 RcSafeAddToResult()

Adds a modifier to a response code as long as the response code allows a modifier and no modifier has already been added.

```
2   TPM_RC
3   RcSafeAddToResult(
4       TPM_RC          responseCode,
5       TPM_RC          modifier
6       )
7   {
8       if((responseCode & RC_FMT1) && !(responseCode & 0xf40))
9          return responseCode + modifier;
10      else
11          return responseCode;
12  }
```

 Copyright © TCG 2006-2020

9.17 TpmFail.c

9.17.1 Includes, Defines, and Types

```
1   #define      TPM_FAIL_C
2   #include     "Tpm.h"
3   #include     <assert.h>
```

On MS C compiler, can save the alignment state and set the alignment to 1 for the duration of the TpmTypes.h include. This will avoid a lot of alignment warnings from the compiler for the unaligned structures. The alignment of the structures is not important as this function does not use any of the structures in TpmTypes.h and only include it for the #defines of the capabilities, properties, and command code values.

```
4   #include "TpmTypes.h"
```

9.17.2 Typedefs

These defines are used primarily for sizing of the local response buffer.

```
5   typedef struct
6   {
7       TPM_ST          tag;
8       UINT32          size;
9       TPM_RC          code;
10  } HEADER;
11  typedef struct
12  {
13      BYTE            tag[sizeof(TPM_ST)];
14      BYTE            size[sizeof(UINT32)];
15      BYTE            code[sizeof(TPM_RC)];
16  } PACKED_HEADER;
17  typedef struct
18  {
19      BYTE             size[sizeof(UINT16)];
20      struct
21      {
22          BYTE         function[sizeof(UINT32)];
23          BYTE         line[sizeof(UINT32)];
24          BYTE         code[sizeof(UINT32)];
25      } values;
26      BYTE             returnCode[sizeof(TPM_RC)];
27  } GET_TEST_RESULT_PARAMETERS;
28  typedef struct
29  {
30      BYTE        moreData[sizeof(TPMI_YES_NO)];
31      BYTE        capability[sizeof(TPM_CAP)]; // Always TPM_CAP_TPM_PROPERTIES
32      BYTE        tpmProperty[sizeof(TPML_TAGGED_TPM_PROPERTY)];
33  } GET_CAPABILITY_PARAMETERS;
34  typedef struct
35  {
36      BYTE        header[sizeof(PACKED_HEADER)];
37      BYTE        getTestResult[sizeof(GET_TEST_RESULT_PARAMETERS)];
38  } TEST_RESPONSE;
39  typedef struct
40  {
41      BYTE        header[sizeof(PACKED_HEADER)];
42      BYTE        getCap[sizeof(GET_CAPABILITY_PARAMETERS)];
43  } CAPABILITY_RESPONSE;
44  typedef union
45  {
```

```
46      BYTE          test[sizeof(TEST_RESPONSE)];
47      BYTE          cap[sizeof(CAPABILITY_RESPONSE)];
48  } RESPONSES;
```

Buffer to hold the responses. This may be a little larger than required due to padding that a compiler might add.

NOTE: This is not in Global.c because of the specialized data definitions above. Since the data contained in this structure is not relevant outside of the execution of a single command (when the TPM is in failure mode. There is no compelling reason to move all the typedefs to Global.h and this structure to Global.c.

```
49  #ifndef __IGNORE_STATE__ // Don't define this value
50  static BYTE response[sizeof(RESPONSES)];
51  #endif
```

9.17.3 Local Functions

9.17.3.1 MarshalUint16()

Function to marshal a 16 bit value to the output buffer.

```
52  static INT32
53  MarshalUint16(
54      UINT16          integer,
55      BYTE            **buffer
56      )
57  {
58      UINT16_TO_BYTE_ARRAY(integer, *buffer);
59      *buffer += 2;
60      return 2;
61  }
```

9.17.3.2 MarshalUint32()

Function to marshal a 32 bit value to the output buffer.

```
62  static INT32
63  MarshalUint32(
64      UINT32           integer,
65      BYTE            **buffer
66      )
67  {
68      UINT32_TO_BYTE_ARRAY(integer, *buffer);
69      *buffer += 4;
70      return 4;
71  }
```

9.17.3.3 Unmarshal32()

```
72  static BOOL Unmarshal32(
73      UINT32          *target,
74      BYTE            **buffer,
75      INT32           *size
76      )
77  {
78      if((*size -= 4) < 0)
79          return FALSE;
80      *target = BYTE_ARRAY_TO_UINT32(*buffer);
81      *buffer += 4;
82      return TRUE;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
83  }
```

9.17.3.4 Unmarshal16()

```
84  static BOOL Unmarshal16(
85      UINT16          *target,
86      BYTE           **buffer,
87      INT32           *size
88  )
89  {
90      if((*size -= 2) < 0)
91          return FALSE;
92      *target = BYTE_ARRAY_TO_UINT16(*buffer);
93      *buffer += 2;
94      return TRUE;
95  }
```

9.17.4 Public Functions

9.17.4.1 SetForceFailureMode()

This function is called by the simulator to enable failure mode testing.

```
96  #if SIMULATION
97  LIB_EXPORT void
98  SetForceFailureMode(
99      void
100     )
101 {
102     g_forceFailureMode = TRUE;
103     return;
104 }
105 #endif
```

9.17.4.2 TpmLogFailure()

This function saves the failure values when the code will continue to operate. It if similar to TpmFail() but returns to the caller. The assumption is that the caller will propagate a failure back up the stack.

```
106 void
107 TpmLogFailure(
108 #if FAIL_TRACE
109     const char      *function,
110     int              line,
111 #endif
112     int              code
113 )
114 {
115     // Save the values that indicate where the error occurred.
116     // On a 64-bit machine, this may truncate the address of the string
117     // of the function name where the error occurred.
118 #if FAIL_TRACE
119     s_failFunction = (UINT32)(ptrdiff_t)function;
120     s_failLine = line;
121 #else
122     s_failFunction = 0;
123     s_failLine = 0;
124 #endif
125     s_failCode = code;
126
127     // We are in failure mode
```

```
128     g_inFailureMode = TRUE;
129
130     return;
131 }
```

9.17.4.3 TpmFail()

This function is called by TPM.lib when a failure occurs. It will set up the failure values to be returned on TPM2_GetTestResult().

```
132 NORETURN void
133 TpmFail(
134 #if FAIL_TRACE
135     const char       *function,
136     int               line,
137 #endif
138     int               code
139     )
140 {
141     // Save the values that indicate where the error occurred.
142     // On a 64-bit machine, this may truncate the address of the string
143     // of the function name where the error occurred.
144 #if FAIL_TRACE
145     s_failFunction = (UINT32)(ptrdiff_t)function;
146     s_failLine = line;
147 #else
148     s_failFunction = (UINT32)(ptrdiff_t)NULL;
149     s_failLine = 0;
150 #endif
151     s_failCode = code;
152
153     // We are in failure mode
154     g_inFailureMode = TRUE;
155
156     // if asserts are enabled, then do an assert unless the failure mode code
157     // is being tested.
158 #if SIMULATION
159 #   ifndef NDEBUG
160     assert(g_forceFailureMode);
161 #   endif
162     // Clear this flag
163     g_forceFailureMode = FALSE;
164 #endif
165     // Jump to the failure mode code.
166     // Note: only get here if asserts are off or if we are testing failure mode
167     _plat__Fail();
168 }
```

9.17.4.4 TpmFailureMode

This function is called by the interface code when the platform is in failure mode.

```
169 void
170 TpmFailureMode(
171     unsigned int      inRequestSize,     // IN: command buffer size
172     unsigned char    *inRequest,         // IN: command buffer
173     unsigned int     *outResponseSize,   // OUT: response buffer size
174     unsigned char   **outResponse        // OUT: response buffer
175     )
176 {
177     UINT32            marshalSize;
178     UINT32            capability;
179     HEADER            header;    // unmarshaled command header
```


 Copyright © TCG 2006-2020

```
180         UINT32              pt;     // unmarshaled property type
181         UINT32              count;  // unmarshaled property count
182         UINT8              *buffer = inRequest;
183         INT32               size = inRequestSize;
184
185         // If there is no command buffer, then just return TPM_RC_FAILURE
186         if(inRequestSize == 0 || inRequest == NULL)
187             goto FailureModeReturn;
188         // If the header is not correct for TPM2_GetCapability() or
189         // TPM2_GetTestResult() then just return the in failure mode response;
190         if(! (Unmarshal16(&header.tag,  &buffer, &size)
191            && Unmarshal32(&header.size, &buffer, &size)
192            && Unmarshal32(&header.code, &buffer, &size)))
193             goto FailureModeReturn;
194         if(header.tag != TPM_ST_NO_SESSIONS
195            || header.size < 10)
196             goto FailureModeReturn;
197         switch(header.code)
198         {
199             case TPM_CC_GetTestResult:
200                 // make sure that the command size is correct
201                 if(header.size != 10)
202                     goto FailureModeReturn;
203                 buffer = &response[10];
204                 marshalSize = MarshalUint16(3 * sizeof(UINT32), &buffer);
205                 marshalSize += MarshalUint32(s_failFunction, &buffer);
206                 marshalSize += MarshalUint32(s_failLine, &buffer);
207                 marshalSize += MarshalUint32(s_failCode, &buffer);
208                 if(s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
209                     marshalSize += MarshalUint32(TPM_RC_NV_UNINITIALIZED, &buffer);
210                 else
211                     marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
212                 break;
213             case TPM_CC_GetCapability:
214                 // make sure that the size of the command is exactly the size
215                 // returned for the capability, property, and count
216                 if(header.size != (10 + (3 * sizeof(UINT32)))
217                         // also verify that this is requesting TPM properties
218                    || !Unmarshal32(&capability, &buffer, &size)
219                    || capability != TPM_CAP_TPM_PROPERTIES
220                    || !Unmarshal32(&pt, &buffer, &size)
221                    || !Unmarshal32(&count, &buffer, &size))
222                     goto FailureModeReturn;
223                 // If in failure mode because of an unrecoverable read error, and the
224                 // property is 0 and the count is 0, then this is an indication to
225                 // re-manufacture the TPM. Do the re-manufacture but stay in failure
226                 // mode until the TPM is reset.
227                 // Note: this behavior is not required by the specification and it is
228                 // OK to leave the TPM permanently bricked due to an unrecoverable NV
229                 // error.
230                 if(count == 0 && pt == 0 && s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
231                 {
232                     g_manufactured = FALSE;
233                     TPM_Manufacture(0);
234                 }
235                 if(count > 0)
236                     count = 1;
237                 else if(pt > TPM_PT_FIRMWARE_VERSION_2)
238                     count = 0;
239                 if(pt < TPM_PT_MANUFACTURER)
240                     pt = TPM_PT_MANUFACTURER;
241                 // set up for return
242                 buffer = &response[10];
243                 // if the request was for a PT less than the last one
244                 // then we indicate more, otherwise, not.
245                 if(pt < TPM_PT_FIRMWARE_VERSION_2)
```

```
246                     *buffer++ = YES;
247                 else
248                     *buffer++ = NO;
249                 marshalSize = 1;
250
251                 // indicate the capability type
252                 marshalSize += MarshalUint32(capability, &buffer);
253                 // indicate the number of values that are being returned (0 or 1)
254                 marshalSize += MarshalUint32(count, &buffer);
255                 // indicate the property
256                 marshalSize += MarshalUint32(pt, &buffer);
257
258                 if(count > 0)
259                     switch(pt)
260                     {
261                         case TPM_PT_MANUFACTURER:
262                         // the vendor ID unique to each TPM manufacturer
263 #ifdef   MANUFACTURER
264                             pt = *(UINT32*)MANUFACTURER;
265 #else
266                             pt = 0;
267 #endif
268                             break;
269                         case TPM_PT_VENDOR_STRING_1:
270                             // the first four characters of the vendor ID string
271 #ifdef   VENDOR_STRING_1
272                             pt = *(UINT32*)VENDOR_STRING_1;
273 #else
274                             pt = 0;
275 #endif
276                             break;
277                         case TPM_PT_VENDOR_STRING_2:
278                             // the second four characters of the vendor ID string
279 #ifdef   VENDOR_STRING_2
280                             pt = *(UINT32*)VENDOR_STRING_2;
281 #else
282                             pt = 0;
283 #endif
284                             break;
285                         case TPM_PT_VENDOR_STRING_3:
286                             // the third four characters of the vendor ID string
287 #ifdef   VENDOR_STRING_3
288                             pt = *(UINT32*)VENDOR_STRING_3;
289 #else
290                             pt = 0;
291 #endif
292                             break;
293                         case TPM_PT_VENDOR_STRING_4:
294                             // the fourth four characters of the vendor ID string
295 #ifdef   VENDOR_STRING_4
296                             pt = *(UINT32*)VENDOR_STRING_4;
297 #else
298                             pt = 0;
299 #endif
300                             break;
301                         case TPM_PT_VENDOR_TPM_TYPE:
302                             // vendor-defined value indicating the TPM model
303                             // We just make up a number here
304                             pt = 1;
305                             break;
306                         case TPM_PT_FIRMWARE_VERSION_1:
307                             // the more significant 32-bits of a vendor-specific value
308                             // indicating the version of the firmware
309 #ifdef   FIRMWARE_V1
310                             pt = FIRMWARE_V1;
311 #else
```

```
312                             pt = 0;
313     #endif
314                             break;
315                         default: // TPM_PT_FIRMWARE_VERSION_2:
316                             // the less significant 32-bits of a vendor-specific value
317                             // indicating the version of the firmware
318     #ifdef  FIRMWARE_V2
319                             pt = FIRMWARE_V2;
320     #else
321                             pt = 0;
322     #endif
323                             break;
324                     }
325                 marshalSize += MarshalUint32(pt, &buffer);
326                 break;
327             default: // default for switch (cc)
328                 goto FailureModeReturn;
329         }
330         // Now do the header
331         buffer = response;
332         marshalSize = marshalSize + 10; // Add the header size to the
333                                         // stuff already marshaled
334         MarshalUint16(TPM_ST_NO_SESSIONS, &buffer); // structure tag
335         MarshalUint32(marshalSize, &buffer);  // responseSize
336         MarshalUint32(TPM_RC_SUCCESS, &buffer); // response code
337
338         *outResponseSize = marshalSize;
339         *outResponse = (unsigned char *)&response;
340         return;
341     FailureModeReturn:
342         buffer = response;
343         marshalSize = MarshalUint16(TPM_ST_NO_SESSIONS, &buffer);
344         marshalSize += MarshalUint32(10, &buffer);
345         marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
346         *outResponseSize = marshalSize;
347         *outResponse = (unsigned char *)response;
348         return;
349     }
```

9.17.4.5 UnmarshalFail()

This is a stub that is used to catch an attempt to unmarshal an entry that is not defined. Don't ever expect this to be called but...

```
350     void
351     UnmarshalFail(
352         void                *type,
353         BYTE                **buffer,
354         INT32               *size
355         )
356     {
357         NOT_REFERENCED(type);
358         NOT_REFERENCED(buffer);
359         NOT_REFERENCED(size);
360         FAIL(FATAL_ERROR_INTERNAL);
361     }
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

10 Cryptographic Functions

10.1 Headers

10.1.1 BnValues.h

10.1.1.1 Introduction

This file contains the definitions needed for defining the internal BIGNUM structure. A BIGNUM is a pointer to a structure. The structure has three fields. The last field is and array (d) of crypt_uword_t. Each word is in machine format (big- or little-endian) with the words in ascending significance (i.e. words in little-endian order). This is the order that seems to be used in every big number library in the worlds, so...

The first field in the structure (allocated) is the number of words in *d*. This is the upper limit on the size of the number that can be held in the structure. This differs from libraries like OpenSSL as this is not intended to deal with numbers of arbitrary size; just numbers that are needed to deal with the algorithms that are defined in the TPM implementation.

The second field in the structure (size) is the number of significant words in *n*. When this number is zero, the number is zero. The word at used-1 should never be zero. All words between d[size] and d[allocated-1] should be zero.

10.1.1.2 Defines

```
1   #ifndef _BN_NUMBERS_H
2   #define _BN_NUMBERS_H
3   #if RADIX_BITS == 64
4   # define RADIX_LOG2         6
5   #elif RADIX_BITS == 32
6   #define RADIX_LOG2          5
7   #else
8   # error "Unsupported radix"
9   #endif
10  #define RADIX_MOD(x)        ((x) & ((1 << RADIX_LOG2) - 1))
11  #define RADIX_DIV(x)        ((x) >> RADIX_LOG2)
12  #define RADIX_MASK  ((((crypt_uword_t)1) << RADIX_LOG2) - 1)
13  #define BITS_TO_CRYPT_WORDS(bits)       RADIX_DIV((bits) + (RADIX_BITS - 1))
14  #define BYTES_TO_CRYPT_WORDS(bytes)     BITS_TO_CRYPT_WORDS(bytes * 8)
15  #define SIZE_IN_CRYPT_WORDS(thing)      BYTES_TO_CRYPT_WORDS(sizeof(thing))
16  #if RADIX_BITS == 64
17  #define SWAP_CRYPT_WORD(x)  REVERSE_ENDIAN_64(x)
18      typedef uint64_t    crypt_uword_t;
19      typedef int64_t     crypt_word_t;
20  #   define TO_CRYPT_WORD_64             BIG_ENDIAN_BYTES_TO_UINT64
21  #   define TO_CRYPT_WORD_32(a, b, c, d) TO_CRYPT_WORD_64(0, 0, 0, 0, a, b, c, d)
22  #elif RADIX_BITS == 32
23  #   define SWAP_CRYPT_WORD(x)  REVERSE_ENDIAN_32((x))
24      typedef uint32_t    crypt_uword_t;
25      typedef int32_t     crypt_word_t;
26  #   define TO_CRYPT_WORD_64(a, b, c, d, e, f, g, h)                        \
27          BIG_ENDIAN_BYTES_TO_UINT32(e, f, g, h),                            \
28          BIG_ENDIAN_BYTES_TO_UINT32(a, b, c, d)
29  #endif
30  #define MAX_CRYPT_UWORD (~((crypt_uword_t)0))
31  #define MAX_CRYPT_WORD  ((crypt_word_t)(MAX_CRYPT_UWORD >> 1))
32  #define MIN_CRYPT_WORD  (~MAX_CRYPT_WORD)
33  #define LARGEST_NUMBER (MAX((ALG_RSA * MAX_RSA_KEY_BYTES),                 \
34                          MAX((ALG_ECC * MAX_ECC_KEY_BYTES), MAX_DIGEST_SIZE)))
35  #define LARGEST_NUMBER_BITS (LARGEST_NUMBER * 8)
36  #define MAX_ECC_PARAMETER_BYTES (MAX_ECC_KEY_BYTES * ALG_ECC)
```

 Copyright © TCG 2006-2020

These are the basic big number formats. This is convertible to the library- specific format without to much difficulty. For the math performed using these numbers, the value is always positive.

```
37  #define BN_STRUCT_DEF(count) struct {          \
38      crypt_uword_t       allocated;             \
39      crypt_uword_t       size;                  \
40      crypt_uword_t       d[count];              \
41      }
42  typedef BN_STRUCT_DEF(1) bignum_t;
43  #ifndef bigNum
44  typedef bignum_t       *bigNum;
45  typedef const bignum_t *bigConst;
46  #endif
47  extern const bignum_t   BnConstZero;
48
49  // The Functions to access the properties of a big number.
50  // Get number of allocated words
51  #define BnGetAllocated(x)    (unsigned)((x)->allocated)
```

Get number of words used

```
52  #define BnGetSize(x)         ((x)->size)
```

Get a pointer to the data array

```
53  #define BnGetArray(x)        ((crypt_uword_t *)&((x)->d[0]))
```

Get the nth word of a BIGNUM (zero-based)

```
54  #define BnGetWord(x, i)      (crypt_uword_t)((x)->d[i])
```

Some things that are done often. Test to see if a bignum_t is equal to zero

```
55  #define BnEqualZero(bn)    (BnGetSize(bn) == 0)
```

Test to see if a bignum_t is equal to a word type

```
56  #define BnEqualWord(bn, word)                                                  \
57              ((BnGetSize(bn) == 1) && (BnGetWord(bn, 0) == (crypt_uword_t)word))
```

Determine if a BIGNUM is even. A zero is even. Although the indication that a number is zero is that it's size is zero, all words of the number are 0 so this test works on zero.

```
58  #define BnIsEven(n)      ((BnGetWord(n, 0) & 1) == 0)
```

The macros below are used to define BIGNUM values of the required size. The values are allocated on the stack so they can be treated like simple local values. This will call the initialization function for a defined bignum_t. This sets the allocated and used fields and clears the words of *n*.

```
59  #define BN_INIT(name)                                                          \
60      (bigNum)BnInit((bigNum)&(name),                                            \
61                  BYTES_TO_CRYPT_WORDS(sizeof(name.d)))
```

In some cases, a function will need the address of the structure associated with a variable. The structure for a BIGNUM variable of *name* is *name_*. Generally, when the structure is created, it is initialized and a parameter is created with a pointer to the structure. The pointer has the *name* and the structure it points to is *name_*

```
62  #define BN_ADDRESS(name)  (bigNum)&name##_
63  #define BN_STRUCT_ALLOCATION(bits)  (BITS_TO_CRYPT_WORDS(bits) + 1)
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

Create a structure of the correct size.

```
64  #define BN_STRUCT(bits)                                                  \
65      BN_STRUCT_DEF(BN_STRUCT_ALLOCATION(bits))
```

Define a BIGNUM type with a specific allocation

```
66  #define BN_TYPE(name, bits)                                              \
67      typedef BN_STRUCT(bits) bn_##name##_t
```

This creates a local BIGNUM variable of a specific size and initializes it from a TPM2B input parameter.

```
68  #define BN_INITIALIZED(name, bits, initializer)                          \
69      BN_STRUCT(bits)   name##_;                                           \
70      bigNum            name = BnFrom2B(BN_INIT(name##_),                  \
71                                   (const TPM2B *)initializer)
```

Create a local variable that can hold a number with *bits*

```
72  #define BN_VAR(name, bits)                                               \
73      BN_STRUCT(bits)   _##name;                                           \
74      bigNum            name = BN_INIT(_##name)
```

Create a type that can hold the largest number defined by the implementation.

```
75  #define BN_MAX(name)    BN_VAR(name, LARGEST_NUMBER_BITS)
76  #define BN_MAX_INITIALIZED(name, initializer)                            \
77      BN_INITIALIZED(name, LARGEST_NUMBER_BITS, initializer)
```

A word size value is useful

```
78  #define BN_WORD(name)       BN_VAR(name, RADIX_BITS)
```

This is used to created a word-size BIGNUM and initialize it with an input parameter to a function.

```
79  #define BN_WORD_INITIALIZED(name, initial)                               \
80      BN_STRUCT(RADIX_BITS)   name##_;                                     \
81      bigNum                  name = BnInitializeWord((bigNum)&name##_,    \
82                                   BN_STRUCT_ALLOCATION(RADIX_BITS), initial)
```

ECC-Specific Values This is the format for a point. It is always in affine format. The Z value is carried as part of the point, primarily to simplify the interface to the support library. Rather than have the interface layer have to create space for the point each time it is used... The x, y, and z values are pointers to *bigNum* values and not in-line versions of the numbers. This is a relic of the days when there was no standard TPM format for the numbers

```
83  typedef struct _bn_point_t
84  {
85      bigNum          x;
86      bigNum          y;
87      bigNum          z;
88  } bn_point_t;
89  typedef bn_point_t          *bigPoint;
90  typedef const bn_point_t    *pointConst;
91  typedef struct constant_point_t
92  {
93      bigConst        x;
94      bigConst        y;
95      bigConst        z;
96  } constant_point_t;
97  #define ECC_BITS    (MAX_ECC_KEY_BYTES * 8)
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
98   BN_TYPE(ecc, ECC_BITS);
99   #define ECC_NUM(name)        BN_VAR(name, ECC_BITS)
100  #define ECC_INITIALIZED(name, initializer)                              \
101      BN_INITIALIZED(name, ECC_BITS, initializer)
102  #define POINT_INSTANCE(name, bits)                                      \
103      BN_STRUCT (bits)    name##_x =                                      \
104                  {BITS_TO_CRYPT_WORDS ( bits ), 0,{0}};                  \
105      BN_STRUCT ( bits )    name##_y =                                    \
106                  {BITS_TO_CRYPT_WORDS ( bits ), 0,{0}};                  \
107      BN_STRUCT ( bits )    name##_z =                                    \
108                  {BITS_TO_CRYPT_WORDS ( bits ), 0,{0}};                  \
109      bn_point_t name##_
110  #define POINT_INITIALIZER(name)                                         \
111      BnInitializePoint(&name##_, (bigNum)&name##_x,                      \
112                      (bigNum)&name##_y, (bigNum)&name##_z)
113  #define POINT_INITIALIZED(name, initValue)                              \
114      POINT_INSTANCE(name, MAX_ECC_KEY_BITS);                             \
115      bigPoint             name = BnPointFrom2B(                          \
116                                      POINT_INITIALIZER(name),            \
117                                      initValue)
118  #define POINT_VAR(name, bits)                                           \
119      POINT_INSTANCE (name, bits);                                        \
120      bigPoint            name = POINT_INITIALIZER(name)
121  #define POINT(name)        POINT_VAR(name, MAX_ECC_KEY_BITS)
```

Structure for the curve parameters. This is an analog to the TPMS_ALGORITHM_DETAIL_ECC

```
122  typedef struct
123  {
124      bigConst             prime;     // a prime number
125      bigConst             order;     // the order of the curve
126      bigConst             h;         // cofactor
127      bigConst             a;         // linear coefficient
128      bigConst             b;         // constant term
129      constant_point_t     base;      // base point
130  } ECC_CURVE_DATA;
```

Access macros for the ECC_CURVE structure. The parameter *C* is a pointer to an ECC_CURVE_DATA structure. In some libraries, the curve structure contains a pointer to an ECC_CURVE_DATA structure as well as some other bits. For those cases, the AccessCurveData() macro is used in the code to first get the pointer to the ECC_CURVE_DATA for access. In some cases, the macro does noting.

```
131  #define CurveGetPrime(C)     ((C)->prime)
132  #define CurveGetOrder(C)     ((C)->order)
133  #define CurveGetCofactor(C)  ((C)->h)
134  #define CurveGet_a(C)        ((C)->a)
135  #define CurveGet_b(C)        ((C)->b)
136  #define CurveGetG(C)         ((pointConst)&((C)->base))
137  #define CurveGetGx(C)        ((C)->base.x)
138  #define CurveGetGy(C)        ((C)->base.y)
```

Convert bytes in initializers according to the endianess of the system. This is used for CryptEccData.c.

```
139  #define     BIG_ENDIAN_BYTES_TO_UINT32(a, b, c, d)                      \
140              (    ((UINT32)(a) << 24)                                    \
141              +    ((UINT32)(b) << 16)                                    \
142              +    ((UINT32)(c) << 8)                                     \
143              +    ((UINT32)(d))                                          \
144              )
145  #define     BIG_ENDIAN_BYTES_TO_UINT64(a, b, c, d, e, f, g, h)          \
146              (    ((UINT64)(a) << 56)                                    \
147              +    ((UINT64)(b) << 48)                                    \
148              +    ((UINT64)(c) << 40)                                    \
149              +    ((UINT64)(d) << 32)                                    \
```

 Copyright © TCG 2006-2020

```
150             +    ((UINT64)(e) << 24)                                    \
151             +    ((UINT64)(f) << 16)                                    \
152             +    ((UINT64)(g) << 8)                                     \
153             +    ((UINT64)(h))                                          \
154             )
155 #ifndef RADIX_BYTES
156 #   if RADIX_BITS == 32
157 #       define RADIX_BYTES 4
158 #   elif RADIX_BITS == 64
159 #       define RADIX_BYTES 8
160 #   else
161 #       error "RADIX_BITS must either be 32 or 64"
162 #   endif
163 #endif
```

Add implementation dependent definitions for other ECC Values and for linkages.

```
164 #include LIB_INCLUDE(MATH_LIB, Math)
165 #endif // _BN_NUMBERS_H
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.1.2 CryptEcc.h

10.1.2.1 Introduction

This file contains structure definitions used for ECC. The structures in this file are only used internally. The ECC-related structures that cross the TPM interface are defined in TpmTypes.h

```
1   #ifndef _CRYPT_ECC_H
2   #define _CRYPT_ECC_H
```

10.1.2.2 Structures

This is used to define the macro that may or may not be in the data set for the curve (CryptEccData.c). If there is a mismatch, the compiler will warn that there is to much/not enough initialization data in the curve. The macro is used because not all versions of the CryptEccData.c need the curve name.

```
3   #ifdef NAMED_CURVES
4   #define CURVE_NAME(a) , a
5   #define CURVE_NAME_DEF const char *name;
6   #else
7   #  define CURVE_NAME(a)
8   #  define CURVE_NAME_DEF
9   #endif
10  typedef struct ECC_CURVE
11  {
12      const TPM_ECC_CURVE          curveId;
13      const UINT16                 keySizeBits;
14      const TPMT_KDF_SCHEME        kdf;
15      const TPMT_ECC_SCHEME        sign;
16      const ECC_CURVE_DATA        *curveData; // the address of the curve data
17      const BYTE                  *OID;
18      CURVE_NAME_DEF
19  } ECC_CURVE;
20  extern const ECC_CURVE eccCurves[ECC_CURVE_COUNT];
21
22  #endif
```

10.1.3 CryptHash.h

10.1.3.1 Introduction

This header contains the hash structure definitions used in the TPM code to define the amount of space to be reserved for the hash state. This allows the TPM code to not have to import all of the symbols used by the hash computations. This lets the build environment of the TPM code not to have include the header files associated with the CryptoEngine() code.

```
1    #ifndef _CRYPT_HASH_H
2    #define _CRYPT_HASH_H
```

10.1.3.2 Hash-related Structures

```
3    union SMAC_STATES;
```

These definitions add the high-level methods for processing state that may be an SMAC

```
4    typedef void(* SMAC_DATA_METHOD)(
5        union SMAC_STATES       *state,
6        UINT32                   size,
7        const BYTE              *buffer
8        );
9    typedef UINT16(* SMAC_END_METHOD)(
10       union SMAC_STATES       *state,
11       UINT32                   size,
12       BYTE                    *buffer
13       );
14   typedef struct sequenceMethods {
15       SMAC_DATA_METHOD           data;
16       SMAC_END_METHOD            end;
17   } SMAC_METHODS;
18   #define SMAC_IMPLEMENTED (CC_MAC || CC_MAC_Start)
```

These definitions are here because the SMAC state is in the union of hash states.

```
19   typedef struct tpmCmacState {
20       TPM_ALG_ID              symAlg;
21       UINT16                  keySizeBits;
22       INT16                   bcount; // current count of bytes accumulated in IV
23       TPM2B_IV                iv;     // IV buffer
24       TPM2B_SYM_KEY           symKey;
25   } tpmCmacState_t;
26   typedef union SMAC_STATES {
27   #if ALG_CMAC
28       tpmCmacState_t          cmac;
29   #endif
30       UINT64                  pad;
31   } SMAC_STATES;
32   typedef struct SMAC_STATE {
33       SMAC_METHODS            smacMethods;
34       SMAC_STATES             state;
35   } SMAC_STATE;
36   typedef union
37   {
38   #if ALG_SHA1
39       tpmHashStateSHA1_t         Sha1;
40   #endif
41   #if ALG_SHA256
42       tpmHashStateSHA256_t       Sha256;
43   #endif
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
44   #if ALG_SHA384
45       tpmHashStateSHA384_t        Sha384;
46   #endif
47   #if ALG_SHA512
48       tpmHashStateSHA512_t        Sha512;
49   #endif
50   #if ALG_SM3_256
51       tpmHashStateSM3_256_t       Sm3_256;
52   #endif
53
54   // Additions for symmetric block cipher MAC
55   #if SMAC_IMPLEMENTED
56       SMAC_STATE                  smac;
57   #endif
58       // to force structure alignment to be no worse than HASH_ALIGNMENT
59   #if HASH_ALIGNMENT == 4
60       uint32_t              align;
61   #else
62       uint64_t              align;
63   #endif
64   } ANY_HASH_STATE;
65   typedef ANY_HASH_STATE *PANY_HASH_STATE;
66   typedef const ANY_HASH_STATE    *PCANY_HASH_STATE;
67   #define ALIGNED_SIZE(x, b) (((x) + (b) - 1) / (b)) * (b))
```

MAX_HASH_STATE_SIZE will change with each implementation. It is assumed that a hash state will not be larger than twice the block size plus some overhead (in this case, 16 bytes). The overall size needs to be as large as any of the hash contexts. The structure needs to start on an alignment boundary and be an even multiple of the alignment

```
68   #define MAX_HASH_STATE_SIZE ((2 * MAX_HASH_BLOCK_SIZE) + 16)
69   #define MAX_HASH_STATE_SIZE_ALIGNED                                    \
70                   ALIGNED_SIZE(MAX_HASH_STATE_SIZE, HASH_ALIGNMENT)
```

This is an aligned byte array that will hold any of the hash contexts.

```
71   typedef  ANY_HASH_STATE ALIGNED_HASH_STATE;
```

The header associated with the hash library is expected to define the methods which include the calling sequence. When not compiling CryptHash.c, the methods are not defined so we need placeholder functions for the structures

```
72   #ifndef HASH_START_METHOD_DEF
73   #   define HASH_START_METHOD_DEF    void (HASH_START_METHOD)(void)
74   #endif
75   #ifndef HASH_DATA_METHOD_DEF
76   #   define HASH_DATA_METHOD_DEF     void (HASH_DATA_METHOD)(void)
77   #endif
78   #ifndef HASH_END_METHOD_DEF
79   #   define HASH_END_METHOD_DEF      void (HASH_END_METHOD)(void)
80   #endif
81   #ifndef HASH_STATE_COPY_METHOD_DEF
82   #   define HASH_STATE_COPY_METHOD_DEF     void (HASH_STATE_COPY_METHOD)(void)
83   #endif
84   #ifndef  HASH_STATE_EXPORT_METHOD_DEF
85   #   define  HASH_STATE_EXPORT_METHOD_DEF   void (HASH_STATE_EXPORT_METHOD)(void)
86   #endif
87   #ifndef  HASH_STATE_IMPORT_METHOD_DEF
88   #   define  HASH_STATE_IMPORT_METHOD_DEF   void (HASH_STATE_IMPORT_METHOD)(void)
89   #endif
```

Define the prototypical function call for each of the methods. This defines the order in which the parameters are passed to the underlying function.

 Copyright © TCG 2006-2020

```
 90     typedef HASH_START_METHOD_DEF;
 91     typedef HASH_DATA_METHOD_DEF;
 92     typedef HASH_END_METHOD_DEF;
 93     typedef HASH_STATE_COPY_METHOD_DEF;
 94     typedef HASH_STATE_EXPORT_METHOD_DEF;
 95     typedef HASH_STATE_IMPORT_METHOD_DEF;
 96     typedef struct _HASH_METHODS
 97     {
 98         HASH_START_METHOD           *start;
 99         HASH_DATA_METHOD            *data;
100         HASH_END_METHOD             *end;
101         HASH_STATE_COPY_METHOD      *copy;       // Copy a hash block
102         HASH_STATE_EXPORT_METHOD    *copyOut;    // Copy a hash block from a hash
103                                                  // context
104         HASH_STATE_IMPORT_METHOD    *copyIn;     // Copy a hash block to a proper hash
105                                                  // context
106     } HASH_METHODS, *PHASH_METHODS;
107     #if ALG_SHA1
108         TPM2B_TYPE(SHA1_DIGEST, SHA1_DIGEST_SIZE);
109     #endif
110     #if ALG_SHA256
111         TPM2B_TYPE(SHA256_DIGEST, SHA256_DIGEST_SIZE);
112     #endif
113     #if ALG_SHA384
114         TPM2B_TYPE(SHA384_DIGEST, SHA384_DIGEST_SIZE);
115     #endif
116     #if ALG_SHA512
117         TPM2B_TYPE(SHA512_DIGEST, SHA512_DIGEST_SIZE);
118     #endif
119     #if ALG_SM3_256
120         TPM2B_TYPE(SM3_256_DIGEST, SM3_256_DIGEST_SIZE);
121     #endif
```

When the TPM implements RSA, the hash-dependent OID pointers are part of the HASH_DEF. These macros conditionally add the OID reference to the HASH_DEF and the HASH_DEF_TEMPLATE.

```
122     #if ALG_RSA
123     #define PKCS1_HASH_REF    const BYTE   *PKCS1;
124     #define PKCS1_OID(NAME)   , OID_PKCS1_##NAME
125     #else
126     #define PKCS1_HASH_REF
127     #define PKCS1_OID(NAME)
128     #endif
```

When the TPM implements ECC, the hash-dependent OID pointers are part of the HASH_DEF. These macros conditionally add the OID reference to the HASH_DEF and the HASH_DEF_TEMPLATE.

```
129     #if ALG_ECDSA
130     #define ECDSA_HASH_REF     const BYTE   *ECDSA;
131     #define ECDSA_OID(NAME)   , OID_ECDSA_##NAME
132     #else
133     #define ECDSA_HASH_REF
134     #define ECDSA_OID(NAME)
135     #endif
136     typedef const struct HASH_DEF
137     {
138         HASH_METHODS            method;
139         uint16_t                blockSize;
140         uint16_t                digestSize;
141         uint16_t                contextSize;
142         uint16_t                hashAlg;
143         const BYTE             *OID;
144         PKCS1_HASH_REF         // PKCS1 OID
145         ECDSA_HASH_REF         // ECDSA OID
```

```
146     } HASH_DEF, *PHASH_DEF;
```

Macro to fill in the HASH_DEF for an algorithm. For SHA1, the instance would be: HASH_DEF_TEMPLATE(Sha1, SHA1) This handles the difference in capitalization for the various pieces.

```
147     #define HASH_DEF_TEMPLATE(HASH, Hash)                                          \
148         HASH_DEF    Hash##_Def= {                                               \
149                         {(HASH_START_METHOD *)&tpmHashStart_##HASH,             \
150                          (HASH_DATA_METHOD *)&tpmHashData_##HASH,               \
151                          (HASH_END_METHOD *)&tpmHashEnd_##HASH,                 \
152                          (HASH_STATE_COPY_METHOD *)&tpmHashStateCopy_##HASH,    \
153                          (HASH_STATE_EXPORT_METHOD *)&tpmHashStateExport_##HASH, \
154                          (HASH_STATE_IMPORT_METHOD *)&tpmHashStateImport_##HASH, \
155                         },                                                      \
156                         HASH##_BLOCK_SIZE,     /*block size */                  \
157                         HASH##_DIGEST_SIZE,    /*data size */                   \
158                         sizeof(tpmHashState##HASH##_t),                         \
159                         TPM_ALG_##HASH, OID_##HASH                              \
160                         PKCS1_OID(HASH) ECDSA_OID(HASH)};
```

These definitions are for the types that can be in a hash state structure. These types are used in the cryptographic utilities. This is a define rather than an enum so that the size of this field can be explicit.

```
161     typedef BYTE    HASH_STATE_TYPE;
162     #define HASH_STATE_EMPTY        ((HASH_STATE_TYPE) 0)
163     #define HASH_STATE_HASH         ((HASH_STATE_TYPE) 1)
164     #define HASH_STATE_HMAC         ((HASH_STATE_TYPE) 2)
165     #if CC_MAC || CC_MAC_Start
166     #define HASH_STATE_SMAC         ((HASH_STATE_TYPE) 3)
167     #endif
```

This is the structure that is used for passing a context into the hashing functions. It should be the same size as the function context used within the hashing functions. This is checked when the hash function is initialized. This version uses a new layout for the contexts and a different definition. The state buffer is an array of HASH_UNIT values so that a decent compiler will put the structure on a HASH_UNIT boundary. If the structure is not properly aligned, the code that manipulates the structure will copy to a properly aligned structure before it is used and copy the result back. This just makes things slower.

NOTE: This version of the state had the pointer to the update method in the state. This is to allow the SMAC functions to use the same structure without having to replicate the entire HASH_DEF structure.

```
168     typedef struct _HASH_STATE
169     {
170         HASH_STATE_TYPE          type;               // type of the context
171         TPM_ALG_ID               hashAlg;
172         PHASH_DEF                def;
173         ANY_HASH_STATE           state;
174     } HASH_STATE, *PHASH_STATE;
175     typedef const HASH_STATE *PCHASH_STATE;
```

10.1.3.3 HMAC State Structures

An HMAC_STATE structure contains an opaque HMAC stack state. A caller would use this structure when performing incremental HMAC operations. This structure contains a hash state and an HMAC key and allows slightly better stack optimization than adding an HMAC key to each hash state.

```
176     typedef struct hmacState
177     {
178         HASH_STATE           hashState;          // the hash state
179         TPM2B_HASH_BLOCK     hmacKey;            // the HMAC key
180     } HMAC_STATE, *PHMAC_STATE;
```

This is for the external hash state. This implementation assumes that the size of the exported hash state is no larger than the internal hash state.

```
181 typedef struct
182 {
183     BYTE                     buffer[sizeof(HASH_STATE)];
184 } EXPORT_HASH_STATE, *PEXPORT_HASH_STATE;
185 typedef const EXPORT_HASH_STATE *PCEXPORT_HASH_STATE;
186 #endif //  _CRYPT_HASH_H
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.1.4 CryptRand.h

10.1.4.1 Introduction

This file contains constant definition shared by CryptUtil() and the parts of the Crypto Engine.

```
1   #ifndef _CRYPT_RAND_H
2   #define _CRYPT_RAND_H
```

10.1.4.2 DRBG Structures and Defines

Values and structures for the random number generator. These values are defined in this header file so that the size of the RNG state can be known to TPM.lib. This allows the allocation of some space in NV memory for the state to be stored on an orderly shutdown. The DRBG based on a symmetric block cipher is defined by three values,

a) the key size

b) the block size (the IV size)

c) the symmetric algorithm

```
3   #define DRBG_KEY_SIZE_BITS       AES_MAX_KEY_SIZE_BITS
4   #define DRBG_IV_SIZE_BITS        (AES_MAX_BLOCK_SIZE * 8)
5   #define DRBG_ALGORITHM           TPM_ALG_AES
6   typedef tpmKeyScheduleAES      DRBG_KEY_SCHEDULE;
7   #define DRBG_ENCRYPT_SETUP(key, keySizeInBits, schedule)        \
8               TpmCryptSetEncryptKeyAES(key, keySizeInBits, schedule)
9   #define DRBG_ENCRYPT(keySchedule, in, out)                      \
10              TpmCryptEncryptAES(SWIZZLE(keySchedule, in, out))
11  #if     ((DRBG_KEY_SIZE_BITS % RADIX_BITS) != 0) \
12      || ((DRBG_IV_SIZE_BITS % RADIX_BITS) != 0)
13  #error "Key size and IV for DRBG must be even multiples of the radix"
14  #endif
15  #if (DRBG_KEY_SIZE_BITS % DRBG_IV_SIZE_BITS) != 0
16  #error "Key size for DRBG must be even multiple of the cypher block size"
17  #endif
```

Derived values

```
18  #define DRBG_MAX_REQUESTS_PER_RESEED (1 << 48)
19  #define DRBG_MAX_REQEST_SIZE (1 << 32)
20  #define pDRBG_KEY(seed)     ((DRBG_KEY *)&(((BYTE *)(seed))[0]))
21  #define pDRBG_IV(seed)      ((DRBG_IV *)&(((BYTE *)(seed))[DRBG_KEY_SIZE_BYTES]))
22  #define DRBG_KEY_SIZE_WORDS      (BITS_TO_CRYPT_WORDS(DRBG_KEY_SIZE_BITS))
23  #define DRBG_KEY_SIZE_BYTES      (DRBG_KEY_SIZE_WORDS * RADIX_BYTES)
24  #define DRBG_IV_SIZE_WORDS       (BITS_TO_CRYPT_WORDS(DRBG_IV_SIZE_BITS))
25  #define DRBG_IV_SIZE_BYTES       (DRBG_IV_SIZE_WORDS * RADIX_BYTES)
26  #define DRBG_SEED_SIZE_WORDS     (DRBG_KEY_SIZE_WORDS + DRBG_IV_SIZE_WORDS)
27  #define DRBG_SEED_SIZE_BYTES     (DRBG_KEY_SIZE_BYTES + DRBG_IV_SIZE_BYTES)
28  typedef union
29  {
30      BYTE            bytes[DRBG_KEY_SIZE_BYTES];
31      crypt_uword_t   words[DRBG_KEY_SIZE_WORDS];
32  } DRBG_KEY;
33  typedef union
34  {
35      BYTE            bytes[DRBG_IV_SIZE_BYTES];
36      crypt_uword_t   words[DRBG_IV_SIZE_WORDS];
37  } DRBG_IV;
38  typedef union
39  {
```

```
40      BYTE            bytes[DRBG_SEED_SIZE_BYTES];
41      crypt_uword_t   words[DRBG_SEED_SIZE_WORDS];
42  } DRBG_SEED;
43  #define CTR_DRBG_MAX_REQUESTS_PER_RESEED        ((UINT64)1 << 20)
44  #define CTR_DRBG_MAX_BYTES_PER_REQUEST          (1 << 16)
45  #   define CTR_DRBG_MIN_ENTROPY_INPUT_LENGTH    DRBG_SEED_SIZE_BYTES
46  #   define CTR_DRBG_MAX_ENTROPY_INPUT_LENGTH    DRBG_SEED_SIZE_BYTES
47  #   define CTR_DRBG_MAX_ADDITIONAL_INPUT_LENGTH DRBG_SEED_SIZE_BYTES
48  #define     TESTING         (1 << 0)
49  #define     ENTROPY         (1 << 1)
50  #define     TESTED          (1 << 2)
51  #define     IsTestStateSet(BIT)    ((g_cryptoSelfTestState.rng & BIT) != 0)
52  #define     SetTestStateBit(BIT)   (g_cryptoSelfTestState.rng |= BIT)
53  #define     ClearTestStateBit(BIT) (g_cryptoSelfTestState.rng &= ~BIT)
54  #define     IsSelfTest()    IsTestStateSet(TESTING)
55  #define     SetSelfTest()   SetTestStateBit(TESTING)
56  #define     ClearSelfTest() ClearTestStateBit(TESTING)
57  #define     IsEntropyBad()      IsTestStateSet(ENTROPY)
58  #define     SetEntropyBad()     SetTestStateBit(ENTROPY)
59  #define     ClearEntropyBad()   ClearTestStateBit(ENTROPY)
60  #define     IsDrbgTested()      IsTestStateSet(TESTED)
61  #define     SetDrbgTested()     SetTestStateBit(TESTED)
62  #define     ClearDrbgTested()   ClearTestStateBit(TESTED)
63  typedef struct
64  {
65      UINT64      reseedCounter;
66      UINT32      magic;
67      DRBG_SEED   seed; // contains the key and IV for the counter mode DRBG
68      UINT32      lastValue[4];   // used when the TPM does continuous self-test
69                                  // for FIPS compliance of DRBG
70  } DRBG_STATE, *pDRBG_STATE;
71  #define DRBG_MAGIC   ((UINT32) 0x47425244) // "DRBG" backwards so that it displays
72  typedef struct
73  {
74      UINT64                counter;
75      UINT32                magic;
76      UINT32                limit;
77      TPM2B                *seed;
78      const TPM2B          *label;
79      TPM2B                *context;
80      TPM_ALG_ID            hash;
81      TPM_ALG_ID            kdf;
82      UINT16                digestSize;
83      TPM2B_DIGEST          residual;
84  } KDF_STATE, *pKDR_STATE;
85  #define KDF_MAGIC    ((UINT32) 0x4048444a) // "KDF " backwards
```

Make sure that any other structures added to this union start with a 64-bit counter and a 32-bit magic number

```
86  typedef union
87  {
88      DRBG_STATE      drbg;
89      KDF_STATE       kdf;
90  } RAND_STATE;
```

This is the state used when the library uses a random number generator. A special function is installed for the library to call. That function picks up the state from this location and uses it for the generation of the random number.

```
91  extern RAND_STATE           *s_random;
92
93  // When instrumenting RSA key sieve
94  #if  RSA_INSTRUMENT
```

```
 95  #define PRIME_INDEX(x)  ((x) == 512 ? 0 : (x) == 1024 ? 1 : 2)
 96  #   define INSTRUMENT_SET(a, b) ((a) = (b))
 97  #   define INSTRUMENT_ADD(a, b) (a) = (a) + (b)
 98  #   define INSTRUMENT_INC(a)    (a) = (a) + 1
 99  extern UINT32  PrimeIndex;
100  extern UINT32  failedAtIteration[10];
101  extern UINT32  PrimeCounts[3];
102  extern UINT32  MillerRabinTrials[3];
103  extern UINT32  totalFieldsSieved[3];
104  extern UINT32  bitsInFieldAfterSieve[3];
105  extern UINT32  emptyFieldsSieved[3];
106  extern UINT32  noPrimeFields[3];
107  extern UINT32  primesChecked[3];
108  extern UINT16  lastSievePrime;
109  #else
110  #   define INSTRUMENT_SET(a, b)
111  #   define INSTRUMENT_ADD(a, b)
112  #   define INSTRUMENT_INC(a)
113  #endif
114  #endif // _CRYPT_RAND_H
```

 Copyright © TCG 2006-2020

10.1.5 CryptRsa.h

This file contains the RSA-related structures and defines.

```
1   #ifndef _CRYPT_RSA_H
2   #define _CRYPT_RSA_H
```

These values are used in the *bigNum* representation of various RSA values.

```
3   BN_TYPE(rsa, MAX_RSA_KEY_BITS);
4   #define BN_RSA(name)       BN_VAR(name, MAX_RSA_KEY_BITS)
5   #define BN_RSA_INITIALIZED(name, initializer)                    \
6       BN_INITIALIZED(name, MAX_RSA_KEY_BITS, initializer)
7   #define BN_PRIME(name)     BN_VAR(name, (MAX_RSA_KEY_BITS / 2))
8   BN_TYPE(prime, (MAX_RSA_KEY_BITS / 2));
9   #define BN_PRIME_INITIALIZED(name, initializer)                  \
10      BN_INITIALIZED(name, MAX_RSA_KEY_BITS / 2, initializer)
11  #if !CRT_FORMAT_RSA
12  #   error   This verson only works with CRT formatted data
13  #endif // !CRT_FORMAT_RSA
14  typedef struct privateExponent
15  {
16      bigNum              P;
17      bigNum              Q;
18      bigNum              dP;
19      bigNum              dQ;
20      bigNum              qInv;
21      bn_prime_t          entries[5];
22  } privateExponent;
23  #define     NEW_PRIVATE_EXPONENT(X)                                          \
24      privateExponent         _##X;                                            \
25      privateExponent         *X = RsaInitializeExponent(&(_##X))
26  #endif      // _CRYPT_RSA_H
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.1.6 CryptTest.h

This file contains constant definitions used for self-test.

```
1   #ifndef _CRYPT_TEST_H
2   #define _CRYPT_TEST_H
```

This is the definition of a bit array with one bit per algorithm.

NOTE: Since bit numbering starts at zero, when ALG_LAST_VALUE is a multiple of 8, ALGORITHM_VECTOR will need to have byte for the single bit in the last byte. So, for example, when ALG_LAST_VECTOR is 8, ALGORITHM_VECTOR will need 2 bytes.

```
3   #define ALGORITHM_VECTOR_BYTES  ((ALG_LAST_VALUE + 8) / 8)
4   typedef BYTE    ALGORITHM_VECTOR[ALGORITHM_VECTOR_BYTES];
5   #ifdef  TEST_SELF_TEST
6   LIB_EXPORT    extern  ALGORITHM_VECTOR    LibToTest;
7   #endif
8
9   // This structure is used to contain self-test tracking information for the
10  // cryptographic modules. Each of the major modules is given a 32-bit value in
11  // which it may maintain its own self test information. The convention for this
12  // state is that when all of the bits in this structure are 0, all functions need
13  // to be tested.
14  typedef struct
15  {
16      UINT32      rng;
17      UINT32      hash;
18      UINT32      sym;
19  #if ALG_RSA
20      UINT32      rsa;
21  #endif
22  #if ALG_ECC
23      UINT32      ecc;
24  #endif
25  } CRYPTO_SELF_TEST_STATE;
26
27  #endif // _CRYPT_TEST_H
```

 Copyright © TCG 2006-2020

10.1.7 HashTestData.h

Hash Test Vectors

```
1   TPM2B_TYPE(HASH_TEST_KEY, 128); // Twice the largest digest size
2   TPM2B_HASH_TEST_KEY        c_hashTestKey = {{128, {
3     0xa0,0xed,0x5c,0x9a,0xd2,0x4a,0x21,0x40,0x1a,0xd0,0x81,0x47,0x39,0x63,0xf9,0x50,
4     0xdc,0x59,0x47,0x11,0x40,0x13,0x99,0x92,0xc0,0x72,0xa4,0x0f,0xe2,0x33,0xe4,0x63,
5     0x9b,0xb6,0x76,0xc3,0x1e,0x6f,0x13,0xee,0xcc,0x99,0x71,0xa5,0xc0,0xcf,0x9a,0x40,
6     0xcf,0xdb,0x66,0x70,0x05,0x63,0x54,0x12,0x25,0xf4,0xe0,0x1b,0x23,0x35,0xe3,0x70,
7     0x7d,0x19,0x5f,0x00,0xe4,0xf1,0x61,0x73,0x05,0xd8,0x58,0x7f,0x60,0x61,0x84,0x36,
8     0xec,0xbe,0x96,0x1b,0x69,0x00,0xf0,0x9a,0x6e,0xe3,0x26,0x73,0x0d,0x17,0x5b,0x33,
9     0x41,0x44,0x9d,0x90,0xab,0xd9,0x6b,0x7d,0x48,0x99,0x25,0x93,0x29,0x14,0x2b,0xce,
10    0x93,0x8d,0x8c,0xaf,0x31,0x0e,0x9c,0x57,0xd8,0x5b,0x57,0x20,0x1b,0x9f,0x2d,0xa5
11    }}};
12
13  TPM2B_TYPE(HASH_TEST_DATA, 256); // Twice the largest block size
14  TPM2B_HASH_TEST_DATA       c_hashTestData = {{256, {
15    0x88,0xac,0xc3,0xe5,0x5f,0x66,0x9d,0x18,0x80,0xc9,0x7a,0x9c,0xa4,0x08,0x90,0x98,
16    0x0f,0x3a,0x53,0x92,0x4c,0x67,0x4e,0xb7,0x37,0xec,0x67,0x87,0xb6,0xbe,0x10,0xca,
17    0x11,0x5b,0x4a,0x0b,0x45,0xc3,0x32,0x68,0x48,0x69,0xce,0x25,0x1b,0xc8,0xaf,0x44,
18    0x79,0x22,0x83,0xc8,0xfb,0xe2,0x63,0x94,0xa2,0x3c,0x59,0x3e,0x3e,0xc6,0x64,0x2c,
19    0x1f,0x8c,0x11,0x93,0x24,0xa3,0x17,0xc5,0x2f,0x37,0xcf,0x95,0x97,0x8e,0x63,0x39,
20    0x68,0xd5,0xca,0xba,0x18,0x37,0x69,0x6e,0x4f,0x19,0xfd,0x8a,0xc0,0x8d,0x87,0x3a,
21    0xbc,0x31,0x42,0x04,0x05,0xef,0xb5,0x02,0xef,0x1e,0x92,0x4b,0xb7,0x73,0x2c,0x8c,
22    0xeb,0x23,0x13,0x81,0x34,0xb9,0xb5,0xc1,0x17,0x37,0x39,0xf8,0x3e,0xe4,0x4c,0x06,
23    0xa8,0x81,0x52,0x2f,0xef,0xc9,0x9c,0x69,0x89,0xbc,0x85,0x9c,0x30,0x16,0x02,0xca,
24    0xe3,0x61,0xd4,0x0f,0xed,0x34,0x1b,0xca,0xc1,0x1b,0xd1,0xfa,0xc1,0xa2,0xe0,0xdf,
25    0x52,0x2f,0x0b,0x4b,0x9f,0x0e,0x45,0x54,0xb9,0x17,0xb6,0xaf,0xd6,0xd5,0xca,0x90,
26    0x29,0x57,0x7b,0x70,0x50,0x94,0x5c,0x8e,0xf6,0x4e,0x21,0x8b,0xc6,0x8b,0xa6,0xbc,
27    0xb9,0x64,0xd4,0x4d,0xf3,0x68,0xd8,0xac,0xde,0xd8,0xd8,0xb5,0x6d,0xcd,0x93,0xeb,
28    0x28,0xa4,0xe2,0x5c,0x44,0xef,0xf0,0xe1,0x6f,0x38,0x1a,0x3c,0xe6,0xef,0xa2,0x9d,
29    0xb9,0xa8,0x05,0x2a,0x95,0xec,0x5f,0xdb,0xb0,0x25,0x67,0x9c,0x86,0x7a,0x8e,0xea,
30    0x51,0xcc,0xc3,0xd3,0xff,0x6e,0xf0,0xed,0xa3,0xae,0xf9,0x5d,0x33,0x70,0xf2,0x11
31    }}};
32
33  #if ALG_SHA1 == YES
34  TPM2B_TYPE(SHA1, 20);
35  TPM2B_SHA1     c_SHA1_digest = {{20, {
36    0xee,0x2c,0xef,0x93,0x76,0xbd,0xf8,0x91,0xbc,0xe6,0xe5,0x57,0x53,0x77,0x01,0xb5,
37    0x70,0x95,0xe5,0x40
38    }}};
39  #endif
40
41  #if ALG_SHA256 == YES
42  TPM2B_TYPE(SHA256, 32);
43  TPM2B_SHA256     c_SHA256_digest = {{32, {
44    0x64,0xe8,0xe0,0xc3,0xa9,0xa4,0x51,0x49,0x10,0x55,0x8d,0x31,0x71,0xe5,0x2f,0x69,
45    0x3a,0xdc,0xc7,0x11,0x32,0x44,0x61,0xbd,0x34,0x39,0x57,0xb0,0xa8,0x75,0x86,0x1b
46    }}};
47  #endif
48
49  #if ALG_SHA384 == YES
50  TPM2B_TYPE(SHA384, 48);
51  TPM2B_SHA384     c_SHA384_digest = {{48, {
52    0x37,0x75,0x29,0xb5,0x20,0x15,0x6e,0xa3,0x7e,0xa3,0x0d,0xcd,0x80,0xa8,0xa3,0x3d,
53    0xeb,0xe8,0xad,0x4e,0x1c,0x77,0x94,0x5a,0xaf,0x6c,0xd0,0xc1,0xfa,0x43,0x3f,0xc7,
54    0xb8,0xf1,0x01,0xc0,0x60,0xbf,0xf2,0x87,0xe8,0x71,0x9e,0x51,0x97,0xa0,0x09,0x8d
55    }}};
56  #endif
57
58  #if ALG_SHA512 == YES
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
59   TPM2B_TYPE(SHA512, 64);
60   TPM2B_SHA512    c_SHA512_digest = {{64, {
61     0xe2,0x7b,0x10,0x3d,0x5e,0x48,0x58,0x44,0x67,0xac,0xa3,0x81,0x8c,0x1d,0xc5,0x71,
62     0x66,0x92,0x8a,0x89,0xaa,0xd4,0x35,0x51,0x60,0x37,0x31,0xd7,0xba,0xe7,0x93,0x0b,
63     0x16,0x4d,0xb3,0xc8,0x34,0x98,0x3c,0xd3,0x53,0xde,0x5e,0xe8,0x0c,0xbc,0xaf,0xc9,
64     0x24,0x2c,0xcc,0xed,0xdb,0xde,0xba,0x1f,0x14,0x14,0x5a,0x95,0x80,0xde,0x66,0xbd
65     }}};
66   #endif
```

10.1.8 KdfTestData.h

Hash Test Vectors

```
1  #define TEST_KDF_KEY_SIZE   20
2  TPM2B_TYPE(KDF_TEST_KEY, TEST_KDF_KEY_SIZE);
3  TPM2B_KDF_TEST_KEY      c_kdfTestKeyIn = {{TEST_KDF_KEY_SIZE, {
4      0x27, 0x1F, 0xA0, 0x8B, 0xBD, 0xC5, 0x06, 0x0E, 0xC3, 0xDF,
5      0xA9, 0x28, 0xFF, 0x9B, 0x73, 0x12, 0x3A, 0x12, 0xDA, 0x0C }}};
6
7  TPM2B_TYPE(KDF_TEST_LABEL, 17);
8  TPM2B_KDF_TEST_LABEL    c_kdfTestLabel = {{17, {
9      0x4B, 0x44, 0x46, 0x53, 0x45, 0x4C, 0x46, 0x54,
10     0x45, 0x53, 0x54, 0x4C, 0x41, 0x42, 0x45, 0x4C, 0x00 }}};
11
12 TPM2B_TYPE(KDF_TEST_CONTEXT, 8);
13 TPM2B_KDF_TEST_CONTEXT  c_kdfTestContextU = {{8, {
14     0xCE, 0x24, 0x4F, 0x39, 0x5D, 0xCA, 0x73, 0x91 }}};
15
16 TPM2B_KDF_TEST_CONTEXT  c_kdfTestContextV = {{8, {
17     0xDA, 0x50, 0x40, 0x31, 0xDD, 0xF1, 0x2E, 0x83 }}};
18
19 #if ALG_SHA512 == ALG_YES
20     TPM2B_KDF_TEST_KEY  c_kdfTestKeyOut = {{20, {
21         0x8b, 0xe2, 0xc1, 0xb8, 0x5b, 0x78, 0x56, 0x9b, 0x9f, 0xa7,
22         0x59, 0xf5, 0x85, 0x7c, 0x56, 0xd6, 0x84, 0x81, 0x0f, 0xd3 }}};
23     #define KDF_TEST_ALG    TPM_ALG_SHA512
24
25 #elif ALG_SHA384 == ALG_YES
26     TPM2B_KDF_TEST_KEY  c_kdfTestKeyOut = {{20, {
27         0x1d, 0xce, 0x70, 0xc9, 0x11, 0x3e, 0xb2, 0xdb, 0xa4, 0x7b,
28         0xd9, 0xcf, 0xc7, 0x2b, 0xf4, 0x6f, 0x45, 0xb0, 0x93, 0x12 }}};
29     #define KDF_TEST_ALG    TPM_ALG_SHA384
30
31 #elif ALG_SHA256 == ALG_YES
32     TPM2B_KDF_TEST_KEY  c_kdfTestKeyOut = {{20, {
33         0xbb, 0x02, 0x59, 0xe1, 0xc8, 0xba, 0x60, 0x7e, 0x6a, 0x2c,
34         0xd7, 0x04, 0xb6, 0x9a, 0x90, 0x2e, 0x9a, 0xde, 0x84, 0xc4 }}};
35     #define KDF_TEST_ALG    TPM_ALG_SHA256
36
37 #elif ALG_SHA1 == ALG_YES
38     TPM2B_KDF_TEST_KEY  c_kdfTestKeyOut = {{20, {
39         0x55, 0xb5, 0xa7, 0x18, 0x4a, 0xa0, 0x74, 0x23, 0xc4, 0x7d,
40         0xae, 0x76, 0x6c, 0x26, 0xa2, 0x37, 0x7d, 0x7c, 0xf8, 0x51 }}};
41     #define KDF_TEST_ALG    TPM_ALG_SHA1
42 #endif
```

 Copyright © TCG 2006-2020

10.1.9 RsaTestData.h

RSA Test Vectors

```
#define RSA_TEST_KEY_SIZE   256
typedef struct
{
    UINT16      size;
    BYTE        buffer[RSA_TEST_KEY_SIZE];
} TPM2B_RSA_TEST_KEY;
typedef TPM2B_RSA_TEST_KEY  TPM2B_RSA_TEST_VALUE;
typedef struct
{
    UINT16       size;
    BYTE        buffer[RSA_TEST_KEY_SIZE / 2];
} TPM2B_RSA_TEST_PRIME;
const TPM2B_RSA_TEST_KEY    c_rsaPublicModulus = {256, {
  0x91,0x12,0xf5,0x07,0x9d,0x5f,0x6b,0x1c,0x90,0xf6,0xcc,0x87,0xde,0x3a,0x7a,0x15,
  0xdc,0x54,0x07,0x6c,0x26,0x8f,0x25,0xef,0x7e,0x66,0xc0,0xe3,0x82,0x12,0x2f,0xab,
  0x52,0x82,0x1e,0x85,0xbc,0x53,0xba,0x2b,0x01,0xad,0x01,0xc7,0x8d,0x46,0x4f,0x7d,
  0xdd,0x7e,0xdc,0xb0,0xad,0xf6,0x0c,0xa1,0x62,0x92,0x97,0x8a,0x3e,0x6f,0x7e,0x3e,
  0xf6,0x9a,0xcc,0xf9,0xa9,0x86,0x77,0xb6,0x85,0x43,0x42,0x04,0x13,0x65,0xe2,0xad,
  0x36,0xc9,0xbf,0xc1,0x97,0x84,0x6f,0xee,0x7c,0xda,0x58,0xd2,0xae,0x07,0x00,0xaf,
  0xc5,0x5f,0x4d,0x3a,0x98,0xb0,0xed,0x27,0x7c,0xc2,0xce,0x26,0x5d,0x87,0xe1,0xe3,
  0xa9,0x69,0x88,0x4f,0x8c,0x08,0x31,0x18,0xae,0x93,0x16,0xe3,0x74,0xde,0xd3,0xf6,
  0x16,0xaf,0xa3,0xac,0x37,0x91,0x8d,0x10,0xc6,0x6b,0x64,0x14,0x3a,0xd9,0xfc,0xe4,
  0xa0,0xf2,0xd1,0x01,0x37,0x4f,0x4a,0xeb,0xe5,0xec,0x98,0xc5,0xd9,0x4b,0x30,0xd2,
  0x80,0x2a,0x5a,0x18,0x5a,0x7d,0xd4,0x3d,0xb7,0x62,0x98,0xce,0x6d,0xa2,0x02,0x6e,
  0x45,0xaa,0x95,0x73,0xe0,0xaa,0x75,0x57,0xb1,0x3d,0x1b,0x05,0x75,0x23,0x6b,0x20,
  0x69,0x9e,0x14,0xb0,0x7f,0xac,0xae,0xd2,0xc7,0x48,0x3b,0xe4,0x56,0x11,0x34,0x1e,
  0x05,0x1a,0x30,0x20,0xef,0x68,0x93,0x6b,0x9d,0x7e,0xdd,0xba,0x96,0x50,0xcc,0x1c,
  0x81,0xb4,0x59,0xb9,0x74,0x36,0xd9,0x97,0xdc,0x8f,0x17,0x82,0x72,0xb3,0x59,0xf6,

0x23,0xfa,0x84,0xf7,0x6d,0xf2,0x05,0xff,0xf1,0xb9,0xcc,0xe9,0xa2,0x82,0x01,0xfb}};

const TPM2B_RSA_TEST_PRIME    c_rsaPrivatePrime = {RSA_TEST_KEY_SIZE / 2, {
  0xb7,0xa0,0x90,0xc7,0x92,0x09,0xde,0x71,0x03,0x37,0x4a,0xb5,0x2f,0xda,0x61,0xb8,
  0x09,0x1b,0xba,0x99,0x70,0x45,0xc1,0x0b,0x15,0x12,0x71,0x8a,0xb3,0x2a,0x4d,0x5a,
  0x41,0x9b,0x73,0x89,0x80,0x0a,0x8f,0x18,0x4c,0x8b,0xa2,0x5b,0xda,0xbd,0x43,0xbe,
  0xdc,0x76,0x4d,0x71,0x0f,0xb9,0xfc,0x7a,0x09,0xfe,0x4f,0xac,0x63,0xd9,0x2e,0x50,
  0x3a,0xa1,0x37,0xc6,0xf2,0xa1,0x89,0x12,0xe7,0x72,0x64,0x2b,0xba,0xc1,0x1f,0xca,
  0x9d,0xb7,0xaa,0x3a,0xa9,0xd3,0xa6,0x6f,0x73,0x02,0xbb,0x85,0x5d,0x9a,0xb9,0x5c,
  0x08,0x83,0x22,0x20,0x49,0x91,0x5f,0x4b,0x86,0xbc,0x3f,0x76,0x43,0x08,0x97,0xbf,

0x82,0x55,0x36,0x2d,0x8b,0x6e,0x9e,0xfb,0xc1,0x67,0x6a,0x43,0xa2,0x46,0x81,0x71}};

const BYTE          c_RsaTestValue[RSA_TEST_KEY_SIZE] = {
  0x2a,0x24,0x3a,0xbb,0x50,0x1d,0xd4,0x2a,0xf9,0x18,0x32,0x34,0xa2,0x0f,0xea,0x5c,
  0x91,0x77,0xe9,0xe1,0x09,0x83,0xdc,0x5f,0x71,0x64,0x5b,0xeb,0x57,0x79,0xa0,0x41,
  0xc9,0xe4,0x5a,0x0b,0xf4,0x9f,0xdb,0x84,0x04,0xa6,0x48,0x24,0xf6,0x3f,0x66,0x1f,
  0xa8,0x04,0x5c,0xf0,0x7a,0x6b,0x4a,0x9c,0x7e,0x21,0xb6,0xda,0x6b,0x65,0x9c,0x3a,
  0x68,0x50,0x13,0x1e,0xa4,0xb7,0xca,0xec,0xd3,0xcc,0xb2,0x9b,0x8c,0x87,0xa4,0x6a,
  0xba,0xc2,0x06,0x3f,0x40,0x48,0x7b,0xa8,0xb8,0x2c,0x03,0x14,0x33,0xf3,0x1d,0xe9,
  0xbd,0x6f,0x54,0x66,0xb4,0x69,0x5e,0xbc,0x80,0x7c,0xe9,0x6a,0x43,0x7f,0xb8,0x6a,
  0xa0,0x5f,0x5d,0x7a,0x20,0xfd,0x7a,0x39,0xe1,0xea,0x0e,0x94,0x91,0x28,0x63,0x7a,
  0xac,0xc9,0xa5,0x3a,0x6d,0x31,0x7b,0x7c,0x54,0x56,0x99,0x56,0xbb,0xb7,0xa1,0x2d,
  0xd2,0x5c,0x91,0x5f,0x1c,0xd3,0x06,0x7f,0x34,0x53,0x2f,0x4c,0xd1,0x8b,0xd2,0x9e,
  0xdc,0xc3,0x94,0x0a,0xe1,0x0f,0xa5,0x15,0x46,0x2a,0x8e,0x10,0xc2,0xfe,0xb7,0x5e,
  0x2d,0x0d,0xd1,0x25,0xfc,0xe4,0xf7,0x02,0x19,0xfe,0xb6,0xe4,0x95,0x9c,0x17,0x4a,
  0x9b,0xdb,0xab,0xc7,0x79,0xe3,0x5e,0x40,0xd0,0x56,0x6d,0x25,0x0a,0x72,0x65,0x80,
  0x92,0x9a,0xa8,0x07,0x70,0x32,0x14,0xfb,0xfe,0x08,0xeb,0x13,0xb4,0x07,0x68,0xb4,
  0x58,0x39,0xbe,0x8e,0x78,0x3a,0x59,0x3f,0x9c,0x4c,0xe9,0xa8,0x64,0x68,0xf7,0xb9,
```

```
57
    0x6e,0x20,0xf5,0xcb,0xca,0x47,0xf2,0x17,0xaa,0x8b,0xbc,0x13,0x14,0x84,0xf6,0xab};
58
59  const TPM2B_RSA_TEST_VALUE    c_RsaepKvt = {RSA_TEST_KEY_SIZE, {
60    0x73,0xbd,0x65,0x49,0xda,0x7b,0xb8,0x50,0x9e,0x87,0xf0,0x0a,0x8a,0x9a,0x07,0xb6,
61    0x00,0x82,0x10,0x14,0x60,0xd8,0x01,0xfc,0xc5,0x18,0xea,0x49,0x5f,0x13,0xcf,0x65,
62    0x66,0x30,0x6c,0x60,0x3f,0x24,0x3c,0xfb,0xe2,0x31,0x16,0x99,0x7e,0x31,0x98,0xab,
63    0x93,0xb8,0x07,0x53,0xcc,0xdb,0x7f,0x44,0xd9,0xee,0x5d,0xe8,0x5f,0x97,0x5f,0xe8,
64    0x1f,0x88,0x52,0x24,0x7b,0xac,0x62,0x95,0xb7,0x7d,0xf5,0xf8,0x9f,0x5a,0xa8,0x24,
65    0x9a,0x76,0x71,0x2a,0x35,0x2a,0xa1,0x08,0xbb,0x95,0xe3,0x64,0xdc,0xdb,0xc2,0x33,
66    0xa9,0x5f,0xbe,0x4c,0xc4,0xcc,0x28,0xc9,0x25,0xff,0xee,0x17,0x15,0x9a,0x50,0x90,
67    0x0e,0x15,0xb4,0xea,0x6a,0x09,0xe6,0xff,0xa4,0xee,0xc7,0x7e,0xce,0xa9,0x73,0xe4,
68    0xa0,0x56,0xbd,0x53,0x2a,0xe4,0xc0,0x2b,0xa8,0x9b,0x09,0x30,0x72,0x62,0x0f,0xf9,
69    0xf6,0xa1,0x52,0xd2,0x8a,0x37,0xee,0xa5,0xc8,0x47,0xe1,0x99,0x21,0x47,0xeb,0xdd,
70    0x37,0xaa,0xe4,0xbd,0x55,0x46,0x5a,0x5a,0x5d,0xfb,0x7b,0xfc,0xff,0xbf,0x26,0x71,
71    0xf6,0x1e,0xad,0xbc,0xbf,0x33,0xca,0xe1,0x92,0x8f,0x2a,0x89,0x6c,0x45,0x24,0xd1,
72    0xa6,0x52,0x56,0x24,0x5e,0x90,0x47,0xe5,0xcb,0x12,0xb0,0x32,0xf9,0xa6,0xbb,0xea,
73    0x37,0xa9,0xbd,0xef,0x23,0xef,0x63,0x07,0x6c,0xc4,0x4e,0x64,0x3c,0xc6,0x11,0x84,
74    0x7d,0x65,0xd6,0x5d,0x7a,0x17,0x58,0xa5,0xf7,0x74,0x3b,0x42,0xe3,0xd2,0xda,0x5f,
75
    0x6f,0xe0,0x1e,0x4b,0xcf,0x46,0xe2,0xdf,0x3e,0x41,0x8e,0x0e,0xb0,0x3f,0x8b,0x65}};
76
77  #define    OAEP_TEST_LABEL     "OAEP Test Value"
78
79  #if ALG_SHA1_VALUE == DEFAULT_TEST_HASH
80
81  const TPM2B_RSA_TEST_VALUE    c_OaepKvt = {RSA_TEST_KEY_SIZE, {
82    0x32,0x68,0x84,0x0b,0x9c,0xc9,0x25,0x26,0xd9,0xc0,0xd0,0xb1,0xde,0x60,0x55,0xae,
83    0x33,0xe5,0xcf,0x6c,0x85,0xbe,0x0d,0x71,0x11,0xe1,0x45,0x60,0xbb,0x42,0x3d,0xf3,
84    0xb1,0x18,0x84,0x7b,0xc6,0x5d,0xce,0x1d,0x5f,0x9a,0x97,0xcf,0xb1,0x97,0x9a,0x85,
85    0x7c,0xa7,0xa1,0x63,0x23,0xb6,0x74,0x0f,0x1a,0xee,0x29,0x51,0xeb,0x50,0x8f,0x3c,
86    0x8e,0x4e,0x31,0x38,0xdc,0x11,0xfc,0x9a,0x4e,0xaf,0x93,0xc9,0x7f,0x6e,0x35,0xf3,
87    0xc9,0xe4,0x89,0x14,0x53,0xe2,0xc2,0x1a,0xf7,0x6b,0x9b,0xf0,0x7a,0xa4,0x69,0x52,
88    0xe0,0x24,0x8f,0xea,0x31,0xa7,0x5c,0x43,0xb0,0x65,0xc9,0xfe,0xba,0xfe,0x80,0x9e,
89    0xa5,0xc0,0xf5,0x8d,0xce,0x41,0xf9,0x83,0x0d,0x8e,0x0f,0xef,0x3d,0x1f,0x6a,0xcc,
90    0x8a,0x3d,0x3b,0xdf,0x22,0x38,0xd7,0x34,0x58,0x7b,0x55,0xc9,0xf6,0xbc,0x7c,0x4c,
91    0x3f,0xd7,0xde,0x4e,0x30,0xa9,0x69,0xf3,0x5f,0x56,0x8f,0xc2,0xe7,0x75,0x79,0xb8,
92    0xa5,0xc8,0x0d,0xc0,0xcd,0xb6,0xc9,0x63,0xad,0x7c,0xe4,0x8f,0x39,0x60,0x4d,0x7d,
93    0xdb,0x34,0x49,0x2a,0x47,0xde,0xc0,0x42,0x4a,0x19,0x94,0x2e,0x50,0x21,0x03,0x47,
94    0xff,0x73,0xb3,0xb7,0x89,0xcc,0x7b,0x2c,0xeb,0x03,0xa7,0x9a,0x06,0xfd,0xed,0x19,
95    0xbb,0x82,0xa0,0x13,0xe9,0xfa,0xac,0x06,0x5f,0xc5,0xa9,0x2b,0xda,0x88,0x23,0xa2,
96    0x5d,0xc2,0x7f,0xda,0xc8,0x5a,0x94,0x31,0xc1,0x21,0xd7,0x1e,0x6b,0xd7,0x89,0xb1,
97
    0x93,0x80,0xab,0xd1,0x37,0xf2,0x6f,0x50,0xcd,0x2a,0xea,0xb1,0xc4,0xcd,0xcb,0xb5}};
98
99  const TPM2B_RSA_TEST_VALUE    c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
100   0x29,0xa4,0x2f,0xbb,0x8a,0x14,0x05,0x1e,0x3c,0x72,0x76,0x77,0x38,0xe7,0x73,0xe3,
101   0x6e,0x24,0x4b,0x38,0xd2,0x1a,0xcf,0x23,0x58,0x78,0x36,0x82,0x23,0x6e,0x6b,0xef,
102   0x2c,0x3d,0xf2,0xe8,0xd6,0xc6,0x87,0x8e,0x78,0x9b,0x27,0x39,0xc0,0xd6,0xef,0x4d,
103   0x0b,0xfc,0x51,0x27,0x18,0xf3,0x51,0x5e,0x4d,0x96,0x3a,0xe2,0x15,0xe2,0x7e,0x42,
104   0xf4,0x16,0xd5,0xc6,0x52,0x5d,0x17,0x44,0x76,0x09,0x7a,0xcf,0xe3,0x30,0xe3,0x84,
105   0xf6,0x6f,0x3a,0x33,0xfb,0x32,0x0d,0x1d,0xe7,0x7c,0x80,0x82,0x4f,0xed,0xda,0x87,
106   0x11,0x9c,0xc3,0x7e,0x85,0xbd,0x18,0x58,0x08,0x2b,0x23,0x37,0xe7,0x9d,0xd0,0xd1,
107   0x79,0xe2,0x05,0xbd,0xf5,0x4f,0x0e,0x0f,0xdb,0x4a,0x74,0xeb,0x09,0x01,0xb3,0xca,
108   0xbd,0xa6,0x7b,0x09,0xb1,0x13,0x77,0x30,0x4d,0x87,0x41,0x06,0x57,0x2e,0x5f,0x36,
109   0x6e,0xfc,0x35,0x69,0xfe,0x0a,0x24,0x6c,0x98,0x8c,0xda,0x97,0xf4,0xfb,0xc7,0x83,
110   0x2d,0x3e,0x7d,0xc0,0x5c,0x34,0xfd,0x11,0x2a,0x12,0xa7,0xae,0x4a,0xde,0xc8,0x4e,
111   0xcf,0xf4,0x85,0x63,0x77,0xc6,0x33,0x34,0xe0,0x27,0xe4,0x9e,0x91,0x0b,0x4b,0x85,
112   0xf0,0xb0,0x79,0xaa,0x7c,0xc6,0xff,0x3b,0xbc,0x04,0x73,0xb8,0x95,0xd7,0x31,0x54,
113   0x3b,0x56,0xec,0x52,0x15,0xd7,0x3e,0x62,0xf5,0x82,0x99,0x3e,0x2a,0xc0,0x4b,0x2e,
114   0x06,0x57,0x6d,0x3f,0x3e,0x77,0x1f,0x2b,0x2d,0xc5,0xb9,0x3b,0x68,0x56,0x73,0x70,
115
    0x32,0x6b,0x6b,0x65,0x25,0x76,0x45,0x6c,0x45,0xf1,0x6c,0x59,0xfc,0x94,0xa7,0x15}};
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
116
117 const TPM2B_RSA_TEST_VALUE    c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
118   0x01,0xfe,0xd5,0x83,0x0b,0x15,0xba,0x90,0x2c,0xdf,0xf7,0x26,0xb7,0x8f,0xb1,0xd7,
119   0x0b,0xfd,0x83,0xf9,0x95,0xd5,0xd7,0xb5,0xc5,0xc5,0x4a,0xde,0xd5,0xe6,0x20,0x78,
120   0xca,0x73,0x77,0x3d,0x61,0x36,0x48,0xae,0x3e,0x8f,0xee,0x43,0x29,0x96,0xdf,0x3f,
121   0x1c,0x97,0x5a,0xbe,0xe5,0xa2,0x7e,0x5b,0xd0,0xc0,0x29,0x39,0x83,0x81,0x77,0x24,
122   0x43,0xdb,0x3c,0x64,0x4d,0xf0,0x23,0xe4,0xae,0x0f,0x78,0x31,0x8c,0xda,0x0c,0xec,
123   0xf1,0xdf,0x09,0xf2,0x14,0x6a,0x4d,0xaf,0x36,0x81,0x6e,0xbd,0xbe,0x36,0x79,0x88,
124   0x98,0xb6,0x6f,0x5a,0xad,0xcf,0x7c,0xee,0xe0,0xdd,0x00,0xbe,0x59,0x97,0x88,0x00,
125   0x34,0xc0,0x8b,0x48,0x42,0x05,0x04,0x5a,0xb7,0x85,0x38,0xa0,0x35,0xd7,0x3b,0x51,
126   0xb8,0x7b,0x81,0x83,0xee,0xff,0x76,0x6f,0x50,0x39,0x4d,0xab,0x89,0x63,0x07,0x6d,
127   0xf5,0xe5,0x01,0x10,0x56,0xfe,0x93,0x06,0x8f,0xd3,0xc9,0x41,0xab,0xc9,0xdf,0x6e,
128   0x59,0xa8,0xc3,0x1d,0xbf,0x96,0x4a,0x59,0x80,0x3c,0x90,0x3a,0x59,0x56,0x4c,0x6d,
129   0x44,0x6d,0xeb,0xdc,0x73,0xcd,0xc1,0xec,0xb8,0x41,0xbf,0x89,0x8c,0x03,0x69,0x4c,
130   0xaf,0x3f,0xc1,0xc5,0xc7,0xe7,0x7d,0xa7,0x83,0x39,0x70,0xa2,0x6b,0x83,0xbc,0xbe,
131   0xf5,0xbf,0x1c,0xee,0x6e,0xa3,0x22,0x1e,0x25,0x2f,0x16,0x68,0x69,0x5a,0x1d,0xfa,
132   0x2c,0x3a,0x0f,0x67,0xe1,0x77,0x12,0xe8,0x3d,0xba,0xaa,0xef,0x96,0x9c,0x1f,0x64,
133
    0x32,0xf4,0xa7,0xb3,0x3f,0x7d,0x61,0xbb,0x9a,0x27,0xad,0xfb,0x2f,0x33,0xc4,0x70}};
134
135 const TPM2B_RSA_TEST_VALUE    c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
136   0x67,0x4e,0xdd,0xc2,0xd2,0x6d,0xe0,0x03,0xc4,0xc2,0x41,0xd3,0xd4,0x61,0x30,0xd0,
137   0xe1,0x68,0x31,0x4a,0xda,0xd9,0xc2,0x5d,0xaa,0xa2,0x7b,0xfb,0x44,0x02,0xf5,0xd6,
138   0xd8,0x2e,0xcd,0x13,0x36,0xc9,0x4b,0xdb,0x1a,0x4b,0x66,0x1b,0x4f,0x9c,0xb7,0x17,
139   0xac,0x53,0x37,0x4f,0x21,0xbd,0x0c,0x66,0xac,0x06,0x65,0x52,0x9f,0x04,0xf6,0xa5,
140   0x22,0x5b,0xf7,0xe6,0x0d,0x3c,0x9f,0x41,0x19,0x09,0x88,0x7c,0x41,0x4c,0x2f,0x9c,
141   0x8b,0x3c,0xdd,0x7c,0x28,0x78,0x24,0xd2,0x09,0xa6,0x5b,0xf7,0x3c,0x88,0x7e,0x73,
142   0x5a,0x2d,0x36,0x02,0x4f,0x65,0xb0,0xcb,0xc8,0xdc,0xac,0xa2,0xda,0x8b,0x84,0x91,
143   0x71,0xe4,0x30,0x8b,0xb6,0x12,0xf2,0xf0,0xd0,0xa0,0x38,0xcf,0x75,0xb7,0x20,0xcb,
144   0x35,0x51,0x52,0x6b,0xc4,0xf4,0x21,0x95,0xc2,0xf7,0x9a,0x13,0xc1,0x1a,0x7b,0x8f,
145   0x77,0xda,0x19,0x48,0xbb,0x6d,0x14,0x5d,0xba,0x65,0xb4,0x9e,0x43,0x42,0x58,0x98,
146   0x0b,0x91,0x46,0xd8,0x4c,0xf3,0x4c,0xaf,0x2e,0x02,0xa6,0xb2,0x49,0x12,0x62,0x43,
147   0x4e,0xa8,0xac,0xbf,0xfd,0xfa,0x37,0x24,0xea,0x69,0x1c,0xf5,0xae,0xfa,0x08,0x82,
148   0x30,0xc3,0xc0,0xf8,0x9a,0x89,0x33,0xe1,0x40,0x6d,0x18,0x5c,0x7b,0x90,0x48,0xbf,
149   0x37,0xdb,0xea,0xfb,0x0e,0xd4,0x2e,0x11,0xfa,0xa9,0x86,0xff,0x00,0x0b,0x7b,0xca,
150   0x09,0x64,0x6a,0x8f,0x0c,0x0e,0x09,0x14,0x36,0x4a,0x74,0x31,0x18,0x5b,0x18,0xeb,
151
    0xea,0x83,0xc3,0x66,0x68,0xa6,0x7d,0x43,0x06,0x0f,0x99,0x60,0xce,0x65,0x08,0xf6}};
152
153 #endif // SHA1
154
155 #if ALG_SHA256_VALUE == DEFAULT_TEST_HASH
156
157 const TPM2B_RSA_TEST_VALUE    c_OaepKvt = {RSA_TEST_KEY_SIZE, {
158   0x33,0x20,0x6e,0x21,0xc3,0xf6,0xcd,0xf8,0xd7,0x5d,0x9f,0xe9,0x05,0x14,0x8c,0x7c,
159   0xbb,0x69,0x24,0x9e,0x52,0x8f,0xaf,0x84,0x73,0x21,0x2c,0x85,0xa5,0x30,0x4d,0xb6,
160   0xb8,0xfa,0x15,0x9b,0xc7,0x8f,0xc9,0x7a,0x72,0x4b,0x85,0xa4,0x1c,0xc5,0xd8,0xe4,
161   0x92,0xb3,0xec,0xd9,0xa8,0xca,0x5e,0x74,0x73,0x89,0x7f,0xb4,0xac,0x7e,0x68,0x12,
162   0xb2,0x53,0x27,0x4b,0xbf,0xd0,0x71,0x69,0x46,0x9f,0xef,0xf4,0x70,0x60,0xf8,0xd7,
163   0xae,0xc7,0x5a,0x27,0x38,0x25,0x2d,0x25,0xab,0x96,0x56,0x66,0x3a,0x23,0x40,0xa8,
164   0xdb,0xbc,0x86,0xe8,0xf3,0xd2,0x58,0x0b,0x44,0xfc,0x94,0x1e,0xb7,0x5d,0xb4,0x57,
165   0xb5,0xf3,0x56,0xee,0x9b,0xcf,0x97,0x91,0x29,0x36,0xe3,0x06,0x13,0xa2,0xea,0xd6,
166   0xd6,0x0b,0x86,0x0b,0x1a,0x27,0xe6,0x22,0xc4,0x7b,0xff,0xde,0x0f,0xbf,0x79,0xc8,
167   0x1b,0xed,0xf1,0x27,0x62,0xb5,0x8b,0xf9,0xd9,0x76,0x90,0xf6,0xcc,0x83,0x0f,0xce,
168   0xce,0x2e,0x63,0x7a,0x9b,0xf4,0x48,0x5b,0xd7,0x81,0x2c,0x3a,0xdb,0x59,0x0d,0x4d,
169   0x9e,0x46,0xe9,0x9e,0x92,0x22,0x27,0x1c,0xb0,0x67,0x8a,0xe6,0x8a,0x16,0x8a,0xdf,
170   0x95,0x76,0x24,0x82,0xad,0xf1,0xbc,0x97,0xbf,0xd3,0x5e,0x6e,0x14,0x0c,0x5b,0x25,
171   0xfe,0x58,0xfa,0x64,0xe5,0x14,0x46,0xb7,0x58,0xc6,0x3f,0x7f,0x42,0xd2,0x8e,0x45,
172   0x13,0x41,0x85,0x12,0x2e,0x96,0x19,0xd0,0x5e,0x7d,0x34,0x06,0x32,0x2b,0xc8,0xd9,
173
    0x0d,0x6c,0x06,0x36,0xa0,0xff,0x47,0x57,0x2c,0x25,0xbc,0x8a,0xa5,0xe2,0xc7,0xe3}};
174
175 const TPM2B_RSA_TEST_VALUE    c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
```

 Copyright © TCG 2006-2020

```
176     0x39,0xfc,0x10,0x5d,0xf4,0x45,0x3d,0x94,0x53,0x06,0x89,0x24,0xe7,0xe8,0xfd,0x03,
177     0xac,0xfd,0xbd,0xb2,0x28,0xd3,0x4a,0x52,0xc5,0xd4,0xdb,0x17,0xd4,0x24,0x05,0xc4,
178     0xeb,0x6a,0xce,0x1d,0xbb,0x37,0xcb,0x09,0xd8,0x6c,0x83,0x19,0x93,0xd4,0xe2,0x88,
179     0x88,0x9b,0xaf,0x92,0x16,0xc4,0x15,0xbd,0x49,0x13,0x22,0xb7,0x84,0xcf,0x23,0xf2,
180     0x6f,0x0c,0x3e,0x8f,0xde,0x04,0x09,0x31,0x2d,0x99,0xdf,0xe6,0x74,0x70,0x30,0xde,
181     0x8c,0xad,0x32,0x86,0xe2,0x7c,0x12,0x90,0x21,0xf3,0x86,0xb7,0xe2,0x64,0xca,0x98,
182     0xcc,0x64,0x4b,0xef,0x57,0x4f,0x5a,0x16,0x6e,0xd7,0x2f,0x5b,0xf6,0x07,0xad,0x33,
183     0xb4,0x8f,0x3b,0x3a,0x8b,0xd9,0x06,0x2b,0xed,0x3c,0x3c,0x76,0xf6,0x21,0x31,0xe3,
184     0xfb,0x2c,0x45,0x61,0x42,0xba,0xe0,0xc3,0x72,0x63,0xd0,0x6b,0x8f,0x36,0x26,0xfb,
185     0x9e,0x89,0x0e,0x44,0x9a,0xc1,0x84,0x5e,0x84,0x8d,0xb6,0xea,0xf1,0x0d,0x66,0xc7,
186     0xdb,0x44,0xbd,0x19,0x7c,0x05,0xbe,0xc4,0xab,0x88,0x32,0xbe,0xc7,0x63,0x31,0xe6,
187     0x38,0xd4,0xe5,0xb8,0x4b,0xf5,0x0e,0x55,0x9a,0x3a,0xe6,0x0a,0xec,0xee,0xe2,0xa8,
188     0x88,0x04,0xf2,0xb8,0xaa,0x5a,0xd8,0x97,0x5d,0xa0,0xa8,0x42,0xfb,0xd9,0xde,0x80,
189     0xae,0x4c,0xb3,0xa1,0x90,0x47,0x57,0x03,0x10,0x78,0xa6,0x8f,0x11,0xba,0x4b,0xce,
190     0x2d,0x56,0xa4,0xe1,0xbd,0xf8,0xa0,0xa4,0xd5,0x48,0x3c,0x63,0x20,0x00,0x38,0xa0,
191
    0xd1,0xe6,0x12,0xe9,0x1d,0xd8,0x49,0xe3,0xd5,0x24,0xb5,0xc5,0x3a,0x1f,0xb0,0xd4}};
192
193 const TPM2B_RSA_TEST_VALUE    c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
194     0x74,0x89,0x29,0x3e,0x1b,0xac,0xc6,0x85,0xca,0xf0,0x63,0x43,0x30,0x7d,0x1c,0x9b,
195     0x2f,0xbd,0x4d,0x69,0x39,0x5e,0x85,0xe2,0xef,0x86,0x0a,0xc6,0x6b,0xa6,0x08,0x19,
196     0x6c,0x56,0x38,0x24,0x55,0x92,0x84,0x9b,0x1b,0x8b,0x04,0xcf,0x24,0x14,0x24,0x13,
197     0x0e,0x8b,0x82,0x6f,0x96,0xc8,0x9a,0x68,0xfc,0x4c,0x02,0xf0,0xdc,0xcd,0x36,0x25,
198     0x31,0xd5,0x82,0xcf,0xc9,0x69,0x72,0xf6,0x1d,0xab,0x68,0x20,0x2e,0x2d,0x19,0x49,
199     0xf0,0x2e,0xad,0xd2,0xda,0xaf,0xff,0xb6,0x92,0x83,0x5b,0x8a,0x06,0x2d,0x0c,0x32,
200     0x11,0x32,0x3b,0x77,0x17,0xf6,0x50,0xfb,0xf8,0x57,0xc9,0xc7,0x9b,0x9e,0xc6,0xd1,
201     0xa9,0x55,0xf0,0x22,0x35,0xda,0xca,0x3c,0x8e,0xc6,0x9a,0xd8,0x25,0xc8,0x5e,0x93,
202     0x0d,0xaa,0xa7,0x06,0xaf,0x11,0x29,0x99,0xe7,0x7c,0xee,0x49,0x82,0x30,0xba,0x2c,
203     0xe2,0x40,0x8f,0x0a,0xa6,0x7b,0x24,0x75,0xc5,0xcd,0x03,0x12,0xf4,0xb2,0x4b,0x3a,
204     0xd1,0x91,0x3c,0x20,0x0e,0x58,0x2b,0x31,0xf8,0x8b,0xee,0xbc,0x1f,0x95,0x35,0x58,
205     0x6a,0x73,0xee,0x99,0xb0,0x01,0x42,0x4f,0x66,0xc0,0x66,0xbb,0x35,0x86,0xeb,0xd9,
206     0x7b,0x55,0x77,0x2d,0x54,0x78,0x19,0x49,0xe8,0xcc,0xfd,0xb1,0xcb,0x49,0xc9,0xea,
207     0x20,0xab,0xed,0xb5,0xed,0xfe,0xb2,0xb5,0xa8,0xcf,0x05,0x06,0xd5,0x7d,0x2b,0xbb,
208     0x0b,0x65,0x6b,0x2b,0x6d,0x55,0x95,0x85,0x44,0x8b,0x12,0x05,0xf3,0x4b,0xd4,0x8e,
209
    0x3d,0x68,0x2d,0x29,0x9c,0x05,0x79,0xd6,0xfc,0x72,0x90,0x6a,0xab,0x46,0x38,0x81}};
210
211 const TPM2B_RSA_TEST_VALUE    c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
212     0x8a,0xb1,0x0a,0xb5,0xe4,0x02,0xf7,0xdd,0x45,0x2a,0xcc,0x2b,0x6b,0x8c,0x0e,0x9a,
213     0x92,0x4f,0x9b,0xc5,0xe4,0x8b,0x82,0xb9,0xb0,0xd9,0x87,0x8c,0xcb,0xf0,0xb0,0x59,
214     0xa5,0x92,0x21,0xa0,0xa7,0x61,0x5c,0xed,0xa8,0x6e,0x22,0x29,0x46,0xc7,0x86,0x37,
215     0x4b,0x1b,0x1e,0x94,0x93,0xc8,0x4c,0x17,0x7a,0xae,0x59,0x91,0xf8,0x83,0x84,0xc4,
216     0x8c,0x38,0xc2,0x35,0x0e,0x7e,0x50,0x67,0x76,0xe7,0xd3,0xec,0x6f,0x0d,0xa0,0x5c,
217     0x2f,0x0a,0x80,0x28,0xd3,0xc5,0x7d,0x2d,0x1a,0x0b,0x96,0xd6,0xe5,0x98,0x05,0x8c,
218     0x4d,0xa0,0x1f,0x8c,0xb6,0xfb,0xb1,0xcf,0xe9,0xcb,0x38,0x27,0x60,0x64,0x17,0xca,
219     0xf4,0x8b,0x61,0xb7,0x1d,0xb6,0x20,0x9d,0x40,0x2a,0x1c,0xfd,0x55,0x40,0x4b,0x95,
220     0x39,0x52,0x18,0x3b,0xab,0x44,0xe8,0x83,0x4b,0x7c,0x47,0xfb,0xed,0x06,0x9c,0xcd,
221     0x4f,0xba,0x81,0xd6,0xb7,0x31,0xcf,0x5c,0x23,0xf8,0x25,0xab,0x95,0x77,0x0a,0x8f,
222     0x46,0xef,0xfb,0x59,0xb8,0x04,0xd7,0x1e,0xf5,0xaf,0x6a,0x1a,0x26,0x9b,0xae,0xf4,
223     0xf5,0x7f,0x84,0x6f,0x3c,0xed,0xf8,0x24,0x0b,0x43,0xd1,0xba,0x74,0x89,0x4e,0x39,
224     0xfe,0xab,0xa5,0x16,0xa5,0x28,0xee,0x96,0x84,0x3e,0x16,0x6d,0x5f,0x4e,0x0b,0x7d,
225     0x94,0x16,0x1b,0x8c,0xf9,0xaa,0x9b,0xc0,0x49,0x02,0x4c,0x3e,0x62,0xff,0xfe,0xa2,
226     0x20,0x33,0x5e,0xa6,0xdd,0xda,0x15,0x2d,0xb7,0xcd,0xda,0xff,0xb1,0x0b,0x45,0x7b,
227
    0xd3,0xa0,0x42,0x29,0xab,0xa9,0x73,0xe9,0xa4,0xd9,0x8d,0xac,0xa1,0x88,0x2c,0x2d}};
228
229 #endif // SHA256
230
231 #if ALG_SHA384_VALUE == DEFAULT_TEST_HASH
232
233 const TPM2B_RSA_TEST_VALUE    c_OaepKvt = {RSA_TEST_KEY_SIZE, {
234     0x0f,0x3c,0x42,0x4d,0x8c,0x91,0x96,0x05,0x3c,0xfd,0x59,0x3b,0x7f,0x29,0xbc,0x03,
235     0x67,0xc1,0xff,0x74,0xe7,0x09,0xf4,0x13,0x45,0xbe,0x13,0x1d,0xc9,0x86,0x94,0xfe,
```

```
236     0xed,0xa6,0xe8,0x3a,0xcb,0x89,0x4d,0xec,0x86,0x63,0x4c,0xdb,0xf1,0x95,0xee,0xc1,
237     0x46,0xc5,0x3b,0xd8,0xf8,0xa2,0x41,0x6a,0x60,0x8b,0x9e,0x5e,0x7f,0x20,0x16,0xe3,
238     0x69,0xb6,0x2d,0x92,0xfc,0x60,0xa2,0x74,0x88,0xd5,0xc7,0xa6,0xd1,0xff,0xe3,0x45,
239     0x02,0x51,0x39,0xd9,0xf3,0x56,0x0b,0x91,0x80,0xe0,0x6c,0xa8,0xc3,0x78,0xef,0x34,
240     0x22,0x8c,0xf5,0xfb,0x47,0x98,0x5d,0x57,0x8e,0x3a,0xb9,0xff,0x92,0x04,0xc7,0xc2,
241     0x6e,0xfa,0x14,0xc1,0xb9,0x68,0x15,0x5c,0x12,0xe8,0xa8,0xbe,0xea,0xe8,0x8d,0x9b,
242     0x48,0x28,0x35,0xdb,0x4b,0x52,0xc1,0x2d,0x85,0x47,0x83,0xd0,0xe9,0xae,0x90,0x6e,
243     0x65,0xd4,0x34,0x7f,0x81,0xce,0x69,0xf0,0x96,0x62,0xf7,0xec,0x41,0xd5,0xc2,0xe3,
244     0x4b,0xba,0x9c,0x8a,0x02,0xce,0xf0,0x5d,0x14,0xf7,0x09,0x42,0x8e,0x4a,0x27,0xfe,
245     0x3e,0x66,0x42,0x99,0x03,0xe1,0x69,0xbd,0xdb,0x7f,0x9b,0x70,0xeb,0x4e,0x9c,0xac,
246     0x45,0x67,0x91,0x9f,0x75,0x10,0xc6,0xfc,0x14,0xe1,0x28,0xc1,0x0e,0xe0,0x7e,0xc0,
247     0x5c,0x1d,0xee,0xe8,0xff,0x45,0x79,0x51,0x86,0x08,0xe6,0x39,0xac,0xb5,0xfd,0xb8,
248     0xf1,0xdd,0x2e,0xf4,0xb2,0x1a,0x69,0x0d,0xd9,0x98,0x8e,0xdb,0x85,0x61,0x70,0x20,
249
    0x82,0x91,0x26,0x87,0x80,0xc4,0x6a,0xd8,0x3b,0x91,0x4d,0xd3,0x33,0x84,0xad,0xb7}};
250
251   const TPM2B_RSA_TEST_VALUE    c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
252     0x44,0xd5,0x9f,0xbc,0x48,0x03,0x3d,0x9f,0x22,0x91,0x2a,0xab,0x3c,0x31,0x71,0xab,
253     0x86,0x3f,0x0f,0x6f,0x59,0x5b,0x93,0x27,0xbc,0xbc,0xcd,0x29,0x38,0x43,0x2a,0x3b,
254     0x3b,0xd2,0xb3,0x45,0x40,0xba,0x15,0xb4,0x45,0xe3,0x56,0xab,0xff,0xb3,0x20,0x26,
255     0x39,0xcc,0x48,0xc5,0x5d,0x41,0x0d,0x2f,0x57,0x7f,0x9d,0x16,0x2e,0x26,0x57,0xc7,
256     0x6b,0xf3,0x36,0x54,0xbd,0xb6,0x1d,0x46,0x4e,0x13,0x50,0xd7,0x61,0x9d,0x8d,0x7b,
257     0xeb,0x21,0x9f,0x79,0xf3,0xfd,0xe0,0x1b,0xa8,0xed,0x6d,0x29,0x33,0x0d,0x65,0x94,
258     0x24,0x1e,0x62,0x88,0x6b,0x2b,0x4e,0x39,0xf5,0x80,0x39,0xca,0x76,0x95,0xbc,0x7c,
259     0x27,0x1d,0xdd,0x3a,0x11,0xf1,0x3e,0x54,0x03,0xb7,0x43,0x91,0x99,0x33,0xfe,0x9d,
260     0x14,0x2c,0x87,0x9a,0x95,0x18,0x1f,0x02,0x04,0x6a,0xe2,0xb7,0x81,0x14,0x13,0x45,
261     0x16,0xfb,0xe4,0xb7,0x8f,0xab,0x2b,0xd7,0x60,0x34,0x8a,0x55,0xbc,0x01,0x8c,0x49,
262     0x02,0x29,0xf1,0x9c,0x94,0x98,0x44,0xd0,0x94,0xcb,0xd4,0x85,0x4c,0x3b,0x77,0x72,
263     0x99,0xd5,0x4b,0xc6,0x3b,0xe4,0xd2,0xc8,0xe9,0x6a,0x23,0x18,0x3b,0x3b,0x5e,0x32,
264     0xec,0x70,0x84,0x5d,0xbb,0x6a,0x8f,0x0c,0x5f,0x55,0xa5,0x30,0x34,0x48,0xbb,0xc2,
265     0xdf,0x12,0xb9,0x81,0xad,0x36,0x3f,0xf0,0x24,0x16,0x48,0x04,0x4a,0x7f,0xfd,0x9f,
266     0x4c,0xea,0xfe,0x1d,0x83,0xd0,0x81,0xad,0x25,0x6c,0x5f,0x45,0x36,0x91,0xf0,0xd5,
267
    0x8b,0x53,0x0a,0xdf,0xec,0x9f,0x04,0x58,0xc4,0x35,0xa0,0x78,0x1f,0x68,0xe0,0x22}};
268
269   const TPM2B_RSA_TEST_VALUE    c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
270     0x3f,0x3a,0x82,0x6d,0x42,0xe3,0x8b,0x4f,0x45,0x9c,0xda,0x6c,0xbe,0xbe,0xcd,0x00,
271     0x98,0xfb,0xbe,0x59,0x30,0xc6,0x3c,0xaa,0xb3,0x06,0x27,0xb5,0xda,0xfa,0xb2,0xc3,
272     0x43,0xb7,0xbd,0xe9,0xd3,0x23,0xed,0x80,0xce,0x74,0xb3,0xb8,0x77,0x8d,0xe6,0x8d,
273     0x3c,0xe5,0xf5,0xd7,0x80,0xcf,0x38,0x55,0x76,0xd7,0x87,0xa8,0xd6,0x3a,0xcf,0xfd,
274     0xd8,0x91,0x65,0xab,0x43,0x66,0x50,0xb7,0x9a,0x13,0x6b,0x45,0x80,0x76,0x86,0x22,
275     0x27,0x72,0xf7,0xbb,0x65,0x22,0x5c,0x55,0x60,0xd8,0x84,0x9f,0xf2,0x61,0x52,0xac,
276     0xf2,0x4f,0x5b,0x7b,0x21,0xe1,0xf5,0x4b,0x8f,0x01,0xf2,0x4b,0xcf,0xd3,0xfb,0x74,
277     0x5e,0x6e,0x96,0xb4,0xa8,0x0f,0x01,0x9b,0x26,0x54,0x0a,0x70,0x55,0x26,0xb7,0x0b,
278     0xe8,0x01,0x68,0x66,0x0d,0x6f,0xb5,0xfc,0x66,0xbd,0x9e,0x44,0xed,0x6a,0x1e,0x3c,
279     0x3b,0x61,0x5d,0xe8,0xdb,0x99,0x5b,0x67,0xbf,0x94,0xfb,0xe6,0x8c,0x4b,0x07,0xcb,
280     0x43,0x3a,0x0d,0xb1,0x1b,0x10,0x66,0x81,0xe2,0x0d,0xe7,0xd1,0xca,0x85,0xa7,0x50,
281     0x82,0x2d,0xbf,0xed,0xcf,0x43,0x6d,0xdb,0x2c,0x7b,0x73,0x20,0xfe,0x73,0x3f,0x19,
282     0xc6,0xdb,0x69,0xb8,0xc3,0xd3,0xf4,0xe5,0x64,0xf8,0x36,0x8e,0xd5,0xd8,0x09,0x2a,
283     0x5f,0x26,0x70,0xa1,0xd9,0x5b,0x14,0xf8,0x22,0xe9,0x9d,0x22,0x51,0xf4,0x52,0xc1,
284     0x6f,0x53,0xf5,0xca,0x0d,0xda,0x39,0x8c,0x29,0x42,0xe8,0x58,0x89,0xbb,0xd1,0x2e,
285
    0xc5,0xdb,0x86,0x8d,0xaf,0xec,0x58,0x36,0x8d,0x8d,0x57,0x23,0xd5,0xdd,0xb9,0x24}};
286
287   const TPM2B_RSA_TEST_VALUE    c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
288     0x39,0x10,0x58,0x7d,0x6d,0xa8,0xd5,0x90,0x07,0xd6,0x2b,0x13,0xe9,0xd8,0x93,0x7e,
289     0xf3,0x5d,0x71,0xe0,0xf0,0x33,0x3a,0x4a,0x22,0xf3,0xe6,0x95,0xd3,0x8e,0x8c,0x41,
290     0xe7,0xb3,0x13,0xde,0x4a,0x45,0xd3,0xd1,0xfb,0xb1,0x3f,0x9b,0x39,0xa5,0x50,0x58,
291     0xef,0xb6,0x3a,0x43,0xdd,0x54,0xab,0xda,0x9d,0x32,0x49,0xe4,0x57,0x96,0xe5,0x1b,
292     0x1d,0x8f,0x33,0x8e,0x07,0x67,0x56,0x14,0xc1,0x18,0x78,0xa2,0x52,0xe6,0x2e,0x07,
293     0x81,0xbe,0xd8,0xca,0x76,0x63,0x68,0xc5,0x47,0xa2,0x92,0x5e,0x4c,0xfd,0x14,0xc7,
294     0x46,0x14,0xbe,0xc7,0x85,0xef,0xe6,0xb8,0x46,0xcb,0x3a,0x67,0x66,0x89,0xc6,0xee,
295     0x9d,0x64,0xf5,0x0d,0x09,0x80,0x9a,0x6f,0x0e,0xeb,0xe4,0xb9,0xe9,0xab,0x90,0x4f,
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
296     0xe7,0x5a,0xc8,0xca,0xf6,0x16,0x0a,0x82,0xbd,0xb7,0x76,0x59,0x08,0x2d,0xd9,0x40,
297     0x5d,0xaa,0xa5,0xef,0xfb,0xe3,0x81,0x2c,0x2c,0x5c,0xa8,0x16,0xbd,0x63,0x20,0xc2,
298     0x4d,0x3b,0x51,0xaa,0x62,0x1f,0x06,0xe5,0xbb,0x78,0x44,0x04,0x0c,0x5c,0xe1,0x1b,
299     0x6b,0x9d,0x21,0x10,0xaf,0x48,0x48,0x98,0x97,0x77,0xc2,0x73,0xb4,0x98,0x64,0xcc,
300     0x94,0x2c,0x29,0x28,0x45,0x36,0xd1,0xc5,0xd0,0x2f,0x97,0x27,0x92,0x65,0x22,0xbb,
301     0x63,0x79,0xea,0xf5,0xff,0x77,0x0f,0x4b,0x56,0x8a,0x9f,0xad,0x1a,0x97,0x67,0x39,
302     0x69,0xb8,0x4c,0x6c,0xc2,0x56,0xc5,0x7a,0xa8,0x14,0x5a,0x24,0x7a,0xa4,0x6e,0x55,
303
    0xb2,0x86,0x1d,0xf4,0x62,0x5a,0x2d,0x87,0x6d,0xde,0x99,0x78,0x2d,0xef,0xd7,0xdc}};
304
305   #endif // SHA384
306
307   #if ALG_SHA512_VALUE == DEFAULT_TEST_HASH
308
309   const TPM2B_RSA_TEST_VALUE    c_OaepKvt = {RSA_TEST_KEY_SIZE, {
310     0x48,0x45,0xa7,0x70,0xb2,0x41,0xb7,0x48,0x5e,0x79,0x8c,0xdf,0x1c,0xc6,0x7e,0xbb,
311     0x11,0x80,0x82,0x52,0xbf,0x40,0x3d,0x90,0x03,0x6e,0x20,0x3a,0xb9,0x65,0xc8,0x51,
312     0x4c,0xbd,0x9c,0xa9,0x43,0x89,0xd0,0x57,0x0c,0xa3,0x69,0x22,0x7e,0x82,0x2a,0x1c,
313     0x1d,0x5a,0x80,0x84,0x81,0xbb,0x5e,0x5e,0xd0,0xc1,0x66,0x9a,0xac,0x00,0xba,0x14,
314     0xa2,0xe9,0xd0,0x3a,0x89,0x5a,0x63,0xe2,0xec,0x92,0x05,0xf4,0x47,0x66,0x12,0x7f,
315     0xdb,0xa7,0x3c,0x5b,0x67,0xe1,0x55,0xca,0x0a,0x27,0xbf,0x39,0x89,0x11,0x05,0xba,
316     0x9b,0x5a,0x9b,0x65,0x44,0xad,0x78,0xcf,0x8f,0x94,0xf6,0x9a,0xb4,0x52,0x39,0x0e,
317     0x00,0xba,0xbc,0xe0,0xbd,0x6f,0x81,0x2d,0x76,0x42,0x66,0x70,0x07,0x77,0xbf,0x09,
318     0x88,0x2a,0x0c,0xb1,0x56,0x3e,0xee,0xfd,0xdc,0xb6,0x3c,0x0d,0xc5,0xa4,0x0d,0x10,
319     0x32,0x80,0x3e,0x1e,0xfe,0x36,0x8f,0xb5,0x42,0xc1,0x21,0x7b,0xdf,0xdf,0x4a,0xd2,
320     0x68,0x0c,0x01,0x9f,0x4a,0xfd,0xd4,0xec,0xf7,0x49,0x06,0xab,0xed,0xc6,0xd5,0x1b,
321     0x63,0x76,0x38,0xc8,0x6c,0xc7,0x4f,0xcb,0x29,0x8a,0x0e,0x6f,0x33,0xaf,0x69,0x31,
322     0x8e,0xa7,0xdd,0x9a,0x36,0xde,0x9b,0xf1,0x0b,0xfb,0x20,0xa0,0x6d,0x33,0x31,0xc9,
323     0x9e,0xb4,0x2e,0xc5,0x40,0x0e,0x60,0x71,0x36,0x75,0x05,0xf9,0x37,0xe0,0xca,0x8e,
324     0x8f,0x56,0xe0,0xea,0x9b,0xeb,0x17,0xf3,0xca,0x40,0xc3,0x48,0x01,0xba,0xdc,0xc6,
325
    0x4b,0x2b,0x5b,0x7b,0x5c,0x81,0xa6,0xbb,0xc7,0x43,0xc0,0xbe,0xc0,0x30,0x7b,0x55}};
326
327   const TPM2B_RSA_TEST_VALUE    c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
328     0x74,0x83,0xfa,0x52,0x65,0x50,0x68,0xd0,0x82,0x05,0x72,0x70,0x78,0x1c,0xac,0x10,
329     0x23,0xc5,0x07,0xf8,0x93,0xd2,0xeb,0x65,0x87,0xbb,0x47,0xc2,0xfb,0x30,0x9e,0x61,
330     0x4c,0xac,0x04,0x57,0x5a,0x7c,0xeb,0x29,0x08,0x84,0x86,0x89,0x1e,0x8f,0x07,0x32,
331     0xa3,0x8b,0x70,0xe7,0xa2,0x9f,0x9c,0x42,0x71,0x3d,0x23,0x59,0x82,0x5e,0x8a,0xde,
332     0xd6,0xfb,0xd8,0xc5,0x8b,0xc0,0xdb,0x10,0x38,0x87,0xd3,0xbf,0x04,0xb0,0x66,0xb9,
333     0x85,0x81,0x54,0x4c,0x69,0xdc,0xba,0x78,0xf3,0x4a,0xdb,0x25,0xa2,0xf2,0x34,0x55,
334     0xdd,0xaa,0xa5,0xc4,0xed,0x55,0x06,0x0e,0x2a,0x30,0x77,0xab,0x82,0x79,0xf0,0xcd,
335     0x9d,0x6f,0x09,0xa0,0xc8,0x82,0xc9,0xe0,0x61,0xda,0x40,0xcd,0x17,0x59,0xc0,0xef,
336     0x95,0x6d,0xa3,0x6d,0x1c,0x2b,0xee,0x24,0xef,0xd8,0x4a,0x55,0x6c,0xd6,0x26,0x42,
337     0x32,0x17,0xfd,0x6a,0xb3,0x4f,0xde,0x07,0x2f,0x10,0xd4,0xac,0x14,0xea,0x89,0x68,
338     0xcc,0xd3,0x07,0xb7,0xcf,0xba,0x39,0x20,0x63,0x20,0x7b,0x44,0x8b,0x48,0x60,0x5d,
339     0x3a,0x2a,0x0a,0xe9,0x68,0xab,0x15,0x46,0x27,0x64,0xb5,0x82,0x06,0x29,0xe7,0x25,
340     0xca,0x46,0x48,0x6e,0x2a,0x34,0x57,0x4b,0x81,0x75,0xae,0xb6,0xfd,0x6f,0x51,0x5f,
341     0x04,0x59,0xc7,0x15,0x1f,0xe0,0x68,0xf7,0x36,0x2d,0xdf,0xc8,0x9d,0x05,0x27,0x2d,
342     0x3f,0x2b,0x59,0x5d,0xcb,0xf3,0xc4,0x92,0x6e,0x00,0xa8,0x8d,0xd0,0x69,0xe5,0x59,
343
    0xda,0xba,0x4f,0x38,0xf5,0xa0,0x8b,0xf1,0x73,0xe9,0x0d,0xee,0x64,0xe5,0xa2,0xd8}};
344
345   const TPM2B_RSA_TEST_VALUE    c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
346     0x1b,0xca,0x8b,0x18,0x15,0x3b,0x95,0x5b,0x0a,0x89,0x10,0x03,0x7f,0x7c,0xa0,0xc9,
347     0x66,0x57,0x86,0x6a,0xc9,0xeb,0x82,0x71,0xf3,0x8d,0x6f,0xa9,0xa4,0x2d,0xd0,0x22,
348     0xdf,0xe9,0xc6,0x71,0x5b,0xf4,0x27,0x38,0x5b,0x2c,0x8a,0x54,0xcc,0x85,0x11,0x69,
349     0x6d,0x6f,0x42,0xe7,0x22,0xcb,0xd6,0xad,0x1a,0xc5,0xab,0x6a,0xa5,0xfc,0xa5,0x70,
350     0x72,0x4a,0x62,0x25,0xd0,0xa2,0x16,0x61,0xab,0xac,0x31,0xa0,0x46,0x24,0x4f,0xdd,
351     0x9a,0x36,0x55,0xb6,0x00,0x9e,0x23,0x50,0x0d,0x53,0x01,0xb3,0x46,0x56,0xb2,0x1d,
352     0x33,0x5b,0xca,0x41,0x7f,0x65,0x7e,0x00,0x5c,0x12,0xff,0x0a,0x70,0x5d,0x8c,0x69,
353     0x4a,0x02,0xee,0x72,0x30,0xa7,0x5c,0xa4,0xbb,0xbe,0x03,0x0c,0xe4,0x5f,0x33,0xb6,
354     0x78,0x91,0x9d,0xd8,0xec,0x34,0x03,0x2e,0x63,0x32,0xc7,0x2a,0x36,0x50,0xd5,0x8b,
355     0x0e,0x7f,0x54,0x4e,0xf4,0x29,0x11,0x1b,0xcd,0x0f,0x37,0xa5,0xbc,0x61,0x83,0x50,
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
356       0xfa,0x18,0x75,0xd9,0xfe,0xa7,0xe8,0x9b,0xc1,0x4f,0x96,0x37,0x81,0x71,0xdf,0x71,
357       0x8b,0x89,0x81,0xf4,0x95,0xb5,0x29,0x66,0x41,0x0c,0x73,0xd7,0x0b,0x21,0xb4,0xfb,
358       0xf9,0x63,0x2f,0xe9,0x7b,0x38,0xaa,0x20,0xc3,0x96,0xcc,0xb7,0xb2,0x24,0xa1,0xe0,
359       0x59,0x9c,0x10,0x9e,0x5a,0xf7,0xe3,0x02,0xe6,0x23,0xe2,0x44,0x21,0x3f,0x6e,0x5e,
360       0x79,0xb2,0x93,0x7d,0xce,0xed,0xe2,0xe1,0xab,0x98,0x07,0xa7,0xbd,0xbc,0xd8,0xf7,
361
    0x06,0xeb,0xc5,0xa6,0x37,0x18,0x11,0x88,0xf7,0x63,0x39,0xb9,0x57,0x29,0xdc,0x03}};
362
363     const TPM2B_RSA_TEST_VALUE    c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
364       0x05,0x55,0x00,0x62,0x01,0xc6,0x04,0x31,0x55,0x73,0x3f,0x2a,0xf9,0xd4,0x0f,0xc1,
365       0x2b,0xeb,0xd8,0xc8,0xdb,0xb2,0xab,0x6c,0x26,0xde,0x2d,0x89,0xc2,0x2d,0x36,0x62,
366       0xc8,0x22,0x5d,0x58,0x03,0xb1,0x46,0x14,0xa5,0xd4,0xbc,0x25,0x6b,0x7f,0x8f,0x14,
367       0x7e,0x03,0x2f,0x3d,0xb8,0x39,0xa5,0x79,0x13,0x7e,0x22,0x2a,0xb9,0x3e,0x8f,0xaa,
368       0x01,0x7c,0x03,0x12,0x21,0x6c,0x2a,0xb4,0x39,0x98,0x6d,0xff,0x08,0x6c,0x59,0x2d,
369       0xdc,0xc6,0xf1,0x77,0x62,0x10,0xa6,0xcc,0xe2,0x71,0x8e,0x97,0x00,0x87,0x5b,0x0e,
370       0x20,0x00,0x3f,0x18,0x63,0x83,0xf0,0xe4,0x0a,0x64,0x8c,0xe9,0x8c,0x91,0xe7,0x89,
371       0x04,0x64,0x2c,0x8b,0x41,0xc8,0xac,0xf6,0x5a,0x75,0xe6,0xa5,0x76,0x43,0xcb,0xa5,
372       0x33,0x8b,0x07,0xc9,0x73,0x0f,0x45,0xa4,0xc3,0xac,0xc1,0xc3,0xe6,0xe7,0x21,0x66,
373       0x1c,0xba,0xbf,0xea,0x3e,0x39,0xfa,0xb2,0xe2,0x8f,0xfe,0x9c,0xb4,0x85,0x89,0x33,
374       0x2a,0x0c,0xc8,0x5d,0x58,0xe1,0x89,0x12,0xe9,0x4d,0x42,0xb3,0x1f,0x99,0x0c,0x3e,
375       0xd8,0xb2,0xeb,0xf5,0x88,0xfb,0xe1,0x4b,0x8e,0xdc,0xd3,0xa8,0xda,0xbe,0x04,0x45,
376       0xbf,0x56,0xc6,0x54,0x70,0x00,0xb8,0x66,0x46,0x3a,0xa3,0x1e,0xb6,0xeb,0x1a,0xa0,
377       0x0b,0xd3,0x9a,0x9a,0x52,0xda,0x60,0x69,0xb7,0xef,0x93,0x47,0x38,0xab,0x1a,0xa0,
378       0x22,0x6e,0x76,0x06,0xb6,0x74,0xaf,0x74,0x8f,0x51,0xc0,0x89,0x5a,0x4b,0xbe,0x6a,
379
    0x91,0x18,0x25,0x7d,0xa6,0x77,0xe6,0xfd,0xc2,0x62,0x36,0x07,0xc6,0xef,0x79,0xc9}};
380
381     #endif // SHA512
```

10.1.10 SelfTest.h

10.1.10.1 Introduction

This file contains the structure definitions for the self-test. It also contains macros for use when the self-test is implemented.

```
1     #ifndef                      _SELF_TEST_H_
2     #define                      _SELF_TEST_H_
```

10.1.10.2 Defines

Was typing this a lot

```
3     #define SELF_TEST_FAILURE   FAIL(FATAL_ERROR_SELF_TEST)
```

Use the definition of key sizes to set algorithm values for key size.

```
4     #define AES_ENTRIES (AES_128 + AES_192 + AES_256)
5     #define SM4_ENTRIES (SM4_128)
6     #define CAMELLIA_ENTRIES (CAMELLIA_128 + CAMELLIA_192 + CAMELLIA_256)
7     #define TDES_ENTRIES (TDES_128 + TDES_192)
8     #define NUM_SYMS     (AES_ENTRIES + SM4_ENTRIES + CAMELLIA_ENTRIES + TDES_ENTRIES)
9     typedef UINT32       SYM_INDEX;
```

These two defines deal with the fact that the TPM_ALG_ID table does not delimit the symmetric mode values with a TPM_SYM_MODE_FIRST and TPM_SYM_MODE_LAST

```
10    #define TPM_SYM_MODE_FIRST        ALG_CTR_VALUE
11    #define TPM_SYM_MODE_LAST         ALG_ECB_VALUE
```

 Copyright © TCG 2006-2020

```
12  #define NUM_SYM_MODES    (TPM_SYM_MODE_LAST - TPM_SYM_MODE_FIRST + 1)
```

Define a type to hold a bit vector for the modes.

```
13  #if NUM_SYM_MODES <= 0
14  #error  "No symmetric modes implemented"
15  #elif NUM_SYM_MODES <= 8
16  typedef BYTE    SYM_MODES;
17  #elif NUM_SYM_MODES <= 16
18  typedef UINT16  SYM_MODES;
19  #elif NUM_SYM_MODES <= 32
20  typedef UINT32  SYM_MODES;
21  #else
22  #error "Too many symmetric modes"
23  #endif
24  typedef struct SYMMETRIC_TEST_VECTOR {
25      const TPM_ALG_ID     alg;                  // the algorithm
26      const UINT16         keyBits;              // bits in the key
27      const BYTE          *key;                  // The test key
28      const UINT32         ivSize;               // block size of the algorithm
29      const UINT32         dataInOutSize;        // size  to encrypt/decrypt
30      const BYTE          *dataIn;               // data to encrypt
31      const BYTE          *dataOut[NUM_SYM_MODES];// data to decrypt
32  } SYMMETRIC_TEST_VECTOR;
33  #if ALG_SHA512
34  #       define  DEFAULT_TEST_HASH               ALG_SHA512_VALUE
35  #       define  DEFAULT_TEST_DIGEST_SIZE        SHA512_DIGEST_SIZE
36  #       define  DEFAULT_TEST_HASH_BLOCK_SIZE    SHA512_BLOCK_SIZE
37  #elif ALG_SHA384
38  #       define  DEFAULT_TEST_HASH               ALG_SHA384_VALUE
39  #       define  DEFAULT_TEST_DIGEST_SIZE        SHA384_DIGEST_SIZE
40  #       define  DEFAULT_TEST_HASH_BLOCK_SIZE    SHA384_BLOCK_SIZE
41  #elif ALG_SHA256
42  #       define  DEFAULT_TEST_HASH               ALG_SHA256_VALUE
43  #       define  DEFAULT_TEST_DIGEST_SIZE        SHA256_DIGEST_SIZE
44  #       define  DEFAULT_TEST_HASH_BLOCK_SIZE    SHA256_BLOCK_SIZE
45  #elif ALG_SHA1
46  #       define  DEFAULT_TEST_HASH               ALG_SHA1_VALUE
47  #       define  DEFAULT_TEST_DIGEST_SIZE        SHA1_DIGEST_SIZE
48  #       define  DEFAULT_TEST_HASH_BLOCK_SIZE    SHA1_BLOCK_SIZE
49  #endif
50  #endif      // _SELF_TEST_H_
```

10.1.11 SupportLibraryFunctionPrototypes_fp.h

10.1.11.1 Introduction

This file contains the function prototypes for the functions that need to be present in the selected math library. For each function listed, there should be a small stub function. That stub provides the interface between the TPM code and the support library. In most cases, the stub function will only need to do a format conversion between the TPM big number and the support library big number. The TPM big number format was chosen to make this relatively simple and fast.

Arithmetic operations return a BOOL to indicate if the operation completed successfully or not.

```
1  #ifndef SUPPORT_LIBRARY_FUNCTION_PROTOTYPES_H
2  #define SUPPORT_LIBRARY_FUNCTION_PROTOTYPES_H
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.1.11.2 SupportLibInit()

This function is called by CryptInit() so that necessary initializations can be performed on the cryptographic library.

```
3   LIB_EXPORT
4   int SupportLibInit(void);
```

10.1.11.3 MathLibraryCompatibililtyCheck()

This function is only used during development to make sure that the library that is being referenced is using the same size of data structures as the TPM.

```
5   BOOL
6   MathLibraryCompatibilityCheck(
7       void
8       );
```

10.1.11.4 BnModMult()

Does *op1* * *op2* and divide by *modulus* returning the remainder of the divide.

```
9   LIB_EXPORT BOOL
10  BnModMult(bigNum result, bigConst op1, bigConst op2, bigConst modulus);
```

10.1.11.5 BnMult()

Multiplies two numbers and returns the result

```
11  LIB_EXPORT BOOL
12  BnMult(bigNum result, bigConst multiplicand, bigConst multiplier);
```

10.1.11.6 BnDiv()

This function divides two *bigNum* values. The function returns FALSE if there is an error in the operation.

```
13  LIB_EXPORT BOOL
14  BnDiv(bigNum quotient, bigNum remainder,
15                        bigConst dividend, bigConst divisor);
```

10.1.11.7 BnMod()

```
16  #define BnMod(a, b)     BnDiv(NULL, (a), (a), (b))
```

10.1.11.8 BnGcd()

Get the greatest common divisor of two numbers. This function is only needed when the TPM implements RSA.

```
17  LIB_EXPORT BOOL
18  BnGcd(bigNum gcd, bigConst number1, bigConst number2);
```

 Copyright © TCG 2006-2020

10.1.11.9 BnModExp()

Do modular exponentiation using *bigNum* values. This function is only needed when the TPM implements RSA.

```
19 LIB_EXPORT BOOL
20 BnModExp(bigNum result, bigConst number,
21                              bigConst exponent, bigConst modulus);
```

10.1.11.10 BnModInverse()

Modular multiplicative inverse. This function is only needed when the TPM implements RSA.

```
22 LIB_EXPORT BOOL BnModInverse(bigNum result, bigConst number,
23                              bigConst modulus);
```

10.1.11.11 BnEccModMult()

This function does a point multiply of the form R = [d]S. A return of FALSE indicates that the result was the point at infinity. This function is only needed if the TPM supports ECC.

```
24 LIB_EXPORT BOOL
25 BnEccModMult(bigPoint R, pointConst S, bigConst d, bigCurve E);
```

10.1.11.12 BnEccModMult2()

This function does a point multiply of the form R = [d]S + [u]Q. A return of FALSE indicates that the result was the point at infinity. This function is only needed if the TPM supports ECC.

```
26 LIB_EXPORT BOOL
27 BnEccModMult2(bigPoint R, pointConst S, bigConst d,
28                              pointConst Q, bigConst u, bigCurve E);
```

10.1.11.13 BnEccAdd()

This function does a point add R = S + Q. A return of FALSE indicates that the result was the point at infinity. This function is only needed if the TPM supports ECC.

```
29 LIB_EXPORT BOOL
30 BnEccAdd(bigPoint R, pointConst S, pointConst Q, bigCurve E);
```

10.1.11.14 BnCurveInitialize()

This function is used to initialize the pointers of a *bnCurve_t* structure. The structure is a set of pointers to *bigNum* values. The curve-dependent values are set by a different function. This function is only needed if the TPM supports ECC.

```
31 LIB_EXPORT bigCurve
32 BnCurveInitialize(bigCurve E, TPM_ECC_CURVE curveId);
```

10.1.11.14.1 BnCurveFree()

This function will free the allocated components of the curve and end the frame in which the curve data exists

```
33 LIB_EXPORT void
34 BnCurveFree(bigCurve E);
35 #endif
```

10.1.12 SymmetricTestData.h

This is a vector for testing either encrypt or decrypt. The premise for decrypt is that the IV for decryption is the same as the IV for encryption. However, the *ivOut* value may be different for encryption and decryption. We will encrypt at least two blocks. This means that the chaining value will be used for each of the schemes (if any) and that implicitly checks that the chaining value is handled properly.

```
1   #if AES_128
2   const BYTE  key_AES128 [] = {
3                   0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
4                   0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
5
6   const BYTE  dataIn_AES128 [] = {
7                   0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
8                   0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
9                   0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
10                  0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};
11
12  const BYTE  dataOut_AES128_ECB [] = {
13                  0x3a, 0xd7, 0x7b, 0xb4, 0x0d, 0x7a, 0x36, 0x60,
14                  0xa8, 0x9e, 0xca, 0xf3, 0x24, 0x66, 0xef, 0x97,
15                  0xf5, 0xd3, 0xd5, 0x85, 0x03, 0xb9, 0x69, 0x9d,
16                  0xe7, 0x85, 0x89, 0x5a, 0x96, 0xfd, 0xba, 0xaf};
17
18  const BYTE  dataOut_AES128_CBC [] = {
19                  0x76, 0x49, 0xab, 0xac, 0x81, 0x19, 0xb2, 0x46,
20                  0xce, 0xe9, 0x8e, 0x9b, 0x12, 0xe9, 0x19, 0x7d,
21                  0x50, 0x86, 0xcb, 0x9b, 0x50, 0x72, 0x19, 0xee,
22                  0x95, 0xdb, 0x11, 0x3a, 0x91, 0x76, 0x78, 0xb2};
23
24  const BYTE  dataOut_AES128_CFB [] = {
25                  0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
26                  0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
27                  0xc8, 0xa6, 0x45, 0x37, 0xa0, 0xb3, 0xa9, 0x3f,
28                  0xcd, 0xe3, 0xcd, 0xad, 0x9f, 0x1c, 0xe5, 0x8b};
29
30  const BYTE  dataOut_AES128_OFB [] = {
31                  0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
32                  0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
33                  0x77, 0x89, 0x50, 0x8d, 0x16, 0x91, 0x8f, 0x03,
34                  0xf5, 0x3c, 0x52, 0xda, 0xc5, 0x4e, 0xd8, 0x25};
35
36  const BYTE  dataOut_AES128_CTR [] = {
37                  0x87, 0x4d, 0x61, 0x91, 0xb6, 0x20, 0xe3, 0x26,
38                  0x1b, 0xef, 0x68, 0x64, 0x99, 0x0d, 0xb6, 0xce,
39                  0x98, 0x06, 0xf6, 0x6b, 0x79, 0x70, 0xfd, 0xff,
40                  0x86, 0x17, 0x18, 0x7b, 0xb9, 0xff, 0xfd, 0xff};
41  #endif
42
43  #if AES_192
44
45  const BYTE  key_AES192 [] = {
46                  0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52,
47                  0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,
48                  0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b};
49
50  const BYTE  dataIn_AES192 [] = {
51                  0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
52                  0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
53                  0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
54                  0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};
55
```


 Copyright © TCG 2006-2020

```
56    const BYTE  dataOut_AES192_ECB [] = {
57                    0xbd, 0x33, 0x4f, 0x1d, 0x6e, 0x45, 0xf2, 0x5f,
58                    0xf7, 0x12, 0xa2, 0x14, 0x57, 0x1f, 0xa5, 0xcc,
59                    0x97, 0x41, 0x04, 0x84, 0x6d, 0x0a, 0xd3, 0xad,
60                    0x77, 0x34, 0xec, 0xb3, 0xec, 0xee, 0x4e, 0xef};
61
62    const BYTE  dataOut_AES192_CBC [] = {
63                    0x4f, 0x02, 0x1d, 0xb2, 0x43, 0xbc, 0x63, 0x3d,
64                    0x71, 0x78, 0x18, 0x3a, 0x9f, 0xa0, 0x71, 0xe8,
65                    0xb4, 0xd9, 0xad, 0xa9, 0xad, 0x7d, 0xed, 0xf4,
66                    0xe5, 0xe7, 0x38, 0x76, 0x3f, 0x69, 0x14, 0x5a};
67
68    const BYTE  dataOut_AES192_CFB [] = {
69                    0xcd, 0xc8, 0x0d, 0x6f, 0xdd, 0xf1, 0x8c, 0xab,
70                    0x34, 0xc2, 0x59, 0x09, 0xc9, 0x9a, 0x41, 0x74,
71                    0x67, 0xce, 0x7f, 0x7f, 0x81, 0x17, 0x36, 0x21,
72                    0x96, 0x1a, 0x2b, 0x70, 0x17, 0x1d, 0x3d, 0x7a};
73
74    const BYTE  dataOut_AES192_OFB [] = {
75                    0xcd, 0xc8, 0x0d, 0x6f, 0xdd, 0xf1, 0x8c, 0xab,
76                    0x34, 0xc2, 0x59, 0x09, 0xc9, 0x9a, 0x41, 0x74,
77                    0xfc, 0xc2, 0x8b, 0x8d, 0x4c, 0x63, 0x83, 0x7c,
78                    0x09, 0xe8, 0x17, 0x00, 0xc1, 0x10, 0x04, 0x01};
79
80    const BYTE  dataOut_AES192_CTR [] = {
81                    0x1a, 0xbc, 0x93, 0x24, 0x17, 0x52, 0x1c, 0xa2,
82                    0x4f, 0x2b, 0x04, 0x59, 0xfe, 0x7e, 0x6e, 0x0b,
83                    0x09, 0x03, 0x39, 0xec, 0x0a, 0xa6, 0xfa, 0xef,
84                    0xd5, 0xcc, 0xc2, 0xc6, 0xf4, 0xce, 0x8e, 0x94};
85    #endif
86
87    #if AES_256
88
89    const BYTE  key_AES256 [] = {
90                    0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
91                    0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
92                    0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
93                    0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4};
94
95    const BYTE  dataIn_AES256 [] = {
96                    0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
97                    0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
98                    0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
99                    0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};
100
101   const BYTE  dataOut_AES256_ECB [] = {
102                   0xf3, 0xee, 0xd1, 0xbd, 0xb5, 0xd2, 0xa0, 0x3c,
103                   0x06, 0x4b, 0x5a, 0x7e, 0x3d, 0xb1, 0x81, 0xf8,
104                   0x59, 0x1c, 0xcb, 0x10, 0xd4, 0x10, 0xed, 0x26,
105                   0xdc, 0x5b, 0xa7, 0x4a, 0x31, 0x36, 0x28, 0x70};
106
107   const BYTE  dataOut_AES256_CBC [] = {
108                   0xf5, 0x8c, 0x4c, 0x04, 0xd6, 0xe5, 0xf1, 0xba,
109                   0x77, 0x9e, 0xab, 0xfb, 0x5f, 0x7b, 0xfb, 0xd6,
110                   0x9c, 0xfc, 0x4e, 0x96, 0x7e, 0xdb, 0x80, 0x8d,
111                   0x67, 0x9f, 0x77, 0x7b, 0xc6, 0x70, 0x2c, 0x7d};
112
113   const BYTE  dataOut_AES256_CFB [] = {
114                   0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x79, 0x16, 0x4b,
115                   0x7e, 0xcd, 0x84, 0x86, 0x98, 0x5d, 0x38, 0x60,
116                   0x39, 0xff, 0xed, 0x14, 0x3b, 0x28, 0xb1, 0xc8,
117                   0x32, 0x11, 0x3c, 0x63, 0x31, 0xe5, 0x40, 0x7b};
118
```

 Copyright © TCG 2006-2020

```
119     const BYTE  dataOut_AES256_OFB [] = {
120                     0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x79, 0x16, 0x4b,
121                     0x7e, 0xcd, 0x84, 0x86, 0x98, 0x5d, 0x38, 0x60,
122                     0x4f, 0xeb, 0xdc, 0x67, 0x40, 0xd2, 0x0b, 0x3a,
123                     0xc8, 0x8f, 0x6a, 0xd8, 0x2a, 0x4f, 0xb0, 0x8d};
124
125     const BYTE  dataOut_AES256_CTR [] = {
126                     0x60, 0x1e, 0xc3, 0x13, 0x77, 0x57, 0x89, 0xa5,
127                     0xb7, 0xa7, 0xf5, 0x04, 0xbb, 0xf3, 0xd2, 0x28,
128                     0xf4, 0x43, 0xe3, 0xca, 0x4d, 0x62, 0xb5, 0x9a,
129                     0xca, 0x84, 0xe9, 0x90, 0xca, 0xca, 0xf5, 0xc5};
130     #endif
```

10.1.13 SymmetricTest.h

10.1.13.1 Introduction

This file contains the structures and data definitions for the symmetric tests. This file references the header file that contains the actual test vectors. This organization was chosen so that the program that is used to generate the test vector values does not have to also re-generate this data.

```
1   #ifndef     SELF_TEST_DATA
2   #error  "This file may only be included in AlgorithmTests.c"
3   #endif
4   #ifndef     _SYMMETRIC_TEST_H
5   #define     _SYMMETRIC_TEST_H
6   #include    "SymmetricTestData.h"
```

10.1.13.2 Symmetric Test Structures

```
7   const SYMMETRIC_TEST_VECTOR   c_symTestValues[NUM_SYMS + 1] = {
8   #if ALG_AES && AES_128
9       {ALG_AES_VALUE, 128, key_AES128, 16, sizeof(dataIn_AES128), dataIn_AES128,
10      {dataOut_AES128_CTR, dataOut_AES128_OFB, dataOut_AES128_CBC,
11       dataOut_AES128_CFB, dataOut_AES128_ECB}},
12  #endif
13  #if ALG_AES && AES_192
14      {ALG_AES_VALUE, 192, key_AES192, 16, sizeof(dataIn_AES192), dataIn_AES192,
15      {dataOut_AES192_CTR, dataOut_AES192_OFB, dataOut_AES192_CBC,
16       dataOut_AES192_CFB, dataOut_AES192_ECB}},
17  #endif
18  #if ALG_AES && AES_256
19      {ALG_AES_VALUE, 256, key_AES256, 16, sizeof(dataIn_AES256), dataIn_AES256,
20      {dataOut_AES256_CTR, dataOut_AES256_OFB, dataOut_AES256_CBC,
21      dataOut_AES256_CFB, dataOut_AES256_ECB}},
22  #endif
23  // There are no SM4 test values yet so...
24  #if ALG_SM4 && SM4_128 && 0
25      {ALG_SM4_VALUE, 128, key_SM4128, 16, sizeof(dataIn_SM4128), dataIn_SM4128,
26      {dataOut_SM4128_CTR, dataOut_SM4128_OFB, dataOut_SM4128_CBC,
27       dataOut_SM4128_CFB, dataOut_AES128_ECB}},
28  #endif
29      {0}
30  };
31  #endif  // _SYMMETRIC_TEST_H
```

 Copyright © TCG 2006-2020

10.1.14 EccTestData.h

This file contains the parameter data for ECC testing.

```
1   #ifdef SELF_TEST_DATA
2   TPM2B_TYPE(EC_TEST, 32);
3   const TPM_ECC_CURVE         c_testCurve = 00003;
4
5   // The "static" key
6
7   const TPM2B_EC_TEST     c_ecTestKey_ds = {{32, {
8    0xdf,0x8d,0xa4,0xa3,0x88,0xf6,0x76,0x96,0x89,0xfc,0x2f,0x2d,0xa1,0xb4,0x39,0x7a,
9
    0x78,0xc4,0x7f,0x71,0x8c,0xa6,0x91,0x85,0xc0,0xbf,0xf3,0x54,0x20,0x91,0x2f,0x73}}}
    ;
10
11  const TPM2B_EC_TEST     c_ecTestKey_QsX = {{32, {
12   0x17,0xad,0x2f,0xcb,0x18,0xd4,0xdb,0x3f,0x2c,0x53,0x13,0x82,0x42,0x97,0xff,0x8d,
13
    0x99,0x50,0x16,0x02,0x35,0xa7,0x06,0xae,0x1f,0xda,0xe2,0x9c,0x12,0x77,0xc0,0xf9}}}
    ;
14
15  const TPM2B_EC_TEST     c_ecTestKey_QsY = {{32, {
16   0xa6,0xca,0xf2,0x18,0x45,0x96,0x6e,0x58,0xe6,0x72,0x34,0x12,0x89,0xcd,0xaa,0xad,
17
    0xcb,0x68,0xb2,0x51,0xdc,0x5e,0xd1,0x6d,0x38,0x20,0x35,0x57,0xb2,0xfd,0xc7,0x52}}}
    ;
18
19  // The "ephemeral" key
20
21  const TPM2B_EC_TEST     c_ecTestKey_de = {{32, {
22   0xb6,0xb5,0x33,0x5c,0xd1,0xee,0x52,0x07,0x99,0xea,0x2e,0x8f,0x8b,0x19,0x18,0x07,
23
    0xc1,0xf8,0xdf,0xdd,0xb8,0x77,0x00,0xc7,0xd6,0x53,0x21,0xed,0x02,0x53,0xee,0xac}}}
    ;
24
25  const TPM2B_EC_TEST     c_ecTestKey_QeX = {{32, {
26   0xa5,0x1e,0x80,0xd1,0x76,0x3e,0x8b,0x96,0xce,0xcc,0x21,0x82,0xc9,0xa2,0xa2,0xed,
27
    0x47,0x21,0x89,0x53,0x44,0xe9,0xc7,0x92,0xe7,0x31,0x48,0x38,0xe6,0xea,0x93,0x47}}}
    ;
28
29  const TPM2B_EC_TEST     c_ecTestKey_QeY = {{32, {
30   0x30,0xe6,0x4f,0x97,0x03,0xa1,0xcb,0x3b,0x32,0x2a,0x70,0x39,0x94,0xeb,0x4e,0xea,
31
    0x55,0x88,0x81,0x3f,0xb5,0x00,0xb8,0x54,0x25,0xab,0xd4,0xda,0xfd,0x53,0x7a,0x18}}}
    ;
32
33  // ECDH test results
34  const TPM2B_EC_TEST     c_ecTestEcdh_X = {{32, {
35   0x64,0x02,0x68,0x92,0x78,0xdb,0x33,0x52,0xed,0x3b,0xfa,0x3b,0x74,0xa3,0x3d,0x2c,
36
    0x2f,0x9c,0x59,0x03,0x07,0xf8,0x22,0x90,0xed,0xe3,0x45,0xf8,0x2a,0x0a,0xd8,0x1d}}}
    ;
37
38  const TPM2B_EC_TEST     c_ecTestEcdh_Y = {{32, {
39   0x58,0x94,0x05,0x82,0xbe,0x5f,0x33,0x02,0x25,0x90,0x3a,0x33,0x90,0x89,0xe3,0xe5,
40
    0x10,0x4a,0xbc,0x78,0xa5,0xc5,0x07,0x64,0xaf,0x91,0xbc,0xe6,0xff,0x85,0x11,0x40}}}
    ;
41
42  TPM2B_TYPE(TEST_VALUE, 64);
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
43  const TPM2B_TEST_VALUE        c_ecTestValue = {{64, {
44   0x78,0xd5,0xd4,0x56,0x43,0x61,0xdb,0x97,0xa4,0x32,0xc4,0x0b,0x06,0xa9,0xa8,0xa0,
45   0xf4,0x45,0x7f,0x13,0xd8,0x13,0x81,0x0b,0xe5,0x76,0xbe,0xaa,0xb6,0x3f,0x8d,0x4d,
46   0x23,0x65,0xcc,0xa7,0xc9,0x19,0x10,0xce,0x69,0xcb,0x0c,0xc7,0x11,0x8d,0xc3,0xff,
47
    0x62,0x69,0xa2,0xbe,0x46,0x90,0xe7,0x7d,0x81,0x77,0x94,0x65,0x1c,0x3e,0xc1,0x3e}}}
    ;
48
49  #if ALG_SHA1_VALUE == DEFAULT_TEST_HASH
50
51  const TPM2B_EC_TEST     c_TestEcDsa_r = {{32, {
52   0x57,0xf3,0x36,0xb7,0xec,0xc2,0xdd,0x76,0x0e,0xe2,0x81,0x21,0x49,0xc5,0x66,0x11,
53
    0x4b,0x8a,0x4f,0x17,0x62,0x82,0xcc,0x06,0xf6,0x64,0x78,0xef,0x6b,0x7c,0xf2,0x6c}}}
    ;
54  const TPM2B_EC_TEST     c_TestEcDsa_s = {{32, {
55   0x1b,0xed,0x23,0x72,0x8f,0x17,0x5f,0x47,0x2e,0xa7,0x97,0x2c,0x51,0x57,0x20,0x70,
56
    0x6f,0x89,0x74,0x8a,0xa8,0xf4,0x26,0xf4,0x96,0xa1,0xb8,0x3e,0xe5,0x35,0xc5,0x94}}}
    ;
57
58  const TPM2B_EC_TEST     c_TestEcSchnorr_r = {{32,{
59   0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x1b,0x08,0x9f,0xde,
60
    0xef,0x62,0xe3,0xf1,0x14,0xcb,0x54,0x28,0x13,0x76,0xfc,0x6d,0x69,0x22,0xb5,0x3e}}}
    ;
61  const TPM2B_EC_TEST     c_TestEcSchnorr_s = {{32,{
62   0xd9,0xd3,0x20,0xfb,0x4d,0x16,0xf2,0xe6,0xe2,0x45,0x07,0x45,0x1c,0x92,0x92,0x92,
63
    0xa9,0x6b,0x48,0xf8,0xd1,0x98,0x29,0x4d,0xd3,0x8f,0x56,0xf2,0xbb,0x2e,0x22,0x3b}}}
    ;
64
65  #endif // SHA1
66
67  #if ALG_SHA256_VALUE == DEFAULT_TEST_HASH
68
69  const TPM2B_EC_TEST     c_TestEcDsa_r = {{32, {
70   0x04,0x7d,0x54,0xeb,0x04,0x6f,0x56,0xec,0xa2,0x6c,0x38,0x8c,0xeb,0x43,0x0b,0x71,
71
    0xf8,0xf2,0xf4,0xa5,0xe0,0x1d,0x3c,0xa2,0x39,0x31,0xe4,0xe7,0x36,0x3b,0xb5,0x5f}}}
    ;
72  const TPM2B_EC_TEST     c_TestEcDsa_s = {{32, {
73   0x8f,0xd0,0x12,0xd9,0x24,0x75,0xf6,0xc4,0x3b,0xb5,0x46,0x75,0x3a,0x41,0x8d,0x80,
74
    0x23,0x99,0x38,0xd7,0xe2,0x40,0xca,0x9a,0x19,0x2a,0xfc,0x54,0x75,0xd3,0x4a,0x6e}}}
    ;
75
76  const TPM2B_EC_TEST     c_TestEcSchnorr_r = {{32, {
77   0xf7,0xb9,0x15,0x4c,0x34,0xf6,0x41,0x19,0xa3,0xd2,0xf1,0xbd,0xf4,0x13,0x6a,0x4f,
78
    0x63,0xb8,0x4d,0xb5,0xc8,0xcd,0xde,0x85,0x95,0xa5,0x39,0x0a,0x14,0x49,0x3d,0x2f}}}
    ;
79  const TPM2B_EC_TEST     c_TestEcSchnorr_s = {{32,{
80   0xfe,0xbe,0x17,0xaa,0x31,0x22,0x9f,0xd0,0xd2,0xf5,0x25,0x04,0x92,0xb0,0xaa,0x4e,
81
    0xcc,0x1c,0xb6,0x79,0xd6,0x42,0xb3,0x4e,0x3f,0xbb,0xfe,0x5f,0xd0,0xd0,0x8b,0xc3}}}
    ;
82
83  #endif // SHA256
84
85  #if ALG_SHA384_VALUE == DEFAULT_TEST_HASH
86
87  const TPM2B_EC_TEST     c_TestEcDsa_r = {{32, {
```

```
88      0xf5,0x74,0x6d,0xd6,0xc6,0x56,0x86,0xbb,0xba,0x1c,0xba,0x75,0x65,0xee,0x64,0x31,
89
     0xce,0x04,0xe3,0x9f,0x24,0x3f,0xbd,0xfe,0x04,0xcd,0xab,0x7e,0xfe,0xad,0xcb,0x82}}}
     ;
90   const TPM2B_EC_TEST     c_TestEcDsa_s = {{32, {
91      0xc2,0x4f,0x32,0xa1,0x06,0xc0,0x85,0x4f,0xc6,0xd8,0x31,0x66,0x91,0x9f,0x79,0xcd,
92
     0x5b,0xe5,0x7b,0x94,0xa1,0x91,0x38,0xac,0xd4,0x20,0xa2,0x10,0xf0,0xd5,0x9d,0xbf}}}
     ;
93
94   const TPM2B_EC_TEST     c_TestEcSchnorr_r = {{32, {
95      0x1e,0xb8,0xe1,0xbf,0xa1,0x9e,0x39,0x1e,0x58,0xa2,0xe6,0x59,0xd0,0x1a,0x6a,0x03,
96
     0x6a,0x1f,0x1c,0x4f,0x36,0x19,0xc1,0xec,0x30,0xa4,0x85,0x1b,0xe9,0x74,0x35,0x66}}}
     ;
97   const TPM2B_EC_TEST     c_TestEcSchnorr_s = {{32,{
98      0xb9,0xe6,0xe3,0x7e,0xcb,0xb9,0xea,0xf1,0xcc,0xf4,0x48,0x44,0x4a,0xda,0xc8,0xd7,
99
     0x87,0xb4,0xba,0x40,0xfe,0x5b,0x68,0x11,0x14,0xcf,0xa0,0x0e,0x85,0x46,0x99,0x01}}}
     ;
100
101  #endif // SHA384
102
103  #if ALG_SHA512_VALUE == DEFAULT_TEST_HASH
104
105  const TPM2B_EC_TEST     c_TestEcDsa_r = {{32, {
106     0xc9,0x71,0xa6,0xb4,0xaf,0x46,0x26,0x8c,0x27,0x00,0x06,0x3b,0x00,0x0f,0xa3,0x17,
107
     0x72,0x48,0x40,0x49,0x4d,0x51,0x4f,0xa4,0xcb,0x7e,0x86,0xe9,0xe7,0xb4,0x79,0xb2}}}
     ;
108  const TPM2B_EC_TEST     c_TestEcDsa_s = {{32,{
109     0x87,0xbc,0xc0,0xed,0x74,0x60,0x9e,0xfa,0x4e,0xe8,0x16,0xf3,0xf9,0x6b,0x26,0x07,
110
     0x3c,0x74,0x31,0x7e,0xf0,0x62,0x46,0xdc,0xd6,0x45,0x22,0x47,0x3e,0x0c,0xa0,0x02}}}
     ;
111
112  const TPM2B_EC_TEST     c_TestEcSchnorr_r = {{32,{
113     0xcc,0x07,0xad,0x65,0x91,0xdd,0xa0,0x10,0x23,0xae,0x53,0xec,0xdf,0xf1,0x50,0x90,
114
     0x16,0x96,0xf4,0x45,0x09,0x73,0x9c,0x84,0xb5,0x5c,0x5f,0x08,0x51,0xcb,0x60,0x01}}}
     ;
115  const TPM2B_EC_TEST     c_TestEcSchnorr_s = {{32,{
116     0x55,0x20,0x21,0x54,0xe2,0x49,0x07,0x47,0x71,0xf4,0x99,0x15,0x54,0xf3,0xab,0x14,
117
     0xdb,0x8e,0xda,0x79,0xb6,0x02,0x0e,0xe3,0x5e,0x6f,0x2c,0xb6,0x05,0xbd,0x14,0x10}}}
     ;
118
119  #endif // SHA512
120
121  #endif // SELF_TEST_DATA
```

10.1.15 CryptSym.h

10.1.15.1 Introduction

This file contains the implementation of the symmetric block cipher modes allowed for a TPM. These functions only use the single block encryption functions of the selected symmetric cryptographic library.

10.1.15.2 Includes, Defines, and Typedefs

```
1   #ifndef CRYPT_SYM_H
2   #define CRYPT_SYM_H
3   typedef union tpmCryptKeySchedule_t {
4   #if ALG_AES
5       tpmKeyScheduleAES           AES;
6   #endif
7   #if ALG_SM4
8       tpmKeyScheduleSM4           SM4;
9   #endif
10  #if ALG_CAMELLIA
11      tpmKeyScheduleCAMELLIA      CAMELLIA;
12  #endif
13
14  #if ALG_TDES
15      tpmKeyScheduleTDES          TDES[3];
16  #endif
17  #if SYMMETRIC_ALIGNMENT == 8
18      uint64_t            alignment;
19  #else
20      uint32_t            alignment;
21  #endif
22  } tpmCryptKeySchedule_t;
```

Each block cipher within a library is expected to conform to the same calling conventions with three parameters (*keySchedule*, *in*, and *out*) in the same order. That means that all algorithms would use the same order of the same parameters. The code is written assuming the (*keySchedule*, *in*, and *out*) order. However, if the library uses a different order, the order can be changed with a SWIZZLE macro that puts the parameters in the correct order. Note that all algorithms have to use the same order and number of parameters because the code to build the calling list is common for each call to encrypt or decrypt with the algorithm chosen by setting a function pointer to select the algorithm that is used.

```
23  #   define ENCRYPT(keySchedule, in, out)                                  \
24        encrypt(SWIZZLE(keySchedule, in, out))
25  #   define DECRYPT(keySchedule, in, out)                                  \
26        decrypt(SWIZZLE(keySchedule, in, out))
```

Note that the macros rely on *encrypt* as local values in the functions that use these macros. Those parameters are set by the macro that set the key schedule to be used for the call.

```
27  #define ENCRYPT_CASE(ALG)                                                \
28      case TPM_ALG_##ALG:                                                  \
29          TpmCryptSetEncryptKey##ALG(key, keySizeInBits, &keySchedule.ALG); \
30          encrypt = (TpmCryptSetSymKeyCall_t)TpmCryptEncrypt##ALG;         \
31          break;
32  #define DECRYPT_CASE(ALG)                                                \
33      case TPM_ALG_##ALG:                                                  \
34          TpmCryptSetDecryptKey##ALG(key, keySizeInBits, &keySchedule.ALG); \
35          decrypt = (TpmCryptSetSymKeyCall_t)TpmCryptDecrypt##ALG;         \
36          break;
37  #if ALG_AES
38  #define ENCRYPT_CASE_AES    ENCRYPT_CASE(AES)
```

```
39  #define ENCRYPT_CASE_AES     DECRYPT_CASE(AES)
40  #else
41  #define ENCRYPT_CASE_AES
42  #define DECRYPT_CASE_AES
43  #endif
44  #if ALG_SM4
45  #define ENCRYPT_CASE_SM4     ENCRYPT_CASE(SM4)
46  #define DECRYPT_CASE_SM4     DECRYPT_CASE(SM4)
47  #else
48  #define ENCRYPT_CASE_SM4
49  #define DECRYPT_CASE_SM4
50  #endif
51  #if ALG_CAMELLIA
52  #define ENCRYPT_CASE_CAMELLIA     ENCRYPT_CASE(CAMELLIA)
53  #define DECRYPT_CASE_CAMELLIA     DECRYPT_CASE(CAMELLIA)
54  #else
55  #define ENCRYPT_CASE_CAMELLIA
56  #define DECRYPT_CASE_CAMELLIA
57  #endif
58  #if ALG_TDES
59  #define ENCRYPT_CASE_TDES     ENCRYPT_CASE(TDES)
60  #define DECRYPT_CASE_TDES     DECRYPT_CASE(TDES)
61  #else
62  #define ENCRYPT_CASE_TDES
63  #define DECRYPT_CASE_TDES
64  #endif
```

For each algorithm the case will either be defined or null.

```
65  #define     SELECT(direction)                                    \
66      switch(algorithm)                                            \
67      {                                                            \
68          direction##_CASE_AES                                     \
69          direction##_CASE_SM4                                     \
70          direction##_CASE_CAMELLIA                                \
71          direction##_CASE_TDES                                    \
72          default:                                                 \
73              FAIL(FATAL_ERROR_INTERNAL);                          \
74      }
75  #endif // CRYPT_SYM_H
```

10.1.16 OIDs.h

```
1  #ifndef _OIDS_H_
2  #define _OIDS_H_
```

All the OIDs in this file are defined as DER-encoded values with a leading tag 0x06 (ASN1_OBJECT_IDENTIFIER), followed by a single length byte. This allows the OID size to be determined by looking at octet[1] of the OID (total size is OID[1] + 2). These macros allow OIDs to be defined (or not) depending on whether the associated hash algorithm is implemented.

NOTE: When one of these macros is used, the NAME needs '_' on each side. The exception is when the macro is used for the hash OID when only a single _ is used.

```
3  #ifndef ALG_SHA1
4  #   define ALG_SHA1 NO
5  #endif
6  #if ALG_SHA1
7  #define SHA1_OID(NAME)    MAKE_OID(NAME##SHA1)
8  #else
9  #define SHA1_OID(NAME)
10 #endif
11 #ifndef ALG_SHA256
12 #   define ALG_SHA256 NO
13 #endif
14 #if ALG_SHA256
15 #define SHA256_OID(NAME)  MAKE_OID(NAME##SHA256)
16 #else
17 #define SHA256_OID(NAME)
18 #endif
19 #ifndef ALG_SHA384
20 #   define ALG_SHA384 NO
21 #endif
22 #if ALG_SHA384
23 #define SHA384_OID(NAME)  MAKE_OID(NAME##SHA384)
24 #else
25 #define SHA#84_OID(NAME)
26 #endif
27 #ifndef ALG_SHA512
28 #   define ALG_SHA512 NO
29 #endif
30 #if ALG_SHA512
31 #define SHA512_OID(NAME)  MAKE_OID(NAME##SHA512)
32 #else
33 #define SHA512_OID(NAME)
34 #endif
35 #ifndef ALG_SM3_256
36 #   define ALG_SM3_256 NO
37 #endif
38 #if ALG_SM3_256
39 #define SM3_256_OID(NAME) MAKE_OID(NAME##SM3_256)
40 #else
41 #define SM3_256_OID(NAME)
42 #endif
43 #ifndef ALG_SHA3_256
44 #   define ALG_SHA3_256 NO
45 #endif
46 #if ALG_SHA3_256
47 #define SHA3_256_OID(NAME) MAKE_OID(NAME##SHA3_256)
48 #else
49 #define SHA3_256_OID(NAME)
50 #endif
51 #ifndef ALG_SHA3_384
52 #   define ALG_SHA3_384 NO
53 #endif
```

```
54  #if ALG_SHA3_384
55  #define SHA3_384_OID(NAME) MAKE_OID(NAME##SHA3_384)
56  #else
57  #define SHA3_384_OID(NAME)
58  #endif
59  #ifndef ALG_SHA3_512
60  #   define ALG_SHA3_512 NO
61  #endif
62  #if ALG_SHA3_512
63  #define SSHA3_512_OID(NAME) MAKE_OID(NAME##SHA3_512)
64  #else
65  #define SHA3_512_OID(NAME)
66  #endif
```

These are encoded to take one additional byte of algorithm selector

```
67  #define NIST_HASH       0x06, 0x09, 0x60, 0x86, 0x48, 1, 101, 3, 4, 2
68  #define NIST_SIG        0x06, 0x09, 0x60, 0x86, 0x48, 1, 101, 3, 4, 3
```

These hash OIDs used in a lot of places.

```
69  #define OID_SHA1_VALUE              0x06, 0x05, 0x2B, 0x0E, 0x03, 0x02, 0x1A
70  SHA1_OID(_);         // Expands to
71                       //      MAKE_OID(_SHA1)
72                       // which expands to:
73                       //      extern BYTE     OID_SHA1[]
74                       // or
75                       //      const BYTE      OID_SHA1[] = {OID_SHA1_VALUE}
76                       // which is:
77                       //      const BYTE      OID_SHA1[] = {0x06, 0x05, 0x2B, 0x0E,
78                       //                                   0x03, 0x02, 0x1A}
79  #define OID_SHA256_VALUE            NIST_HASH, 1
80  SHA256_OID(_);
81  #define OID_SHA384_VALUE            NIST_HASH, 2
82  SHA384_OID(_);
83  #define OID_SHA512_VALUE            NIST_HASH, 3
84  SHA512_OID(_);
85  #define OID_SM3_256_VALUE           0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55, 0x01, \
86                                      0x83, 0x11
87  SM3_256_OID(_);          // (1.2.156.10197.1.401)
88  #define OID_SHA3_256_VALUE          NIST_HASH, 8
89  SHA3_256_OID(_);
90  #define OID_SHA3_384_VALUE          NIST_HASH, 9
91  SHA3_384_OID(_);
92  #define OID_SHA3_512_VALUE          NIST_HASH, 10
93  SHA3_512_OID(_);
```

These are used for RSA-PSS

```
94  #if ALG_RSA
95  #define OID_MGF1_VALUE              0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, \
96                                      0x01, 0x01, 0x08
97  MAKE_OID(_MGF1);
98  #define OID_RSAPSS_VALUE            0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, \
99                                      0x01, 0x01, 0x0A
100 MAKE_OID(_RSAPSS);
```

This is the OID to designate the public part of an RSA key.

```
101 #define OID_PKCS1_PUB_VALUE         0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, \
102                                     0x01, 0x01, 0x01
103 MAKE_OID(_PKCS1_PUB);
```

These are used for RSA PKCS1 signature Algorithms

```
104   #define OID_PKCS1_SHA1_VALUE        0x06,0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7,       \
105                                       0x0D, 0x01, 0x01, 0x05
106   SHA1_OID(_PKCS1_);       // (1.2.840.113549.1.1.5)
107   #define OID_PKCS1_SHA256_VALUE      0x06,0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7,       \
108                                       0x0D, 0x01, 0x01, 0x0B
109   SHA256_OID(_PKCS1_);     // (1.2.840.113549.1.1.11)
110   #define OID_PKCS1_SHA384_VALUE      0x06,0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7,       \
111                                       0x0D, 0x01, 0x01, 0x0C
112   SHA384_OID(_PKCS1_);     // (1.2.840.113549.1.1.12)
113   #define OID_PKCS1_SHA512_VALUE      0x06,0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7,       \
114                                       0x0D, 0x01, 0x01, 0x0D
115   SHA512_OID(_PKCS1_);     //(1.2.840.113549.1.1.13)
116   #define OID_PKCS1_SM3_256_VALUE     0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55,      \
117                                       0x01, 0x83, 0x78
118   SM3_256_OID(_PKCS1_);    // 1.2.156.10197.1.504
119   #define OID_PKCS1_SHA3_256_VALUE    NIST_SIG, 14
120   SHA3_256_OID(_PKCS1_);
121   #define OID_PKCS1_SHA3_384_VALUE    NIST_SIG, 15
122   SHA3_256_OID(_PKCS1_);
123   #define OID_PKCS1_SHA3_512_VALUE    NIST_SIG, 16
124   SHA3_512_OID(_PKCS1_);
125   #endif // ALG_RSA
126   #if ALG_ECDSA
127   #define OID_ECDSA_SHA1_VALUE        0x06, 0x07, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, \
128                                       0x01
129   SHA1_OID(_ECDSA_);       // (1.2.840.10045.4.1) SHA1 digest signed by an ECDSA key.
130   #define OID_ECDSA_SHA256_VALUE      0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, \
131                                       0x03, 0x02
132   SHA256_OID(_ECDSA_);     // (1.2.840.10045.4.3.2) SHA256 digest signed by an ECDSA key.
133   #define OID_ECDSA_SHA384_VALUE      0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, \
134                                       0x03, 0x03
135   SHA384_OID(_ECDSA_);     // (1.2.840.10045.4.3.3) SHA384 digest signed by an ECDSA key.
136   #define OID_ECDSA_SHA512_VALUE      0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, \
137                                       0x03, 0x04
138   SHA512_OID(_ECDSA_);     // (1.2.840.10045.4.3.4) SHA512 digest signed by an ECDSA key.
139   #define OID_ECDSA_SM3_256_VALUE     0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55, 0x01,  \
140                                       0x83, 0x75
141   SM3_256_OID(_ECDSA_);    // 1.2.156.10197.1.501
142   #define OID_ECDSA_SHA3_256_VALUE    NIST_SIG, 10
143   SHA3_256_OID(_ECDSA_);
144   #define OID_ECDSA_SHA3_384_VALUE    NIST_SIG, 11
145   SHA3_384_OID(_ECDSA_);
146   #define OID_ECDSA_SHA3_512_VALUE    NIST_SIG, 12
147   SHA3_512_OID(_ECDSA_);
148   #endif // ALG_ECDSA
149   #if ALG_ECC
150   #define OID_ECC_PUBLIC_VALUE        0x06, 0x07, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x02, \
151                                       0x01
152   MAKE_OID(_ECC_PUBLIC);
153   #define OID_ECC_NIST_P192_VALUE     0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x03, \
154                                       0x01, 0x01
155   #if ECC_NIST_P192
156   MAKE_OID(_ECC_NIST_P192);    // (1.2.840.10045.3.1.1) 'nistP192'
157   #endif // ECC_NIST_P192
158   #define OID_ECC_NIST_P224_VALUE     0x06, 0x05, 0x2B, 0x81, 0x04, 0x00, 0x21
159   #if ECC_NIST_P224
160   MAKE_OID(_ECC_NIST_P224);    // (1.3.132.0.33)        'nistP224'
161   #endif // ECC_NIST_P224
162   #define OID_ECC_NIST_P256_VALUE     0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x03, \
163                                       0x01, 0x07
164   #if ECC_NIST_P256
165   MAKE_OID(_ECC_NIST_P256);    // (1.2.840.10045.3.1.7)  'nistP256'
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
166 #endif // ECC_NIST_P256
167 #define OID_ECC_NIST_P384_VALUE     0x06, 0x05, 0x2B, 0x81, 0x04, 0x00, 0x22
168 #if ECC_NIST_P384
169 MAKE_OID(_ECC_NIST_P384);    // (1.3.132.0.34)          'nistP384'
170 #endif // ECC_NIST_P384
171 #define OID_ECC_NIST_P521_VALUE     0x06, 0x05, 0x2B, 0x81, 0x04, 0x00, 0x23
172 #if ECC_NIST_P521
173 MAKE_OID(_ECC_NIST_P521);    // (1.3.132.0.35)          'nistP521'
174 #endif // ECC_NIST_P521
```

No OIDs defined for these anonymous curves

```
175 #define OID_ECC_BN_P256_VALUE       0x00
176 #if ECC_BN_P256
177 MAKE_OID(_ECC_BN_P256);
178 #endif // ECC_BN_P256
179 #define OID_ECC_BN_P638_VALUE       0x00
180 #if ECC_BN_P638
181 MAKE_OID(_ECC_BN_P638);
182 #endif // ECC_BN_P638
183 #define OID_ECC_SM2_P256_VALUE      0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55, 0x01, \
184                                     0x82, 0x2D
185 #if ECC_SM2_P256
186 MAKE_OID(_ECC_SM2_P256);     // Don't know where I found this OID. It needs checking
187 #endif // ECC_SM2_P256
188 #if ECC_BN_P256
189 #define OID_ECC_BN_P256     NULL
190 #endif // ECC_BN_P256
191 #endif // ALG_ECC
192 #define OID_SIZE(OID)   (OID[1] + 2)
193 #endif // !_OIDS_H_
```

10.1.17 PRNG_TestVectors.h

```
1 #ifndef     _MSBN_DRBG_TEST_VECTORS_H
2 #define     _MSBN_DRBG_TEST_VECTORS_H
3 //#if DRBG_ALGORITHM == TPM_ALG_AES && DRBG_KEY_BITS == 256
4 #if DRBG_KEY_SIZE_BITS == 256
```

Entropy is the size of the state. The state is the size of the key plus the IV. The IV is a block. If Key = 256 and Block = 128 then State = 384

```
5  #   define DRBG_TEST_INITIATE_ENTROPY                       \
6          0x0d, 0x15, 0xaa, 0x80, 0xb1, 0x6c, 0x3a, 0x10, \
7          0x90, 0x6c, 0xfe, 0xdb, 0x79, 0x5d, 0xae, 0x0b, \
8          0x5b, 0x81, 0x04, 0x1c, 0x5c, 0x5b, 0xfa, 0xcb, \
9          0x37, 0x3d, 0x44, 0x40, 0xd9, 0x12, 0x0f, 0x7e, \
10         0x3d, 0x6c, 0xf9, 0x09, 0x86, 0xcf, 0x52, 0xd8, \
11         0x5d, 0x3e, 0x94, 0x7d, 0x8c, 0x06, 0x1f, 0x91
12 #   define DRBG_TEST_RESEED_ENTROPY                         \
13         0x6e, 0xe7, 0x93, 0xa3, 0x39, 0x55, 0xd7, 0x2a, \
14         0xd1, 0x2f, 0xd8, 0x0a, 0x8a, 0x3f, 0xcf, 0x95, \
15         0xed, 0x3b, 0x4d, 0xac, 0x57, 0x95, 0xfe, 0x25, \
16         0xcf, 0x86, 0x9f, 0x7c, 0x27, 0x57, 0x3b, 0xbc, \
17         0x56, 0xf1, 0xac, 0xae, 0x13, 0xa6, 0x50, 0x42, \
18         0xb3, 0x40, 0x09, 0x3c, 0x46, 0x4a, 0x7a, 0x22
19 #   define DRBG_TEST_GENERATED_INTERM                       \
20         0x28, 0xe0, 0xeb, 0xb8, 0x21, 0x01, 0x66, 0x50, \
21         0x8c, 0x8f, 0x65, 0xf2, 0x20, 0x7b, 0xd0, 0xa3
22 #   define DRBG_TEST_GENERATED                              \
23         0x94, 0x6f, 0x51, 0x82, 0xd5, 0x45, 0x10, 0xb9, \
24         0x46, 0x12, 0x48, 0xf5, 0x71, 0xca, 0x06, 0xc9
25 #elif DRBG_KEY_SIZE_BITS == 128
```


 Copyright © TCG 2006-2020

```
26  #   define DRBG_TEST_INITIATE_ENTROPY                           \
27          0x8f, 0xc1, 0x1b, 0xdb, 0x5a, 0xab, 0xb7, 0xe0, \
28          0x93, 0xb6, 0x14, 0x28, 0xe0, 0x90, 0x73, 0x03, \
29          0xcb, 0x45, 0x9f, 0x3b, 0x60, 0x0d, 0xad, 0x87, \
30          0x09, 0x55, 0xf2, 0x2d, 0xa8, 0x0a, 0x44, 0xf8
31  #   define DRBG_TEST_RESEED_ENTROPY                             \
32          0x0c, 0xd5, 0x3c, 0xd5, 0xec, 0xcd, 0x5a, 0x10, \
33          0xd7, 0xea, 0x26, 0x61, 0x11, 0x25, 0x9b, 0x05, \
34          0x57, 0x4f, 0xc6, 0xdd, 0xd8, 0xbe, 0xd8, 0xbd, \
35          0x72, 0x37, 0x8c, 0xf8, 0x2f, 0x1d, 0xba, 0x2a
36  #define DRBG_TEST_GENERATED_INTERM                              \
37          0xdc, 0x3c, 0xf6, 0xbf, 0x5b, 0xd3, 0x41, 0x13, \
38          0x5f, 0x2c, 0x68, 0x11, 0xa1, 0x07, 0x1c, 0x87
39  #   define DRBG_TEST_GENERATED                                  \
40          0xb6, 0x18, 0x50, 0xde, 0xcf, 0xd7, 0x10, 0x6d, \
41          0x44, 0x76, 0x9a, 0x8e, 0x6e, 0x8c, 0x1a, 0xd4
42  #endif
43  #endif      //      _MSBN_DRBG_TEST_VECTORS_H
```

10.1.18 TpmAsn1.h

10.1.18.1 Introduction

This file contains the macro and structure definitions for the X509 commands and functions.

```
1   #ifndef _TPMASN1_H_
2   #define _TPMASN1_H_
```

10.1.18.2 Includes

```
3   #include "Tpm.h"
4   #include "OIDs.h"
```

10.1.18.3 Defined Constants

10.1.18.3.1 ASN.1 Universal Types (Class 00b

```
5   #define ASN1_EOC                0x00
6   #define ASN1_BOOLEAN            0x01
7   #define ASN1_INTEGER            0x02
8   #define ASN1_BITSTRING          0x03
9   #define ASN1_OCTET_STRING       0x04
10  #define ASN1_NULL               0x05
11  #define ASN1_OBJECT_IDENTIFIER  0x06
12  #define ASN1_OBJECT_DESCRIPTOR  0x07
13  #define ASN1_EXTERNAL           0x08
14  #define ASN1_REAL               0x09
15  #define ASN1_ENUMERATED         0x0A
16  #define ASN1_EMBEDDED           0x0B
17  #define ASN1_UTF8String         0x0C
18  #define ASN1_RELATIVE_OID       0x0D
19  #define ASN1_SEQUENCE           0x10        // Primitive + Constructed + 0x10
20  #define ASN1_SET                0x11        // Primitive + Constructed + 0x11
21  #define ASN1_NumericString      0x12
22  #define ASN1_PrintableString    0x13
23  #define ASN1_T61String          0x14
24  #define ASN1_VideoString        0x15
25  #define ASN1_IA5String          0x16
26  #define ASN1_UTCTime            0x17
27  #define ASN1_GeneralizeTime     0x18
28  #define ASN1_VisibleString      0x1A
```

```
29  #define ASN1_GeneralString      0x1B
30  #define ASN1_UniversalString    0x1C
31  #define ASN1_CHARACTER STRING   0x1D
32  #define ASN1_BMPString          0x1E
33  #define ASN1_CONSTRUCTED        0x20
34  #define ASN1_APPLICAIION_SPECIFIC   0xA0
35  #define ASN1_CONSTRUCTED_SEQUENCE (ASN1_SEQUENCE + ASN1_CONSTRUCTED)
36  #define MAX_DEPTH               10  // maximum push depth for marshaling context.
```

10.1.18.4 Macros

10.1.18.4.1 Unmarshaling Macros

```
37  #ifndef VERIFY
38  #define VERIFY(_X_) {if(!(_X_)) goto Error; }
39  #endif
```

Checks the validity of the size making sure that there is no wrap around

```
40  #define CHECK_SIZE(context, length)                                     \
41          VERIFY(   (((length) + (context)->offset) >= (context)->offset)   \
42                 && (((length) + (context)->offset) <= (context)->size))
43  #define NEXT_OCTET(context) ((context)->buffer[(context)->offset++])
44  #define PEEK_NEXT(context) ((context)->buffer[(context)->offset])
```

10.1.18.4.2 Marshaling Macros

Marshaling works in reverse order. The offset is set to the top of the buffer and, as the buffer is filled, *offset* counts down to zero. When the full thing is encoded it can be moved to the top of the buffer. This happens when the last context is closed.

```
45  #define CHECK_SPACE(context, length)    VERIFY(context->offset > length)
```

10.1.18.5 Structures

```
46  typedef struct ASN1UnmarshalContext {
47      BYTE            *buffer;    // pointer to the buffer
48      INT16            size;      // size of the buffer (a negative number indicates
49                                  // a parsing failure).
50      INT16            offset;    // current offset into the buffer (a negative number
51                                  // indicates a parsing failure). Not used
52      BYTE             tag;       // The last unmarshaled tag
53  } ASN1UnmarshalContext;
54  typedef struct ASN1MarshalContext {
55      BYTE            *buffer;    // pointer to the start of the buffer
56      INT16            offset;    // place on the top where the last entry was added
57                                  // items are added from the bottom up.
58      INT16            end;       // the end offset of the current value
59      INT16            depth;     // how many pushed end values.
60      INT16            ends[MAX_DEPTH];
61  } ASN1MarshalContext;
62  #endif // _TPMASN1_H_
```

10.1.19 X509.h

10.1.19.1 Introduction

This file contains the macro and structure definitions for the X509 commands and functions.

```
1  #ifndef _X509_H_
2  #define _X509_H_
```

10.1.19.2 Includes

```
3  #include "Tpm.h"
4  #include "TpmASN1.h"
```

10.1.19.3 Defined Constants

10.1.19.3.1 X509 Application-specific types

```
5  #define X509_SELECTION          0xA0
6  #define X509_ISSUER_UNIQUE_ID   0xA1
7  #define X509_SUBJECT_UNIQUE_ID  0xA2
8  #define X509_EXTENSIONS         0xA3
```

These defines give the order in which values appear in the TBScertificate of an x.509 certificate. These values are used to index into an array of

```
9  #define ENCODED_SIZE_REF        0
10 #define VERSION_REF             (ENCODED_SIZE_REF + 1)
11 #define SERIAL_NUMBER_REF       (VERSION_REF + 1)
12 #define SIGNATURE_REF           (SERIAL_NUMBER_REF + 1)
13 #define ISSUER_REF              (SIGNATURE_REF + 1)
14 #define VALIDITY_REF            (ISSUER_REF + 1)
15 #define SUBJECT_KEY_REF         (VALIDITY_REF + 1)
16 #define SUBJECT_PUBLIC_KEY_REF  (SUBJECT_KEY_REF + 1)
17 #define EXTENSIONS_REF          (SUBJECT_PUBLIC_KEY_REF + 1)
18 #define REF_COUNT               (EXTENSIONS_REF + 1)
```

10.1.19.4 Structures

Used to access the fields of a TBSsignature some of which are in the *in_CertifyX509* structure and some of which are in the *out_CertifyX509* structure.

```
19 typedef struct stringRef
20 {
21     BYTE        *buf;
22     INT16        len;
23 } stringRef;
```

This is defined to avoid bit by bit comparisons within a UINT32

```
24 typedef union x509KeyUsageUnion {
25     TPMA_X509_KEY_USAGE     x509;
26     UINT32                  integer;
27 } x509KeyUsageUnion;
```

10.1.19.5 Global X509 Constants

These values are instanced by X509_spt.c and referenced by other X509-related files. This is the DER-encoded value for the Key Usage OID (2.5.29.15). This is the full OID, not just the numeric value

```
28 #define OID_KEY_USAGE_EXTENSION_VALUE  0x06, 0x03, 0x55, 0x1D, 0x0F
29 MAKE_OID(_KEY_USAGE_EXTENSION);
```

This is the DER-encoded value for the TCG-defined TPMA_OBJECT OID (2.23.133.10.1.1.1)

 Copyright © TCG 2006-2020

```
30 #define OID_TCG_TPMA_OBJECT_VALUE       0x06, 0x07, 0x67, 0x81, 0x05, 0x0a, 0x01,   \
31                                         0x01, 0x01
32 MAKE_OID(_TCG_TPMA_OBJECT);
33 #ifdef _X509_SPT_
```

If a bit is SET in KEY_USAGE_SIGN is also SET in *keyUsagem* then the associated key has to have *sign* SET.

```
34 const x509KeyUsageUnion KEY_USAGE_SIGN =
35 { TPMA_X509_KEY_USAGE_INITIALIZER(
36     /* bits_at_0        */ 0, /* decipheronly    */ 0,  /* encipheronly   */ 0,
37     /* crlsign          */ 1, /* keycertsign     */ 1,  /* keyagreement   */ 0,
38     /* dataencipherment */ 0, /* keyencipherment */ 0,  /* nonrepudiation */ 0,
39     /* digitalsignature */ 1) };
```

If a bit is SET in KEY_USAGE_DECRYPT is also SET in *keyUsagem* then the associated key has to have *decrypt* SET.

```
40 const x509KeyUsageUnion KEY_USAGE_DECRYPT =
41 { TPMA_X509_KEY_USAGE_INITIALIZER(
42     /* bits_at_0        */ 0, /* decipheronly    */ 1,  /* encipheronly   */ 1,
43     /* crlsign          */ 0, /* keycertsign     */ 0,  /* keyagreement   */ 1,
44     /* dataencipherment */ 1, /* keyencipherment */ 1,  /* nonrepudiation */ 0,
45     /* digitalsignature */ 0) };
46 #else
47 extern x509KeyUsageUnion KEY_USAGE_SIGN;
48 extern x509KeyUsageUnion KEY_USAGE_DECRYPT;
49 #endif
50
51 #endif // _X509_H_
```

10.1.20 TpmAlgorithmDefines.h

This file contains the algorithm values from the TCG Algorithm Registry.

```
1 #ifndef _TPM_ALGORITHM_DEFINES_H_
2 #define _TPM_ALGORITHM_DEFINES_H_
```

Table 2:3 - Definition of Base Types Base Types are in BaseTypes.h

```
3 #define ECC_CURVES                              \
4             {TPM_ECC_BN_P256,   TPM_ECC_BN_P638,   TPM_ECC_NIST_P192,            \
5              TPM_ECC_NIST_P224, TPM_ECC_NIST_P256, TPM_ECC_NIST_P384,            \
6              TPM_ECC_NIST_P521, TPM_ECC_SM2_P256}
7 #define ECC_CURVE_COUNT                         \
8             (ECC_BN_P256   + ECC_BN_P638   + ECC_NIST_P192 + ECC_NIST_P224 +     \
9              ECC_NIST_P256 + ECC_NIST_P384 + ECC_NIST_P521 + ECC_SM2_P256)
10 #define MAX_ECC_KEY_BITS                        \
11            MAX(ECC_BN_P256 * 256,   MAX(ECC_BN_P638 * 638,                       \
12            MAX(ECC_NIST_P192 * 192, MAX(ECC_NIST_P224 * 224,                     \
13            MAX(ECC_NIST_P256 * 256, MAX(ECC_NIST_P384 * 384,                     \
14            MAX(ECC_NIST_P521 * 521, MAX(ECC_SM2_P256 * 256,                      \
15            0))))))))
16 #define MAX_ECC_KEY_BYTES                       BITS_TO_BYTES(MAX_ECC_KEY_BITS)
```

Table 0:6 - Defines for PLATFORM Values

```
17 #define PLATFORM_FAMILY          TPM_SPEC_FAMILY
18 #define PLATFORM_LEVEL           TPM_SPEC_LEVEL
19 #define PLATFORM_VERSION         TPM_SPEC_VERSION
20 #define PLATFORM_YEAR            TPM_SPEC_YEAR
21 #define PLATFORM_DAY_OF_YEAR     TPM_SPEC_DAY_OF_YEAR
```

 Copyright © TCG 2006-2020

Table 1:3 - Defines for RSA Asymmetric Cipher Algorithm Constants

```
22  #define RSA_KEY_SIZES_BITS          \
23              (1024 * RSA_1024),  (2048 * RSA_2048), (3072 * RSA_3072),             \
24               (4096 * RSA_4096)
25  #if   RSA_4096
26  #   define RSA_MAX_KEY_SIZE_BITS    4096
27  #elif RSA_3072
28  #   define RSA_MAX_KEY_SIZE_BITS    3072
29  #elif RSA_2048
30  #   define RSA_MAX_KEY_SIZE_BITS    2048
31  #elif RSA_1024
32  #   define RSA_MAX_KEY_SIZE_BITS    1024
33  #else
34  #   define RSA_MAX_KEY_SIZE_BITS    0
35  #endif
36  #define MAX_RSA_KEY_BITS            RSA_MAX_KEY_SIZE_BITS
37  #define MAX_RSA_KEY_BYTES           ((RSA_MAX_KEY_SIZE_BITS + 7) / 8)
```

Table 1:13 - Defines for SHA1 Hash Values

```
38  #define SHA1_DIGEST_SIZE    20
39  #define SHA1_BLOCK_SIZE     64
```

Table 1:14 - Defines for SHA256 Hash Values

```
40  #define SHA256_DIGEST_SIZE  32
41  #define SHA256_BLOCK_SIZE   64
```

Table 1:15 - Defines for SHA384 Hash Values

```
42  #define SHA384_DIGEST_SIZE  48
43  #define SHA384_BLOCK_SIZE   128
```

Table 1:16 - Defines for SHA512 Hash Values

```
44  #define SHA512_DIGEST_SIZE  64
45  #define SHA512_BLOCK_SIZE   128
```

Table 1:17 - Defines for SM3_256 Hash Values

```
46  #define SM3_256_DIGEST_SIZE     32
47  #define SM3_256_BLOCK_SIZE      64
```

Table 1:18 - Defines for SHA3_256 Hash Values

```
48  #define SHA3_256_DIGEST_SIZE    32
49  #define SHA3_256_BLOCK_SIZE     136
```

Table 1:19 - Defines for SHA3_384 Hash Values

```
50  #define SHA3_384_DIGEST_SIZE    48
51  #define SHA3_384_BLOCK_SIZE     104
```

Table 1:20 - Defines for SHA3_512 Hash Values

```
52  #define SHA3_512_DIGEST_SIZE    64
53  #define SHA3_512_BLOCK_SIZE     72
```

Table 1:21 - Defines for AES Symmetric Cipher Algorithm Constants

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
54  #define AES_KEY_SIZES_BITS           \
55              (128 * AES_128), (192 * AES_192), (256 * AES_256)
56  #if   AES_256
57  #   define AES_MAX_KEY_SIZE_BITS    256
58  #elif AES_192
59  #   define AES_MAX_KEY_SIZE_BITS    192
60  #elif AES_128
61  #   define AES_MAX_KEY_SIZE_BITS    128
62  #else
63  #   define AES_MAX_KEY_SIZE_BITS    0
64  #endif
65  #define MAX_AES_KEY_BITS            AES_MAX_KEY_SIZE_BITS
66  #define MAX_AES_KEY_BYTES           ((AES_MAX_KEY_SIZE_BITS + 7) / 8)
67  #define AES_128_BLOCK_SIZE_BYTES    (AES_128 * 16)
68  #define AES_192_BLOCK_SIZE_BYTES    (AES_192 * 16)
69  #define AES_256_BLOCK_SIZE_BYTES    (AES_256 * 16)
70  #define AES_BLOCK_SIZES             \
71              AES_128_BLOCK_SIZE_BYTES, AES_192_BLOCK_SIZE_BYTES,                    \
72              AES_256_BLOCK_SIZE_BYTES
73  #if   ALG_AES
74  #   define AES_MAX_BLOCK_SIZE       16
75  #else
76  #   define AES_MAX_BLOCK_SIZE       0
77  #endif
78  #define MAX_AES_BLOCK_SIZE_BYTES    AES_MAX_BLOCK_SIZE
```

Table 1:22 - Defines for SM4 Symmetric Cipher Algorithm Constants

```
79  #define SM4_KEY_SIZES_BITS          (128 * SM4_128)
80  #if   SM4_128
81  #   define SM4_MAX_KEY_SIZE_BITS    128
82  #else
83  #   define SM4_MAX_KEY_SIZE_BITS    0
84  #endif
85  #define MAX_SM4_KEY_BITS            SM4_MAX_KEY_SIZE_BITS
86  #define MAX_SM4_KEY_BYTES           ((SM4_MAX_KEY_SIZE_BITS + 7) / 8)
87  #define SM4_128_BLOCK_SIZE_BYTES    (SM4_128 * 16)
88  #define SM4_BLOCK_SIZES             SM4_128_BLOCK_SIZE_BYTES
89  #if   ALG_SM4
90  #   define SM4_MAX_BLOCK_SIZE       16
91  #else
92  #   define SM4_MAX_BLOCK_SIZE       0
93  #endif
94  #define MAX_SM4_BLOCK_SIZE_BYTES    SM4_MAX_BLOCK_SIZE
```

Table 1:23 - Defines for CAMELLIA Symmetric Cipher Algorithm Constants

```
95  #define CAMELLIA_KEY_SIZES_BITS         \
96              (128 * CAMELLIA_128), (192 * CAMELLIA_192), (256 * CAMELLIA_256)
97  #if   CAMELLIA_256
98  #   define CAMELLIA_MAX_KEY_SIZE_BITS   256
99  #elif CAMELLIA_192
100 #   define CAMELLIA_MAX_KEY_SIZE_BITS   192
101 #elif CAMELLIA_128
102 #   define CAMELLIA_MAX_KEY_SIZE_BITS   128
103 #else
104 #   define CAMELLIA_MAX_KEY_SIZE_BITS   0
105 #endif
106 #define MAX_CAMELLIA_KEY_BITS           CAMELLIA_MAX_KEY_SIZE_BITS
107 #define MAX_CAMELLIA_KEY_BYTES          ((CAMELLIA_MAX_KEY_SIZE_BITS + 7) / 8)
108 #define CAMELLIA_128_BLOCK_SIZE_BYTES   (CAMELLIA_128 * 16)
109 #define CAMELLIA_192_BLOCK_SIZE_BYTES   (CAMELLIA_192 * 16)
110 #define CAMELLIA_256_BLOCK_SIZE_BYTES   (CAMELLIA_256 * 16)
111 #define CAMELLIA_BLOCK_SIZES            \
112             CAMELLIA_128_BLOCK_SIZE_BYTES, CAMELLIA_192_BLOCK_SIZE_BYTES,          \
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
113                 CAMELLIA_256_BLOCK_SIZE_BYTES
114     #if   ALG_CAMELLIA
115     #   define CAMELLIA_MAX_BLOCK_SIZE      16
116     #else
117     #   define CAMELLIA_MAX_BLOCK_SIZE      0
118     #endif
119     #define MAX_CAMELLIA_BLOCK_SIZE_BYTES   CAMELLIA_MAX_BLOCK_SIZE
```

Table 1:24 - Defines for TDES Symmetric Cipher Algorithm Constants

```
120     #define TDES_KEY_SIZES_BITS         (128 * TDES_128), (192 * TDES_192)
121     #if   TDES_192
122     #   define TDES_MAX_KEY_SIZE_BITS   192
123     #elif TDES_128
124     #   define TDES_MAX_KEY_SIZE_BITS   128
125     #else
126     #   define TDES_MAX_KEY_SIZE_BITS   0
127     #endif
128     #define MAX_TDES_KEY_BITS           TDES_MAX_KEY_SIZE_BITS
129     #define MAX_TDES_KEY_BYTES          ((TDES_MAX_KEY_SIZE_BITS + 7) / 8)
130     #define TDES_128_BLOCK_SIZE_BYTES   (TDES_128 * 8)
131     #define TDES_192_BLOCK_SIZE_BYTES   (TDES_192 * 8)
132     #define TDES_BLOCK_SIZES            \
133                 TDES_128_BLOCK_SIZE_BYTES, TDES_192_BLOCK_SIZE_BYTES
134     #if   ALG_TDES
135     #   define TDES_MAX_BLOCK_SIZE      8
136     #else
137     #   define TDES_MAX_BLOCK_SIZE      0
138     #endif
139     #define MAX_TDES_BLOCK_SIZE_BYTES   TDES_MAX_BLOCK_SIZE
```

Additional values for benefit of code

```
140     #define TPM_CC_FIRST                        0x0000011F
141     #define TPM_CC_LAST                         0x00000198
142     #if COMPRESSED_LISTS
143     #define ADD_FILL            0
144     #else
145     #define ADD_FILL            1
146     #endif
```

Size the array of library commands based on whether or not the array is packed (only defined commands) or dense (having entries for unimplemented commands)

```
147     #define LIBRARY_COMMAND_ARRAY_SIZE      (0      \
148         + (ADD_FILL || CC_NV_UndefineSpaceSpecial)          /* 0x0000011F */        \
149         + (ADD_FILL || CC_EvictControl)                     /* 0x00000120 */        \
150         + (ADD_FILL || CC_HierarchyControl)                 /* 0x00000121 */        \
151         + (ADD_FILL || CC_NV_UndefineSpace)                 /* 0x00000122 */        \
152         +  ADD_FILL                                         /* 0x00000123 */        \
153         + (ADD_FILL || CC_ChangeEPS)                        /* 0x00000124 */        \
154         + (ADD_FILL || CC_ChangePPS)                        /* 0x00000125 */        \
155         + (ADD_FILL || CC_Clear)                            /* 0x00000126 */        \
156         + (ADD_FILL || CC_ClearControl)                     /* 0x00000127 */        \
157         + (ADD_FILL || CC_ClockSet)                         /* 0x00000128 */        \
158         + (ADD_FILL || CC_HierarchyChangeAuth)              /* 0x00000129 */        \
159         + (ADD_FILL || CC_NV_DefineSpace)                   /* 0x0000012A */        \
160         + (ADD_FILL || CC_PCR_Allocate)                     /* 0x0000012B */        \
161         + (ADD_FILL || CC_PCR_SetAuthPolicy)                /* 0x0000012C */        \
162         + (ADD_FILL || CC_PP_Commands)                      /* 0x0000012D */        \
163         + (ADD_FILL || CC_SetPrimaryPolicy)                 /* 0x0000012E */        \
164         + (ADD_FILL || CC_FieldUpgradeStart)                /* 0x0000012F */        \
165         + (ADD_FILL || CC_ClockRateAdjust)                  /* 0x00000130 */        \
166         + (ADD_FILL || CC_CreatePrimary)                    /* 0x00000131 */        \
```

```
167       + (ADD_FILL || CC_NV_GlobalWriteLock)                  /* 0x00000132 */       \
168       + (ADD_FILL || CC_GetCommandAuditDigest)               /* 0x00000133 */       \
169       + (ADD_FILL || CC_NV_Increment)                        /* 0x00000134 */       \
170       + (ADD_FILL || CC_NV_SetBits)                          /* 0x00000135 */       \
171       + (ADD_FILL || CC_NV_Extend)                           /* 0x00000136 */       \
172       + (ADD_FILL || CC_NV_Write)                            /* 0x00000137 */       \
173       + (ADD_FILL || CC_NV_WriteLock)                        /* 0x00000138 */       \
174       + (ADD_FILL || CC_DictionaryAttackLockReset)           /* 0x00000139 */       \
175       + (ADD_FILL || CC_DictionaryAttackParameters)          /* 0x0000013A */       \
176       + (ADD_FILL || CC_NV_ChangeAuth)                       /* 0x0000013B */       \
177       + (ADD_FILL || CC_PCR_Event)                           /* 0x0000013C */       \
178       + (ADD_FILL || CC_PCR_Reset)                           /* 0x0000013D */       \
179       + (ADD_FILL || CC_SequenceComplete)                    /* 0x0000013E */       \
180       + (ADD_FILL || CC_SetAlgorithmSet)                     /* 0x0000013F */       \
181       + (ADD_FILL || CC_SetCommandCodeAuditStatus)           /* 0x00000140 */       \
182       + (ADD_FILL || CC_FieldUpgradeData)                    /* 0x00000141 */       \
183       + (ADD_FILL || CC_IncrementalSelfTest)                 /* 0x00000142 */       \
184       + (ADD_FILL || CC_SelfTest)                            /* 0x00000143 */       \
185       + (ADD_FILL || CC_Startup)                             /* 0x00000144 */       \
186       + (ADD_FILL || CC_Shutdown)                            /* 0x00000145 */       \
187       + (ADD_FILL || CC_StirRandom)                          /* 0x00000146 */       \
188       + (ADD_FILL || CC_ActivateCredential)                  /* 0x00000147 */       \
189       + (ADD_FILL || CC_Certify)                             /* 0x00000148 */       \
190       + (ADD_FILL || CC_PolicyNV)                            /* 0x00000149 */       \
191       + (ADD_FILL || CC_CertifyCreation)                     /* 0x0000014A */       \
192       + (ADD_FILL || CC_Duplicate)                           /* 0x0000014B */       \
193       + (ADD_FILL || CC_GetTime)                             /* 0x0000014C */       \
194       + (ADD_FILL || CC_GetSessionAuditDigest)               /* 0x0000014D */       \
195       + (ADD_FILL || CC_NV_Read)                             /* 0x0000014E */       \
196       + (ADD_FILL || CC_NV_ReadLock)                         /* 0x0000014F */       \
197       + (ADD_FILL || CC_ObjectChangeAuth)                    /* 0x00000150 */       \
198       + (ADD_FILL || CC_PolicySecret)                        /* 0x00000151 */       \
199       + (ADD_FILL || CC_Rewrap)                              /* 0x00000152 */       \
200       + (ADD_FILL || CC_Create)                              /* 0x00000153 */       \
201       + (ADD_FILL || CC_ECDH_ZGen)                           /* 0x00000154 */       \
202       + (ADD_FILL || CC_HMAC || CC_MAC)                      /* 0x00000155 */       \
203       + (ADD_FILL || CC_Import)                              /* 0x00000156 */       \
204       + (ADD_FILL || CC_Load)                                /* 0x00000157 */       \
205       + (ADD_FILL || CC_Quote)                               /* 0x00000158 */       \
206       + (ADD_FILL || CC_RSA_Decrypt)                         /* 0x00000159 */       \
207       +  ADD_FILL                                            /* 0x0000015A */       \
208       + (ADD_FILL || CC_HMAC_Start || CC_MAC_Start)          /* 0x0000015B */       \
209       + (ADD_FILL || CC_SequenceUpdate)                      /* 0x0000015C */       \
210       + (ADD_FILL || CC_Sign)                                /* 0x0000015D */       \
211       + (ADD_FILL || CC_Unseal)                              /* 0x0000015E */       \
212       +  ADD_FILL                                            /* 0x0000015F */       \
213       + (ADD_FILL || CC_PolicySigned)                        /* 0x00000160 */       \
214       + (ADD_FILL || CC_ContextLoad)                         /* 0x00000161 */       \
215       + (ADD_FILL || CC_ContextSave)                         /* 0x00000162 */       \
216       + (ADD_FILL || CC_ECDH_KeyGen)                         /* 0x00000163 */       \
217       + (ADD_FILL || CC_EncryptDecrypt)                      /* 0x00000164 */       \
218       + (ADD_FILL || CC_FlushContext)                        /* 0x00000165 */       \
219       +  ADD_FILL                                            /* 0x00000166 */       \
220       + (ADD_FILL || CC_LoadExternal)                        /* 0x00000167 */       \
221       + (ADD_FILL || CC_MakeCredential)                      /* 0x00000168 */       \
222       + (ADD_FILL || CC_NV_ReadPublic)                       /* 0x00000169 */       \
223       + (ADD_FILL || CC_PolicyAuthorize)                     /* 0x0000016A */       \
224       + (ADD_FILL || CC_PolicyAuthValue)                     /* 0x0000016B */       \
225       + (ADD_FILL || CC_PolicyCommandCode)                   /* 0x0000016C */       \
226       + (ADD_FILL || CC_PolicyCounterTimer)                  /* 0x0000016D */       \
227       + (ADD_FILL || CC_PolicyCpHash)                        /* 0x0000016E */       \
228       + (ADD_FILL || CC_PolicyLocality)                      /* 0x0000016F */       \
229       + (ADD_FILL || CC_PolicyNameHash)                      /* 0x00000170 */       \
230       + (ADD_FILL || CC_PolicyOR)                            /* 0x00000171 */       \
231       + (ADD_FILL || CC_PolicyTicket)                        /* 0x00000172 */       \
232       + (ADD_FILL || CC_ReadPublic)                          /* 0x00000173 */       \
```

 Copyright © TCG 2006-2020

```
233      + (ADD_FILL || CC_RSA_Encrypt)                        /* 0x00000174 */       \
234      +  ADD_FILL                                           /* 0x00000175 */       \
235      + (ADD_FILL || CC_StartAuthSession)                   /* 0x00000176 */       \
236      + (ADD_FILL || CC_VerifySignature)                    /* 0x00000177 */       \
237      + (ADD_FILL || CC_ECC_Parameters)                     /* 0x00000178 */       \
238      + (ADD_FILL || CC_FirmwareRead)                       /* 0x00000179 */       \
239      + (ADD_FILL || CC_GetCapability)                      /* 0x0000017A */       \
240      + (ADD_FILL || CC_GetRandom)                          /* 0x0000017B */       \
241      + (ADD_FILL || CC_GetTestResult)                      /* 0x0000017C */       \
242      + (ADD_FILL || CC_Hash)                               /* 0x0000017D */       \
243      + (ADD_FILL || CC_PCR_Read)                           /* 0x0000017E */       \
244      + (ADD_FILL || CC_PolicyPCR)                          /* 0x0000017F */       \
245      + (ADD_FILL || CC_PolicyRestart)                      /* 0x00000180 */       \
246      + (ADD_FILL || CC_ReadClock)                          /* 0x00000181 */       \
247      + (ADD_FILL || CC_PCR_Extend)                         /* 0x00000182 */       \
248      + (ADD_FILL || CC_PCR_SetAuthValue)                   /* 0x00000183 */       \
249      + (ADD_FILL || CC_NV_Certify)                         /* 0x00000184 */       \
250      + (ADD_FILL || CC_EventSequenceComplete)              /* 0x00000185 */       \
251      + (ADD_FILL || CC_HashSequenceStart)                  /* 0x00000186 */       \
252      + (ADD_FILL || CC_PolicyPhysicalPresence)             /* 0x00000187 */       \
253      + (ADD_FILL || CC_PolicyDuplicationSelect)            /* 0x00000188 */       \
254      + (ADD_FILL || CC_PolicyGetDigest)                    /* 0x00000189 */       \
255      + (ADD_FILL || CC_TestParms)                          /* 0x0000018A */       \
256      + (ADD_FILL || CC_Commit)                             /* 0x0000018B */       \
257      + (ADD_FILL || CC_PolicyPassword)                     /* 0x0000018C */       \
258      + (ADD_FILL || CC_ZGen_2Phase)                        /* 0x0000018D */       \
259      + (ADD_FILL || CC_EC_Ephemeral)                       /* 0x0000018E */       \
260      + (ADD_FILL || CC_PolicyNvWritten)                    /* 0x0000018F */       \
261      + (ADD_FILL || CC_PolicyTemplate)                     /* 0x00000190 */       \
262      + (ADD_FILL || CC_CreateLoaded)                       /* 0x00000191 */       \
263      + (ADD_FILL || CC_PolicyAuthorizeNV)                  /* 0x00000192 */       \
264      + (ADD_FILL || CC_EncryptDecrypt2)                    /* 0x00000193 */       \
265      + (ADD_FILL || CC_AC_GetCapability)                   /* 0x00000194 */       \
266      + (ADD_FILL || CC_AC_Send)                            /* 0x00000195 */       \
267      + (ADD_FILL || CC_Policy_AC_SendSelect)               /* 0x00000196 */       \
268      + (ADD_FILL || CC_CertifyX509)                        /* 0x00000197 */       \
269      + (ADD_FILL || CC_ACT_SetTimeout)                     /* 0x00000198 */       \
270      )
271  #define VENDOR_COMMAND_ARRAY_SIZE   (0 + CC_Vendor_TCG_Test)
272  #define COMMAND_COUNT       (LIBRARY_COMMAND_ARRAY_SIZE + VENDOR_COMMAND_ARRAY_SIZE)
273  #define HASH_COUNT          \
274              (ALG_SHA1     + ALG_SHA256    + ALG_SHA384    + ALG_SHA3_256 +          \
275               ALG_SHA3_384 + ALG_SHA3_512 + ALG_SHA512    + ALG_SM3_256)
276  #define MAX_HASH_BLOCK_SIZE \
277              (MAX(ALG_SHA1     * SHA1_BLOCK_SIZE,                                   \
278               MAX(ALG_SHA256   * SHA256_BLOCK_SIZE,                                 \
279               MAX(ALG_SHA384   * SHA384_BLOCK_SIZE,                                 \
280               MAX(ALG_SHA3_256 * SHA3_256_BLOCK_SIZE,                               \
281               MAX(ALG_SHA3_384 * SHA3_384_BLOCK_SIZE,                               \
282               MAX(ALG_SHA3_512 * SHA3_512_BLOCK_SIZE,                               \
283               MAX(ALG_SHA512   * SHA512_BLOCK_SIZE,                                 \
284               MAX(ALG_SM3_256  * SM3_256_BLOCK_SIZE,                                \
285               0))))))))
286  #define MAX_DIGEST_SIZE     \
287              (MAX(ALG_SHA1     * SHA1_DIGEST_SIZE,                                  \
288               MAX(ALG_SHA256   * SHA256_DIGEST_SIZE,                                \
289               MAX(ALG_SHA384   * SHA384_DIGEST_SIZE,                                \
290               MAX(ALG_SHA3_256 * SHA3_256_DIGEST_SIZE,                              \
291               MAX(ALG_SHA3_384 * SHA3_384_DIGEST_SIZE,                              \
292               MAX(ALG_SHA3_512 * SHA3_512_DIGEST_SIZE,                              \
293               MAX(ALG_SHA512   * SHA512_DIGEST_SIZE,                                \
294               MAX(ALG_SM3_256  * SM3_256_DIGEST_SIZE,                               \
295               0))))))))
296  #if MAX_DIGEST_SIZE == 0 || MAX_HASH_BLOCK_SIZE == 0
297  #error "Hash data not valid"
298  #endif
```


 Copyright © TCG 2006-2020

Define the 2B structure that would hold any hash block

```
299     TPM2B_TYPE(MAX_HASH_BLOCK, MAX_HASH_BLOCK_SIZE);
```

Following typedef is for some old code

```
300     typedef TPM2B_MAX_HASH_BLOCK    TPM2B_HASH_BLOCK;
```

Additional symmetric constants

```
301     #define MAX_SYM_KEY_BITS        \
302                 (MAX(AES_MAX_KEY_SIZE_BITS,      MAX(CAMELLIA_MAX_KEY_SIZE_BITS,       \
303                  MAX(SM4_MAX_KEY_SIZE_BITS,      MAX(TDES_MAX_KEY_SIZE_BITS,           \
304                  0)))))
305     #define MAX_SYM_KEY_BYTES       ((MAX_SYM_KEY_BITS + 7) / 8)
306     #define MAX_SYM_BLOCK_SIZE      \
307                 (MAX(AES_MAX_BLOCK_SIZE,      MAX(CAMELLIA_MAX_BLOCK_SIZE,             \
308                  MAX(SM4_MAX_BLOCK_SIZE,      MAX(TDES_MAX_BLOCK_SIZE,                 \
309                  0)))))
310     #if MAX_SYM_KEY_BITS == 0 || MAX_SYM_BLOCK_SIZE == 0
311     #   error Bad size for MAX_SYM_KEY_BITS or MAX_SYM_BLOCK
312     #endif
313     #endif // _TPM_ALGORITHM_DEFINES_H_
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.2 Source

10.2.1 AlgorithmTests.c

10.2.1.1 Introduction

This file contains the code to perform the various self-test functions.

NOTE: In this implementation, large local variables are made static to minimize stack usage, which is critical for stack-constrained platforms.

10.2.1.2 Includes and Defines

```
1   #include     "Tpm.h"
2   #define      SELF_TEST_DATA
3   #if SELF_TEST
```

These includes pull in the data structures. They contain data definitions for the various tests.

```
4   #include     "SelfTest.h"
5   #include     "SymmetricTest.h"
6   #include     "RsaTestData.h"
7   #include     "EccTestData.h"
8   #include     "HashTestData.h"
9   #include     "KdfTestData.h"
10  #define TEST_DEFAULT_TEST_HASH(vector)                                    \
11              if(TEST_BIT(DEFAULT_TEST_HASH, g_toTest))                     \
12                  TestHash(DEFAULT_TEST_HASH, vector);
```

Make sure that the algorithm has been tested

```
13  #define CLEAR_BOTH(alg)      {    CLEAR_BIT(alg, *toTest);                \
14                                    if(toTest != &g_toTest)                 \
15                                        CLEAR_BIT(alg, g_toTest); }
16  #define SET_BOTH(alg)      {    SET_BIT(alg, *toTest);                    \
17                                    if(toTest != &g_toTest)                 \
18                                        SET_BIT(alg, g_toTest); }
19  #define TEST_BOTH(alg)        ((toTest != &g_toTest)                      \
20                               ? TEST_BIT(alg, *toTest) || TEST_BIT(alg, g_toTest)   \
21                               : TEST_BIT(alg, *toTest))
```

Can only cancel if doing a list.

```
22  #define CHECK_CANCELED                                                    \
23      if(_plat__IsCanceled() && toTest != &g_toTest)                        \
24          return TPM_RC_CANCELED;
```

10.2.1.3 Hash Tests

10.2.1.3.1 Description

The hash test does a known-value HMAC using the specified hash algorithm.

10.2.1.3.2 TestHash()

The hash test function.

```
25    static TPM_RC
26    TestHash(
27        TPM_ALG_ID            hashAlg,
28        ALGORITHM_VECTOR      *toTest
29        )
30    {
31        static TPM2B_DIGEST       computed;  // value computed
32        static HMAC_STATE         state;
33        UINT16                    digestSize;
34        const TPM2B              *testDigest = NULL;
35    //    TPM2B_TYPE(HMAC_BLOCK, DEFAULT_TEST_HASH_BLOCK_SIZE);
36
37        pAssert(hashAlg != ALG_NULL_VALUE);
38        switch(hashAlg)
39        {
40    #if ALG_SHA1
41            case ALG_SHA1_VALUE:
42                testDigest = &c_SHA1_digest.b;
43                break;
44    #endif
45    #if ALG_SHA256
46            case ALG_SHA256_VALUE:
47                testDigest = &c_SHA256_digest.b;
48                break;
49    #endif
50    #if ALG_SHA384
51            case ALG_SHA384_VALUE:
52                testDigest = &c_SHA384_digest.b;
53                break;
54    #endif
55    #if ALG_SHA512
56            case ALG_SHA512_VALUE:
57                testDigest = &c_SHA512_digest.b;
58                break;
59    #endif
60    #if ALG_SM3_256
61            case ALG_SM3_256_VALUE:
62    // There are currently not test vectors for SM3
63    //          testDigest = &c_SM3_256_digest.b;
64                testDigest = NULL;
65                break;
66    #endif
67            default:
68                FAIL(FATAL_ERROR_INTERNAL);
69        }
70        // Clear the to-test bits
71        CLEAR_BOTH(hashAlg);
72
73        // If there is an algorithm without test vectors, then assume that things are OK.
74        if(testDigest == NULL)
75            return TPM_RC_SUCCESS;
76
77        // Set the HMAC key to twice the digest size
78        digestSize = CryptHashGetDigestSize(hashAlg);
79        CryptHmacStart(&state, hashAlg, digestSize * 2,
80                       (BYTE *)c_hashTestKey.t.buffer);
81        CryptDigestUpdate(&state.hashState, 2 * CryptHashGetBlockSize(hashAlg),
82                          (BYTE *)c_hashTestData.t.buffer);
83        computed.t.size = digestSize;
84        CryptHmacEnd(&state, digestSize, computed.t.buffer);
85        if((testDigest->size != computed.t.size)
86           || (memcmp(testDigest->buffer, computed.t.buffer, computed.b.size) != 0))
87            SELF_TEST_FAILURE;
88        return TPM_RC_SUCCESS;
89    }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.2.1.4 Symmetric Test Functions

10.2.1.4.1 MakeIv()

Internal function to make the appropriate IV depending on the mode.

```
 90  static UINT32
 91  MakeIv(
 92      TPM_ALG_ID    mode,     // IN: symmetric mode
 93      UINT32        size,     // IN: block size of the algorithm
 94      BYTE         *iv        // OUT: IV to fill in
 95      )
 96  {
 97      BYTE          i;
 98
 99      if(mode == ALG_ECB_VALUE)
100          return 0;
101      if(mode == ALG_CTR_VALUE)
102      {
103          // The test uses an IV that has 0xff in the last byte
104          for(i = 1; i <= size; i++)
105              *iv++ = 0xff - (BYTE)(size - i);
106      }
107      else
108      {
109          for(i = 0; i < size; i++)
110              *iv++ = i;
111      }
112      return size;
113  }
```

10.2.1.4.2 TestSymmetricAlgorithm()

Function to test a specific algorithm, key size, and mode.

```
114  static void
115  TestSymmetricAlgorithm(
116      const SYMMETRIC_TEST_VECTOR     *test,          //
117      TPM_ALG_ID                       mode           //
118      )
119  {
120      static BYTE                  encrypted[MAX_SYM_BLOCK_SIZE * 2];
121      static BYTE                  decrypted[MAX_SYM_BLOCK_SIZE * 2];
122      static TPM2B_IV              iv;
123  //
124      // Get the appropriate IV
125      iv.t.size = (UINT16)MakeIv(mode, test->ivSize, iv.t.buffer);
126
127      // Encrypt known data
128      CryptSymmetricEncrypt(encrypted, test->alg, test->keyBits, test->key, &iv,
129                            mode, test->dataInOutSize, test->dataIn);
130      // Check that it matches the expected value
131      if(!MemoryEqual(encrypted, test->dataOut[mode - ALG_CTR_VALUE],
132                      test->dataInOutSize))
133          SELF_TEST_FAILURE;
134      // Reinitialize the iv for decryption
135      MakeIv(mode, test->ivSize, iv.t.buffer);
136      CryptSymmetricDecrypt(decrypted, test->alg, test->keyBits, test->key, &iv,
137                            mode, test->dataInOutSize,
138                            test->dataOut[mode - ALG_CTR_VALUE]);
139      // Make sure that it matches what we started with
140      if(!MemoryEqual(decrypted, test->dataIn, test->dataInOutSize))
141          SELF_TEST_FAILURE;
```

 Copyright © TCG 2006-2020

```
142 }
```

10.2.1.4.3 AllSymsAreDone()

Checks if both symmetric algorithms have been tested. This is put here so that addition of a symmetric algorithm will be relatively easy to handle

Return Value	Meaning
TRUE(1)	all symmetric algorithms tested
FALSE(0)	not all symmetric algorithms tested

```
143 static BOOL
144 AllSymsAreDone(
145     ALGORITHM_VECTOR        *toTest
146     )
147 {
148     return (!TEST_BOTH(ALG_AES_VALUE) && !TEST_BOTH(ALG_SM4_VALUE));
149 }
```

10.2.1.4.4 AllModesAreDone()

Checks if all the modes have been tested

Return Value	Meaning
TRUE(1)	all modes tested
FALSE(0)	all modes not tested

```
150 static BOOL
151 AllModesAreDone(
152     ALGORITHM_VECTOR            *toTest
153     )
154 {
155     TPM_ALG_ID                  alg;
156     for(alg = TPM_SYM_MODE_FIRST; alg <= TPM_SYM_MODE_LAST; alg++)
157         if(TEST_BOTH(alg))
158             return FALSE;
159     return TRUE;
160 }
```

10.2.1.4.5 TestSymmetric()

If *alg* is a symmetric block cipher, then all of the modes that are selected are tested. If *alg* is a mode, then all algorithms of that mode are tested.

```
161 static TPM_RC
162 TestSymmetric(
163     TPM_ALG_ID                   alg,
164     ALGORITHM_VECTOR            *toTest
165     )
166 {
167     SYM_INDEX                    index;
168     TPM_ALG_ID                   mode;
169 //
170     if(!TEST_BIT(alg, *toTest))
171         return TPM_RC_SUCCESS;
172     if(alg == ALG_AES_VALUE || alg == ALG_SM4_VALUE || alg == ALG_CAMELLIA_VALUE)
173     {
```

 Copyright © TCG 2006-2020

```
174             // Will test the algorithm for all modes and key sizes
175             CLEAR_BOTH(alg);
176
177             // A test this algorithm for all modes
178             for(index = 0; index < NUM_SYMS; index++)
179             {
180                 if(c_symTestValues[index].alg == alg)
181                 {
182                     for(mode = TPM_SYM_MODE_FIRST;
183                     mode <= TPM_SYM_MODE_LAST;
184                         mode++)
185                     {
186                         if(TEST_BIT(mode, *toTest))
187                             TestSymmetricAlgorithm(&c_symTestValues[index], mode);
188                     }
189                 }
190             }
191             // if all the symmetric tests are done
192             if(AllSymsAreDone(toTest))
193             {
194                 // all symmetric algorithms tested so no modes should be set
195                 for(alg = TPM_SYM_MODE_FIRST; alg <= TPM_SYM_MODE_LAST; alg++)
196                     CLEAR_BOTH(alg);
197             }
198         }
199         else if(TPM_SYM_MODE_FIRST <= alg && alg <= TPM_SYM_MODE_LAST)
200         {
201             // Test this mode for all key sizes and algorithms
202             for(index = 0; index < NUM_SYMS; index++)
203             {
204                 // The mode testing only comes into play when doing self tests
205                 // by command. When doing self tests by command, the block ciphers are
206                 // tested first. That means that all of their modes would have been
207                 // tested for all key sizes. If there is no block cipher left to
208                 // test, then clear this mode bit.
209                 if(!TEST_BIT(ALG_AES_VALUE, *toTest)
210                    && !TEST_BIT(ALG_SM4_VALUE, *toTest))
211                 {
212                     CLEAR_BOTH(alg);
213                 }
214                 else
215                 {
216                     for(index = 0; index < NUM_SYMS; index++)
217                     {
218                         if(TEST_BIT(c_symTestValues[index].alg, *toTest))
219                             TestSymmetricAlgorithm(&c_symTestValues[index], alg);
220                     }
221                     // have tested this mode for all algorithms
222                     CLEAR_BOTH(alg);
223                 }
224             }
225             if(AllModesAreDone(toTest))
226             {
227                 CLEAR_BOTH(ALG_AES_VALUE);
228                 CLEAR_BOTH(ALG_SM4_VALUE);
229             }
230         }
231         else
232             pAssert(alg == 0 && alg != 0);
233         return TPM_RC_SUCCESS;
234     }
```

10.2.1.5 RSA Tests

```
235     #if ALG_RSA
```


 Copyright © TCG 2006-2020

10.2.1.5.1 Introduction

The tests are for public key only operations and for private key operations. Signature verification and encryption are public key operations. They are tested by using a KVT. For signature verification, this means that a known good signature is checked by CryptRsaValidateSignature(). If it fails, then the TPM enters failure mode. For encryption, the TPM encrypts known values using the selected scheme and checks that the returned value matches the expected value.

For private key operations, a full scheme check is used. For a signing key, a known key is used to sign a known message. Then that signature is verified. since the signature may involve use of random values, the signature will be different each time and we can't always check that the signature matches a known value. The same technique is used for decryption (RSADP/RSAEP).

When an operation uses the public key and the verification has not been tested, the TPM will do a KVT.

The test for the signing algorithm is built into the call for the algorithm

10.2.1.5.2 RsaKeyInitialize()

The test key is defined by a public modulus and a private prime. The TPM's RSA code computes the second prime and the private exponent.

```
236  static void
237  RsaKeyInitialize(
238      OBJECT          *testObject
239      )
240  {
241      MemoryCopy2B(&testObject->publicArea.unique.rsa.b, (P2B)&c_rsaPublicModulus,
242                   sizeof(c_rsaPublicModulus));
243      MemoryCopy2B(&testObject->sensitive.sensitive.rsa.b, (P2B)&c_rsaPrivatePrime,
244                   sizeof(testObject->sensitive.sensitive.rsa.t.buffer));
245      testObject->publicArea.parameters.rsaDetail.keyBits = RSA_TEST_KEY_SIZE * 8;
246      // Use the default exponent
247      testObject->publicArea.parameters.rsaDetail.exponent = 0;
248  }
```

10.2.1.5.3 TestRsaEncryptDecrypt()

These tests are for a public key encryption that uses a random value.

```
249  static TPM_RC
250  TestRsaEncryptDecrypt(
251      TPM_ALG_ID           scheme,            // IN: the scheme
252      ALGORITHM_VECTOR    *toTest             //
253      )
254  {
255      static TPM2B_PUBLIC_KEY_RSA      testInput;
256      static TPM2B_PUBLIC_KEY_RSA      testOutput;
257      static OBJECT                    testObject;
258      const TPM2B_RSA_TEST_KEY        *kvtValue = NULL;
259      TPM_RC                           result = TPM_RC_SUCCESS;
260      const TPM2B                     *testLabel = NULL;
261      TPMT_RSA_DECRYPT                 rsaScheme;
262  //
263      // Don't need to initialize much of the test object
264      RsaKeyInitialize(&testObject);
265      rsaScheme.scheme = scheme;
266      rsaScheme.details.anySig.hashAlg = DEFAULT_TEST_HASH;
267      CLEAR_BOTH(scheme);
268      CLEAR_BOTH(ALG_NULL_VALUE);
269      if(scheme == ALG_NULL_VALUE)
270      {
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
271             // This is an encryption scheme using the private key without any encoding.
272             memcpy(testInput.t.buffer, c_RsaTestValue, sizeof(c_RsaTestValue));
273             testInput.t.size = sizeof(c_RsaTestValue);
274             if(TPM_RC_SUCCESS != CryptRsaEncrypt(&testOutput, &testInput.b,
275                                                  &testObject, &rsaScheme, NULL, NULL))
276                 SELF_TEST_FAILURE;
277             if(!MemoryEqual(testOutput.t.buffer, c_RsaepKvt.buffer, c_RsaepKvt.size))
278                 SELF_TEST_FAILURE;
279             MemoryCopy2B(&testInput.b, &testOutput.b, sizeof(testInput.t.buffer));
280             if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b,
281                                                  &testObject, &rsaScheme, NULL))
282                 SELF_TEST_FAILURE;
283             if(!MemoryEqual(testOutput.t.buffer, c_RsaTestValue,
284                             sizeof(c_RsaTestValue)))
285                 SELF_TEST_FAILURE;
286         }
287         else
288         {
289             // ALG_RSAES_VALUE:
290             // This is an decryption scheme using padding according to
291             // PKCS#1v2.1, 7.2. This padding uses random bits. To test a public
292             // key encryption that uses random data, encrypt a value and then
293             // decrypt the value and see that we get the encrypted data back.
294             // The hash is not used by this encryption so it can be TMP_ALG_NULL
295
296             // ALG_OAEP_VALUE:
297             // This is also an decryption scheme and it also uses a
298             // pseudo-random
299             // value. However, this also uses a hash algorithm. So, we may need
300             // to test that algorithm before use.
301             if(scheme == ALG_OAEP_VALUE)
302             {
303                 TEST_DEFAULT_TEST_HASH(toTest);
304                 kvtValue = &c_OaepKvt;
305                 testLabel = OAEP_TEST_STRING;
306             }
307             else if(scheme == ALG_RSAES_VALUE)
308             {
309                 kvtValue = &c_RsaesKvt;
310                 testLabel = NULL;
311             }
312             else
313                 SELF_TEST_FAILURE;
314             // Only use a digest-size portion of the test value
315             memcpy(testInput.t.buffer, c_RsaTestValue, DEFAULT_TEST_DIGEST_SIZE);
316             testInput.t.size = DEFAULT_TEST_DIGEST_SIZE;
317
318             // See if the encryption works
319             if(TPM_RC_SUCCESS != CryptRsaEncrypt(&testOutput, &testInput.b,
320                                                  &testObject, &rsaScheme, testLabel,
321                                                  NULL))
322                 SELF_TEST_FAILURE;
323             MemoryCopy2B(&testInput.b, &testOutput.b, sizeof(testInput.t.buffer));
324             // see if we can decrypt this value and get the original data back
325             if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b,
326                                                  &testObject, &rsaScheme, testLabel))
327                 SELF_TEST_FAILURE;
328             // See if the results compare
329             if(testOutput.t.size != DEFAULT_TEST_DIGEST_SIZE
330                || !MemoryEqual(testOutput.t.buffer, c_RsaTestValue,
331                                DEFAULT_TEST_DIGEST_SIZE))
332                 SELF_TEST_FAILURE;
333             // Now check that the decryption works on a known value
334             MemoryCopy2B(&testInput.b, (P2B)kvtValue,
335                          sizeof(testInput.t.buffer));
336             if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b,
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
337                                                     &testObject, &rsaScheme, testLabel))
338                SELF_TEST_FAILURE;
339            if(testOutput.t.size != DEFAULT_TEST_DIGEST_SIZE
340               || !MemoryEqual(testOutput.t.buffer, c_RsaTestValue,
341                               DEFAULT_TEST_DIGEST_SIZE))
342                SELF_TEST_FAILURE;
343        }
344        return result;
345    }
```

10.2.1.5.4 TestRsaSignAndVerify()

This function does the testing of the RSA sign and verification functions. This test does a KVT.

```
346    static TPM_RC
347    TestRsaSignAndVerify(
348        TPM_ALG_ID                 scheme,
349        ALGORITHM_VECTOR          *toTest
350        )
351    {
352        TPM_RC                         result = TPM_RC_SUCCESS;
353        static OBJECT                  testObject;
354        static TPM2B_DIGEST            testDigest;
355        static TPMT_SIGNATURE          testSig;
356
357        // Do a sign and signature verification.
358        // RSASSA:
359        // This is a signing scheme according to PKCS#1-v2.1 8.2. It does not
360        // use random data so there is a KVT for the signing operation. On
361        // first use of the scheme for signing, use the TPM's RSA key to
362        // sign a portion of c_RsaTestData and compare the results to c_RsassaKvt. Then
363        // decrypt the data to see that it matches the starting value. This verifies
364        // the signature with a KVT
365
366        // Clear the bits indicating that the function has not been checked. This is to
367        // prevent looping
368        CLEAR_BOTH(scheme);
369        CLEAR_BOTH(ALG_NULL_VALUE);
370        CLEAR_BOTH(ALG_RSA_VALUE);
371
372        RsaKeyInitialize(&testObject);
373        memcpy(testDigest.t.buffer, (BYTE *)c_RsaTestValue, DEFAULT_TEST_DIGEST_SIZE);
374        testDigest.t.size = DEFAULT_TEST_DIGEST_SIZE;
375        testSig.sigAlg = scheme;
376        testSig.signature.rsapss.hash = DEFAULT_TEST_HASH;
377
378        // RSAPSS:
379        // This is a signing scheme a according to PKCS#1-v2.2 8.1 it uses
380        // random data in the signature so there is no KVT for the signing
381        // operation. To test signing, the TPM will use the TPM's RSA key
382        // to sign a portion of c_RsaTestValue and then it will verify the
383        // signature. For verification, c_RsapssKvt is verified before the
384        // user signature blob is verified. The worst case for testing of this
385        // algorithm is two private and one public key operation.
386
387        // The process is to sign known data. If RSASSA is being done, verify that the
388        // signature matches the precomputed value. For both, use the signed value and
389        // see that the verification says that it is a good signature. Then
390        // if testing RSAPSS, do a verify of a known good signature. This ensures that
391        // the validation function works.
392
393        if(TPM_RC_SUCCESS != CryptRsaSign(&testSig, &testObject, &testDigest, NULL))
394            SELF_TEST_FAILURE;
395        // For RSASSA, make sure the results is what we are looking for
```

```
396     if(testSig.sigAlg == ALG_RSASSA_VALUE)
397     {
398         if(testSig.signature.rsassa.sig.t.size != RSA_TEST_KEY_SIZE
399            || !MemoryEqual(c_RsassaKvt.buffer,
400                            testSig.signature.rsassa.sig.t.buffer,
401                            RSA_TEST_KEY_SIZE))
402             SELF_TEST_FAILURE;
403     }
404     // See if the TPM will validate its own signatures
405     if(TPM_RC_SUCCESS != CryptRsaValidateSignature(&testSig, &testObject,
406                                                    &testDigest))
407         SELF_TEST_FAILURE;
408     // If this is RSAPSS, check the verification with known signature
409     // Have to copy because  CrytpRsaValidateSignature() eats the signature
410     if(ALG_RSAPSS_VALUE == scheme)
411     {
412         MemoryCopy2B(&testSig.signature.rsapss.sig.b, (P2B)&c_RsapssKvt,
413                      sizeof(testSig.signature.rsapss.sig.buffer));
414         if(TPM_RC_SUCCESS != CryptRsaValidateSignature(&testSig, &testObject,
415                                                        &testDigest))
416             SELF_TEST_FAILURE;
417     }
418     return result;
419 }
```

10.2.1.5.5 TestRSA()

Function uses the provided vector to indicate which tests to run. It will clear the vector after each test is run and also clear *g_toTest*

```
420 static TPM_RC
421 TestRsa(
422     TPM_ALG_ID                alg,
423     ALGORITHM_VECTOR         *toTest
424     )
425 {
426     TPM_RC                   result = TPM_RC_SUCCESS;
427 //
428     switch(alg)
429     {
430         case ALG_NULL_VALUE:
431         // This is the RSAEP/RSADP function. If we are processing a list, don't
432         // need to test these now because any other test will validate
433         // RSAEP/RSADP. Can tell this is list of test by checking to see if
434         // 'toTest' is pointing at g_toTest. If so, this is an isolated test
435         // an need to go ahead and do the test;
436             if((toTest == &g_toTest)
437                || (!TEST_BIT(ALG_RSASSA_VALUE, *toTest)
438                    && !TEST_BIT(ALG_RSAES_VALUE, *toTest)
439                    && !TEST_BIT(ALG_RSAPSS_VALUE, *toTest)
440                    && !TEST_BIT(ALG_OAEP_VALUE, *toTest)))
441                // Not running a list of tests or no other tests on the list
442                // so run the test now
443                 result = TestRsaEncryptDecrypt(alg, toTest);
444             // if not running the test now, leave the bit on, just in case things
445             // get interrupted
446             break;
447         case ALG_OAEP_VALUE:
448         case ALG_RSAES_VALUE:
449             result = TestRsaEncryptDecrypt(alg, toTest);
450             break;
451         case ALG_RSAPSS_VALUE:
452         case ALG_RSASSA_VALUE:
453             result = TestRsaSignAndVerify(alg, toTest);
```

```
454                 break;
455             default:
456                 SELF_TEST_FAILURE;
457         }
458         return result;
459 }
460 #endif // ALG_RSA
```

10.2.1.6 ECC Tests

```
461 #if ALG_ECC
```

10.2.1.6.1 LoadEccParameter()

This function is mostly for readability and type checking

```
462 static void
463 LoadEccParameter(
464     TPM2B_ECC_PARAMETER         *to,        // target
465     const TPM2B_EC_TEST         *from       // source
466     )
467 {
468     MemoryCopy2B(&to->b, &from->b, sizeof(to->t.buffer));
469 }
```

10.2.1.6.2 LoadEccPoint()

```
470 static void
471 LoadEccPoint(
472     TPMS_ECC_POINT              *point,     // target
473     const TPM2B_EC_TEST         *x,         // source
474     const TPM2B_EC_TEST         *y
475     )
476 {
477     MemoryCopy2B(&point->x.b, (TPM2B *)x, sizeof(point->x.t.buffer));
478     MemoryCopy2B(&point->y.b, (TPM2B *)y, sizeof(point->y.t.buffer));
479 }
```

10.2.1.6.3 TestECDH()

This test does a KVT on a point multiply.

```
480 static TPM_RC
481 TestECDH(
482     TPM_ALG_ID          scheme,         // IN: for consistency
483     ALGORITHM_VECTOR    *toTest         // IN/OUT: modified after test is run
484     )
485 {
486     static TPMS_ECC_POINT       Z;
487     static TPMS_ECC_POINT       Qe;
488     static TPM2B_ECC_PARAMETER  ds;
489     TPM_RC                      result = TPM_RC_SUCCESS;
490 //
491     NOT_REFERENCED(scheme);
492     CLEAR_BOTH(ALG_ECDH_VALUE);
493     LoadEccParameter(&ds, &c_ecTestKey_ds);
494     LoadEccPoint(&Qe, &c_ecTestKey_QeX, &c_ecTestKey_QeY);
495     if(TPM_RC_SUCCESS != CryptEccPointMultiply(&Z, c_testCurve, &Qe, &ds,
496                                                NULL, NULL))
497         SELF_TEST_FAILURE;
498     if(!MemoryEqual2B(&c_ecTestEcdh_X.b, &Z.x.b)
```

```
499         || !MemoryEqual2B(&c_ecTestEcdh_Y.b, &Z.y.b))
500          SELF_TEST_FAILURE;
501     return result;
502 }
```

10.2.1.6.4 TestEccSignAndVerify()

```
503 static TPM_RC
504 TestEccSignAndVerify(
505     TPM_ALG_ID                       scheme,
506     ALGORITHM_VECTOR                *toTest
507     )
508 {
509     static OBJECT                    testObject;
510     static TPMT_SIGNATURE            testSig;
511     static TPMT_ECC_SCHEME           eccScheme;
512
513     testSig.sigAlg = scheme;
514     testSig.signature.ecdsa.hash = DEFAULT_TEST_HASH;
515
516     eccScheme.scheme = scheme;
517     eccScheme.details.anySig.hashAlg = DEFAULT_TEST_HASH;
518
519     CLEAR_BOTH(scheme);
520     CLEAR_BOTH(ALG_ECDH_VALUE);
521
522     // ECC signature verification testing uses a KVT.
523     switch(scheme)
524     {
525         case ALG_ECDSA_VALUE:
526             LoadEccParameter(&testSig.signature.ecdsa.signatureR, &c_TestEcDsa_r);
527             LoadEccParameter(&testSig.signature.ecdsa.signatureS, &c_TestEcDsa_s);
528             break;
529         case ALG_ECSCHNORR_VALUE:
530             LoadEccParameter(&testSig.signature.ecschnorr.signatureR,
531                              &c_TestEcSchnorr_r);
532             LoadEccParameter(&testSig.signature.ecschnorr.signatureS,
533                              &c_TestEcSchnorr_s);
534             break;
535         case ALG_SM2_VALUE:
536             // don't have a test for SM2
537             return TPM_RC_SUCCESS;
538         default:
539             SELF_TEST_FAILURE;
540             break;
541     }
542     TEST_DEFAULT_TEST_HASH(toTest);
543
544     // Have to copy the key. This is because the size used in the test vectors
545     // is the size of the ECC parameter for the test key while the size of a point
546     // is TPM dependent
547     MemoryCopy2B(&testObject.sensitive.sensitive.ecc.b, &c_ecTestKey_ds.b,
548                  sizeof(testObject.sensitive.sensitive.ecc.t.buffer));
549     LoadEccPoint(&testObject.publicArea.unique.ecc, &c_ecTestKey_QsX,
550                  &c_ecTestKey_QsY);
551     testObject.publicArea.parameters.eccDetail.curveID = c_testCurve;
552
553     if(TPM_RC_SUCCESS != CryptEccValidateSignature(&testSig, &testObject,
554                                                   (TPM2B_DIGEST *)&c_ecTestValue.b))
555     {
556         SELF_TEST_FAILURE;
557     }
558     CHECK_CANCELED;
559
560     // Now sign and verify some data
```

 Copyright © TCG 2006-2020

```
561     if(TPM_RC_SUCCESS != CryptEccSign(&testSig, &testObject,
562                                       (TPM2B_DIGEST *)&c_ecTestValue,
563                                       &eccScheme, NULL))
564         SELF_TEST_FAILURE;
565
566     CHECK_CANCELED;
567
568     if(TPM_RC_SUCCESS != CryptEccValidateSignature(&testSig, &testObject,
569                                                    (TPM2B_DIGEST *)&c_ecTestValue))
570         SELF_TEST_FAILURE;
571
572     CHECK_CANCELED;
573
574     return TPM_RC_SUCCESS;
575 }
```

10.2.1.6.5 TestKDFa()

```
576 static TPM_RC
577 TestKDFa(
578     ALGORITHM_VECTOR        *toTest
579     )
580 {
581     static TPM2B_KDF_TEST_KEY   keyOut;
582     UINT32                      counter = 0;
583 //
584     CLEAR_BOTH(ALG_KDF1_SP800_108_VALUE);
585
586     keyOut.t.size = CryptKDFa(KDF_TEST_ALG, &c_kdfTestKeyIn.b, &c_kdfTestLabel.b,
587                               &c_kdfTestContextU.b, &c_kdfTestContextV.b,
588                               TEST_KDF_KEY_SIZE * 8, keyOut.t.buffer,
589                               &counter, FALSE);
590     if (   keyOut.t.size != TEST_KDF_KEY_SIZE
591         || !MemoryEqual(keyOut.t.buffer, c_kdfTestKeyOut.t.buffer,
592                         TEST_KDF_KEY_SIZE))
593         SELF_TEST_FAILURE;
594
595     return TPM_RC_SUCCESS;
596 }
```

10.2.1.6.6 TestEcc()

```
597 static TPM_RC
598 TestEcc(
599     TPM_ALG_ID              alg,
600     ALGORITHM_VECTOR        *toTest
601     )
602 {
603     TPM_RC                  result = TPM_RC_SUCCESS;
604     NOT_REFERENCED(toTest);
605     switch(alg)
606     {
607         case ALG_ECC_VALUE:
608         case ALG_ECDH_VALUE:
609             // If this is in a loop then see if another test is going to deal with
610             // this.
611             // If toTest is not a self-test list
612             if((toTest == &g_toTest)
613                 // or this is the only ECC test in the list
614                || !(TEST_BIT(ALG_ECDSA_VALUE, *toTest)
615                     || TEST_BIT(ALG_ECSCHNORR, *toTest)
616                     || TEST_BIT(ALG_SM2_VALUE, *toTest)))
617             {
618                 result = TestECDH(alg, toTest);
```

```
619                 }
620                 break;
621             case ALG_ECDSA_VALUE:
622             case ALG_ECSCHNORR_VALUE:
623             case ALG_SM2_VALUE:
624                 result = TestEccSignAndVerify(alg, toTest);
625                 break;
626             default:
627                 SELF_TEST_FAILURE;
628                 break;
629         }
630         return result;
631     }
632     #endif // ALG_ECC
```

10.2.1.6.7 TestAlgorithm()

Dispatches to the correct test function for the algorithm or gets a list of testable algorithms.

If *toTest* is not NULL, then the test decisions are based on the algorithm selections in *toTest*. Otherwise, *g_toTest* is used. When bits are clear in *g_toTest* they will also be cleared *toTest*.

If there doesn't happen to be a test for the algorithm, its associated bit is quietly cleared.

If *alg* is zero (TPM_ALG_ERROR), then the *toTest* vector is cleared of any bits for which there is no test (i.e. no tests are actually run but the vector is cleared).

NOTE: *toTest* will only ever have bits set for implemented algorithms but *alg* can be anything.

Error Returns	Meaning
TPM_RC_CANCELED	test was canceled

```
633     LIB_EXPORT
634     TPM_RC
635     TestAlgorithm(
636         TPM_ALG_ID                  alg,
637         ALGORITHM_VECTOR           *toTest
638         )
639     {
640         TPM_ALG_ID                 first = (alg == ALG_ERROR_VALUE) ? ALG_FIRST_VALUE : alg;
641         TPM_ALG_ID                 last = (alg == ALG_ERROR_VALUE) ? ALG_LAST_VALUE : alg;
642         BOOL                       doTest = (alg != ALG_ERROR_VALUE);
643         TPM_RC                     result = TPM_RC_SUCCESS;
644
645         if(toTest == NULL)
646             toTest = &g_toTest;
647
648         // This is kind of strange. This function will either run a test of the selected
649         // algorithm or just clear a bit if there is no test for the algorithm. So,
650         // either this loop will be executed once for the selected algorithm or once for
651         // each of the possible algorithms. If it is executed more than once ('alg' ==
652         // ALG_ERROR), then no test will be run but bits will be cleared for
653         // unimplemented algorithms. This was done this way so that there is only one
654         // case statement with all of the algorithms. It was easier to have one case
655         // statement than to have multiple ones to manage whenever an algorithm ID is
656         // added.
657         for(alg = first; (alg <= last); alg++)
658         {
659             // if 'alg' was TPM_ALG_ERROR, then we will be cycling through
660             // values, some of which may not be implemented. If the bit in toTest
661             // happens to be set, then we could either generated an assert, or just
662             // silently CLEAR it. Decided to just clear.
663             if(!TEST_BIT(alg, g_implementedAlgorithms))
```

```
664             {
665                 CLEAR_BIT(alg, *toTest);
666                 continue;
667             }
668             // Process whatever is left.
669             // NOTE: since this switch will only be called if the algorithm is
670             // implemented, it is not necessary to modify this list except to comment
671             // out the algorithms for which there is no test
672             switch(alg)
673             {
674             // Symmetric block ciphers
675     #if ALG_AES
676                 case ALG_AES_VALUE:
677     #endif  // ALG_AES
678     #if ALG_SM4
679                 // if SM4 is implemented, its test is like other block ciphers but there
680                 // aren't any test vectors for it yet
681     //            case ALG_SM4_VALUE:
682     #endif  // ALG_SM4
683     #if ALG_CAMELLIA
684                 // no test vectors for camellia
685     //            case ALG_CAMELLIA_VALUE:
686     #endif
687             // Symmetric modes
688     #if     !ALG_CFB
689     #   error   CFB is required in all TPM implementations
690     #endif // !ALG_CFB
691                 case ALG_CFB_VALUE:
692                     if(doTest)
693                         result = TestSymmetric(alg, toTest);
694                     break;
695     #if ALG_CTR
696                 case ALG_CTR_VALUE:
697     #endif // ALG_CRT
698     #if ALG_OFB
699                 case ALG_OFB_VALUE:
700     #endif // ALG_OFB
701     #if ALG_CBC
702                 case ALG_CBC_VALUE:
703     #endif // ALG_CBC
704     #if ALG_ECB
705                 case ALG_ECB_VALUE:
706     #endif
707                     if(doTest)
708                         result = TestSymmetric(alg, toTest);
709                     else
710                         // If doing the initialization of g_toTest vector, only need
711                         // to test one of the modes for the symmetric algorithms. If
712                         // initializing for a SelfTest(FULL_TEST), allow all the modes.
713                         if(toTest == &g_toTest)
714                             CLEAR_BIT(alg, *toTest);
715                     break;
716     #if     !ALG_HMAC
717     #   error   HMAC is required in all TPM implementations
718     #endif
719                 case ALG_HMAC_VALUE:
720                     // Clear the bit that indicates that HMAC is required because
721                     // HMAC is used as the basic test for all hash algorithms.
722                     CLEAR_BOTH(alg);
723                     // Testing HMAC means test the default hash
724                     if(doTest)
725                         TestHash(DEFAULT_TEST_HASH, toTest);
726                     else
727                         // If not testing, then indicate that the hash needs to be
728                         // tested because this uses HMAC
729                         SET_BOTH(DEFAULT_TEST_HASH);
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
730                     break;
731     #if ALG_SHA1
732                 case ALG_SHA1_VALUE:
733     #endif // ALG_SHA1
734     #if ALG_SHA256
735                 case ALG_SHA256_VALUE:
736     #endif // ALG_SHA256
737     #if ALG_SHA384
738                 case ALG_SHA384_VALUE:
739     #endif // ALG_SHA384
740     #if ALG_SHA512
741                 case ALG_SHA512_VALUE:
742     #endif // ALG_SHA512
743                 // if SM3 is implemented its test is like any other hash, but there
744                 // aren't any test vectors yet.
745     #if ALG_SM3_256
746     //            case ALG_SM3_256_VALUE:
747     #endif // ALG_SM3_256
748                     if(doTest)
749                         result = TestHash(alg, toTest);
750                     break;
751         // RSA-dependent
752     #if ALG_RSA
753                 case ALG_RSA_VALUE:
754                     CLEAR_BOTH(alg);
755                     if(doTest)
756                         result = TestRsa(ALG_NULL_VALUE, toTest);
757                     else
758                         SET_BOTH(ALG_NULL_VALUE);
759                     break;
760                 case ALG_RSASSA_VALUE:
761                 case ALG_RSAES_VALUE:
762                 case ALG_RSAPSS_VALUE:
763                 case ALG_OAEP_VALUE:
764                 case ALG_NULL_VALUE:     // used or RSADP
765                     if(doTest)
766                         result = TestRsa(alg, toTest);
767                     break;
768     #endif // ALG_RSA
769     #if ALG_KDF1_SP800_108
770                 case ALG_KDF1_SP800_108_VALUE:
771                     if(doTest)
772                         result = TestKDFa(toTest);
773                     break;
774     #endif // ALG_KDF1_SP800_108
775     #if ALG_ECC
776         // ECC dependent but no tests
777         //        case ALG_ECDAA_VALUE:
778         //        case ALG_ECMQV_VALUE:
779         //        case ALG_KDF1_SP800_56a_VALUE:
780         //        case ALG_KDF2_VALUE:
781         //        case ALG_MGF1_VALUE:
782                 case ALG_ECC_VALUE:
783                     CLEAR_BOTH(alg);
784                     if(doTest)
785                         result = TestEcc(ALG_ECDH_VALUE, toTest);
786                     else
787                         SET_BOTH(ALG_ECDH_VALUE);
788                     break;
789                 case ALG_ECDSA_VALUE:
790                 case ALG_ECDH_VALUE:
791                 case ALG_ECSCHNORR_VALUE:
792     //            case ALG_SM2_VALUE:
793                     if(doTest)
794                         result = TestEcc(alg, toTest);
795                     break;
```

 Copyright © TCG 2006-2020

```
796  #endif // ALG_ECC
797              default:
798                  CLEAR_BIT(alg, *toTest);
799                  break;
800          }
801          if(result != TPM_RC_SUCCESS)
802              break;
803      }
804      return result;
805  }
806  #endif // SELF_TESTS
```


 Copyright © TCG 2006-2020

10.2.2 BnConvert.c

10.2.2.1 Introduction

This file contains the basic conversion functions that will convert TPM2B to/from the internal format. The internal format is a *bigNum*,

10.2.2.2 Includes

```
1   #include "Tpm.h"
```

10.2.2.3 Functions

10.2.2.3.1 BnFromBytes()

This function will convert a big-endian byte array to the internal number format. If bn is NULL, then the output is NULL. If bytes is null or the required size is 0, then the output is set to zero

```
2   LIB_EXPORT bigNum
3   BnFromBytes(
4       bigNum              bn,
5       const BYTE         *bytes,
6       NUMBYTES            nBytes
7       )
8   {
9       const BYTE         *pFrom; // 'p' points to the least significant bytes of source
10      BYTE               *pTo;   // points to least significant bytes of destination
11      crypt_uword_t       size;
12  //
13
14      size = (bytes != NULL) ? BYTES_TO_CRYPT_WORDS(nBytes) : 0;
15
16      // If nothing in, nothing out
17      if(bn == NULL)
18          return NULL;
19
20      // make sure things fit
21      pAssert(BnGetAllocated(bn) >= size);
22
23      if(size > 0)
24      {
25          // Clear the topmost word in case it is not filled with data
26          bn->d[size - 1] = 0;
27          // Moving the input bytes from the end of the list (LSB) end
28          pFrom = bytes + nBytes - 1;
29          // To the LS0 of the LSW of the bigNum.
30          pTo = (BYTE *)bn->d;
31          for(; nBytes != 0; nBytes--)
32              *pTo++ = *pFrom--;
33          // For a little-endian machine, the conversion is a straight byte
34          // reversal. For a big-endian machine, we have to put the words in
35          // big-endian byte order
36  #if BIG_ENDIAN_TPM
37          {
38              crypt_word_t   t;
39              for(t = (crypt_word_t)size - 1; t >= 0; t--)
40                  bn->d[t] = SWAP_CRYPT_WORD(bn->d[t]);
41          }
42  #endif
43      }
```

```
44      BnSetTop(bn, size);
45      return bn;
46  }
```

10.2.2.3.2 BnFrom2B()

Convert an TPM2B to a BIG_NUM. If the input value does not exist, or the output does not exist, or the input will not fit into the output the function returns NULL

```
47  LIB_EXPORT bigNum
48  BnFrom2B(
49      bigNum           bn,          // OUT:
50      const TPM2B     *a2B          // IN: number to convert
51      )
52  {
53      if(a2B != NULL)
54          return BnFromBytes(bn, a2B->buffer, a2B->size);
55      // Make sure that the number has an initialized value rather than whatever
56      // was there before
57      BnSetTop(bn, 0);    // Function accepts NULL
58      return NULL;
59  }
```

10.2.2.3.3 BnFromHex()

Convert a hex string into a *bigNum*. This is primarily used in debugging.

```
60  LIB_EXPORT bigNum
61  BnFromHex(
62      bigNum          bn,          // OUT:
63      const char      *hex         // IN:
64      )
65  {
66  #define FromHex(a)  ((a) - (((a) > 'a') ? ('a' + 10)                          \
67                                         : ((a) > 'A') ? ('A' - 10) : '0'))
68      unsigned             i;
69      unsigned             wordCount;
70      const char          *p;
71      BYTE                *d = (BYTE *)&(bn->d[0]);
72  //
73      pAssert(bn && hex);
74      i = (unsigned)strlen(hex);
75      wordCount = BYTES_TO_CRYPT_WORDS((i + 1) / 2);
76      if((i == 0) || (wordCount >= BnGetAllocated(bn)))
77          BnSetWord(bn, 0);
78      else
79      {
80          bn->d[wordCount - 1] = 0;
81          p = hex + i - 1;
82          for(;i > 1; i -= 2)
83          {
84              BYTE a;
85              a = FromHex(*p);
86              p--;
87              *d++ = a + (FromHex(*p) << 4);
88              p--;
89          }
90          if(i == 1)
91              *d = FromHex(*p);
92      }
93  #if !BIG_ENDIAN_TPM
94      for(i = 0; i < wordCount; i++)
95          bn->d[i] = SWAP_CRYPT_WORD(bn->d[i]);
```

```
 96  #endif // BIG_ENDIAN_TPM
 97      BnSetTop(bn, wordCount);
 98      return bn;
 99  }
```

10.2.2.3.4 BnToBytes()

This function converts a BIG_NUM to a byte array. It converts the *bigNum* to a big-endian byte string and sets *size* to the normalized value. If *size* is an input 0, then the receiving buffer is guaranteed to be large enough for the result and the size will be set to the size required for *bigNum* (leading zeros suppressed).

The conversion for a little-endian machine simply requires that all significant bytes of the *bigNum* be reversed. For a big-endian machine, rather than unpack each word individually, the *bigNum* is converted to little-endian words, copied, and then converted back to big-endian.

```
100  LIB_EXPORT BOOL
101  BnToBytes(
102      bigConst             bn,
103      BYTE                *buffer,
104      NUMBYTES            *size           // This the number of bytes that are
105                                          // available in the buffer. The result
106                                          // should be this big.
107      )
108  {
109      crypt_uword_t        requiredSize;
110      BYTE                *pFrom;
111      BYTE                *pTo;
112      crypt_uword_t        count;
113  //
114      // validate inputs
115      pAssert(bn && buffer && size);
116
117      requiredSize = (BnSizeInBits(bn) + 7) / 8;
118      if(requiredSize == 0)
119      {
120          // If the input value is 0, return a byte of zero
121          *size = 1;
122          *buffer = 0;
123      }
124      else
125      {
126  #if BIG_ENDIAN_TPM
127          // Copy the constant input value into a modifiable value
128          BN_VAR(bnL, LARGEST_NUMBER_BITS * 2);
129          BnCopy(bnL, bn);
130          // byte swap the words in the local value to make them little-endian
131          for(count = 0; count < bnL->size; count++)
132              bnL->d[count] = SWAP_CRYPT_WORD(bnL->d[count]);
133          bn = (bigConst)bnL;
134  #endif
135          if(*size == 0)
136              *size = (NUMBYTES)requiredSize;
137          pAssert(requiredSize <= *size);
138          // Byte swap the number (not words but the whole value)
139          count = *size;
140          // Start from the least significant word and offset to the most significant
141          // byte which is in some high word
142          pFrom = (BYTE *)(&bn->d[0]) + requiredSize - 1;
143          pTo = buffer;
144
145          // If the number of output bytes is larger than the number bytes required
146          // for the input number, pad with zeros
147          for(count = *size; count > requiredSize; count--)
148              *pTo++ = 0;
```

```
149             // Move the most significant byte at the end of the BigNum to the next most
150             // significant byte position of the 2B and repeat for all significant bytes.
151             for(; requiredSize > 0; requiredSize--)
152                 *pTo++ = *pFrom--;
153         }
154         return TRUE;
155     }
```

10.2.2.3.5 BnTo2B()

Function to convert a BIG_NUM to TPM2B. The TPM2B size is set to the requested *size* which may require padding. If *size* is non-zero and less than required by the value in *bn* then an error is returned. If *size* is zero, then the TPM2B is assumed to be large enough for the data and a2b->size will be adjusted accordingly.

```
156     LIB_EXPORT BOOL
157     BnTo2B(
158         bigConst         bn,                  // IN:
159         TPM2B           *a2B,                 // OUT:
160         NUMBYTES         size                 // IN: the desired size
161         )
162     {
163         // Set the output size
164         if(bn && a2B)
165         {
166             a2B->size = size;
167             return BnToBytes(bn, a2B->buffer, &a2B->size);
168         }
169         return FALSE;
170     }
171     #if ALG_ECC
```

10.2.2.3.6 BnPointFrom2B()

Function to create a BIG_POINT structure from a 2B point. A point is going to be two ECC values in the same buffer. The values are going to be the size of the modulus. They are in modular form.

```
172     LIB_EXPORT bn_point_t   *
173     BnPointFrom2B(
174         bigPoint             ecP,          // OUT: the preallocated point structure
175         TPMS_ECC_POINT      *p             // IN: the number to convert
176         )
177     {
178         if(p == NULL)
179             return NULL;
180
181         if(NULL != ecP)
182         {
183             BnFrom2B(ecP->x, &p->x.b);
184             BnFrom2B(ecP->y, &p->y.b);
185             BnSetWord(ecP->z, 1);
186         }
187         return ecP;
188     }
```

10.2.2.3.7 BnPointTo2B()

This function converts a BIG_POINT into a TPMS_ECC_POINT. A TPMS_ECC_POINT contains two TPM2B_ECC_PARAMETER values. The maximum size of the parameters is dependent on the maximum

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

EC key size used in an implementation. The presumption is that the TPMS_ECC_POINT is large enough to hold 2 TPM2B values, each as large as a MAX_ECC_PARAMETER_BYTES

```
189   LIB_EXPORT BOOL
190   BnPointTo2B(
191       TPMS_ECC_POINT  *p,             // OUT: the converted 2B structure
192       bigPoint         ecP,           // IN: the values to be converted
193       bigCurve         E              // IN: curve descriptor for the point
194       )
195   {
196       UINT16           size;
197   //
198       pAssert(p && ecP && E);
199       pAssert(BnEqualWord(ecP->z, 1));
200       // BnMsb is the bit number of the MSB. This is one less than the number of bits
201       size = (UINT16)BITS_TO_BYTES(BnSizeInBits(CurveGetOrder(AccessCurveData(E))));
202       BnTo2B(ecP->x, &p->x.b, size);
203       BnTo2B(ecP->y, &p->y.b, size);
204       return TRUE;
205   }
206   #endif // ALG_ECC
```

10.2.3 BnMath.c

10.2.3.1 Introduction

The simulator code uses the canonical form whenever possible in order to make the code in Part 3 more accessible. The canonical data formats are simple and not well suited for complex big number computations. When operating on big numbers, the data format is changed for easier manipulation. The format is native words in little-endian format. As the magnitude of the number decreases, the length of the array containing the number decreases but the starting address doesn't change.

The functions in this file perform simple operations on these big numbers. Only the more complex operations are passed to the underlying support library. Although the support library would have most of these functions, the interface code to convert the format for the values is greater than the size of the code to implement the functions here. So, rather than incur the overhead of conversion, they are done here.

If an implementer would prefer, the underlying library can be used simply by making code substitutions here.

NOTE: There is an intention to continue to augment these functions so that there would be no need to use an external big number library.

Many of these functions have no error returns and will always return TRUE. This is to allow them to be used in **guarded** sequences. That is: OK = OK || BnSomething(s); where the BnSomething() function should not be called if OK isn't true.

10.2.3.2 Includes

```
1  #include "Tpm.h"
```

A constant value of zero as a stand in for NULL *bigNum* values

```
2  const bignum_t   BnConstZero = {1, 0, {0}};
3
4  //** Functions
5
6  //*** AddSame()
7  // Adds two values that are the same size. This function allows 'result' to be
8  // the same as either of the addends. This is a nice function to put into assembly
9  // because handling the carry for multi-precision stuff is not as easy in C
10 // (unless there is a REALLY smart compiler). It would be nice if there were idioms
11 // in a language that a compiler could recognize what is going on and optimize
12 // loops like this.
13 //  Return Type: int
14 //      0           no carry out
15 //      1           carry out
16 static BOOL
17 AddSame(
18     crypt_uword_t           *result,
19     const crypt_uword_t     *op1,
20     const crypt_uword_t     *op2,
21     int                      count
22     )
23 {
24     int         carry = 0;
25     int         i;
26
27     for(i = 0; i < count; i++)
28     {
29         crypt_uword_t        a = op1[i];
30         crypt_uword_t        sum = a + op2[i];
31         result[i] = sum + carry;
```

```
32            // generate a carry if the sum is less than either of the inputs
33            // propagate a carry if there was a carry and the sum + carry is zero
34            // do this using bit operations rather than logical operations so that
35            // the time is about the same.
36            //             propagate term      | generate term
37            carry = ((result[i] == 0) & carry) | (sum < a);
38        }
39        return carry;
40    }
```

10.2.3.2.1 CarryProp()

Propagate a carry

```
41    static int
42    CarryProp(
43        crypt_uword_t           *result,
44        const crypt_uword_t     *op,
45        int                      count,
46        int                      carry
47        )
48    {
49        for(; count; count--)
50            carry = ((*result++ = *op++ + carry) == 0) & carry;
51        return carry;
52    }
53    static void
54    CarryResolve(
55        bigNum          result,
56        int             stop,
57        int             carry
58        )
59    {
60        if(carry)
61        {
62            pAssert((unsigned)stop < result->allocated);
63            result->d[stop++] = 1;
64        }
65        BnSetTop(result, stop);
66    }
```

10.2.3.2.2 BnAdd()

This function adds two *bigNum* values. This function always returns TRUE.

```
67    LIB_EXPORT BOOL
68    BnAdd(
69        bigNum           result,
70        bigConst         op1,
71        bigConst         op2
72        )
73    {
74        crypt_uword_t    stop;
75        int              carry;
76        const bignum_t   *n1 = op1;
77        const bignum_t   *n2 = op2;
78
79    //
80        if(n2->size > n1->size)
81        {
82            n1 = op2;
83            n2 = op1;
84        }
```

```
85      pAssert(result->allocated >= n1->size);
86      stop = MIN(n1->size, n2->allocated);
87      carry = (int)AddSame(result->d, n1->d, n2->d, (int)stop);
88      if(n1->size > stop)
89          carry = CarryProp(&result->d[stop], &n1->d[stop], (int)(n1->size - stop),
carry);
90      CarryResolve(result, (int)n1->size, carry);
91      return TRUE;
92  }
```

10.2.3.2.3 BnAddWord()

This function adds a word value to a *bigNum*. This function always returns TRUE.

```
93  LIB_EXPORT BOOL
94  BnAddWord(
95      bigNum            result,
96      bigConst          op,
97      crypt_uword_t     word
98      )
99  {
100     int               carry;
101 //
102     carry = (result->d[0] = op->d[0] + word) < word;
103     carry = CarryProp(&result->d[1], &op->d[1], (int)(op->size - 1), carry);
104     CarryResolve(result, (int)op->size, carry);
105     return TRUE;
106 }
```

10.2.3.2.4 SubSame()

This function subtracts two values that have the same size.

```
107 static int
108 SubSame(
109     crypt_uword_t           *result,
110     const crypt_uword_t     *op1,
111     const crypt_uword_t     *op2,
112     int                      count
113     )
114 {
115     int                 borrow = 0;
116     int                 i;
117     for(i = 0; i < count; i++)
118     {
119         crypt_uword_t    a = op1[i];
120         crypt_uword_t    diff = a - op2[i];
121         result[i] = diff - borrow;
122         //       generate    |       propagate
123         borrow = (diff > a) | ((diff == 0) & borrow);
124     }
125     return borrow;
126 }
```

10.2.3.2.5 BorrowProp()

This propagates a borrow. If borrow is true when the end of the array is reached, then it means that op2 was larger than op1 and we don't handle that case so an assert is generated. This design choice was made because our only *bigNum* computations are on large positive numbers (primes) or on fields. Propagate a borrow.

 Copyright © TCG 2006-2020

```
127     static int
128     BorrowProp(
129         crypt_uword_t           *result,
130         const crypt_uword_t     *op,
131         int                      size,
132         int                      borrow
133         )
134     {
135         for(; size > 0; size--)
136             borrow = ((*result++ = *op++ - borrow) == MAX_CRYPT_UWORD) && borrow;
137         return borrow;
138     }
```

10.2.3.2.6 BnSub()

This function does subtraction of two *bigNum* values and returns result = op1 - op2 when op1 is greater than op2. If op2 is greater than op1, then a fault is generated. This function always returns TRUE.

```
139     LIB_EXPORT BOOL
140     BnSub(
141         bigNum            result,
142         bigConst          op1,
143         bigConst          op2
144         )
145     {
146         int              borrow;
147         int              stop = (int)MIN(op1->size, op2->allocated);
148     //
149         // Make sure that op2 is not obviously larger than op1
150         pAssert(op1->size >= op2->size);
151         borrow = SubSame(result->d, op1->d, op2->d, stop);
152         if(op1->size > (crypt_uword_t)stop)
153             borrow = BorrowProp(&result->d[stop], &op1->d[stop], (int)(op1->size - stop),
154                                 borrow);
155         pAssert(!borrow);
156         BnSetTop(result, op1->size);
157         return TRUE;
158     }
```

10.2.3.2.7 BnSubWord()

This function subtracts a word value from a *bigNum*. This function always returns TRUE.

```
159     LIB_EXPORT BOOL
160     BnSubWord(
161         bigNum            result,
162         bigConst          op,
163         crypt_uword_t     word
164         )
165     {
166         int              borrow;
167     //
168         pAssert(op->size > 1 || word <= op->d[0]);
169         borrow = word > op->d[0];
170         result->d[0] = op->d[0] - word;
171         borrow = BorrowProp(&result->d[1], &op->d[1], (int)(op->size - 1), borrow);
172         pAssert(!borrow);
173         BnSetTop(result, op->size);
174         return TRUE;
175     }
```

 Copyright © TCG 2006-2020

10.2.3.2.8 BnUnsignedCmp()

This function performs a comparison of op1 to op2. The compare is approximately constant time if the size of the values used in the compare is consistent across calls (from the same line in the calling code).

Return Value	Meaning
0	op1 is less than op2
0	op1 is equal to op2
0	op1 is greater than op2

```
176  LIB_EXPORT int
177  BnUnsignedCmp(
178      bigConst                 op1,
179      bigConst                 op2
180      )
181  {
182      int              retVal;
183      int              diff;
184      int              i;
185  //
186      pAssert((op1 != NULL) && (op2 != NULL));
187      retVal = (int)(op1->size - op2->size);
188      if(retVal == 0)
189      {
190          for(i = (int)(op1->size - 1); i >= 0; i--)
191          {
192              diff = (op1->d[i] < op2->d[i]) ? -1 : (op1->d[i] != op2->d[i]);
193              retVal = retVal == 0 ? diff : retVal;
194          }
195      }
196      else
197          retVal = (retVal < 0) ? -1 : 1;
198      return retVal;
199  }
```

10.2.3.2.9 BnUnsignedCmpWord()

Compare a *bigNum* to a crypt_uword_t.

Return Value	Meaning
-1	op1 is less that word
0	op1 is equal to word
1	op1 is greater than word

```
200  LIB_EXPORT int
201  BnUnsignedCmpWord(
202      bigConst              op1,
203      crypt_uword_t         word
204      )
205  {
206      if(op1->size > 1)
207          return 1;
208      else if(op1->size == 1)
209          return (op1->d[0] < word) ? -1 : (op1->d[0] > word);
210      else // op1 is zero
211          // equal if word is zero
212          return (word == 0) ? 0 : -1;
213  }
```

 Copyright © TCG 2006-2020

10.2.3.2.10 BnModWord()

This function does modular division of a big number when the modulus is a word value.

```
214 LIB_EXPORT crypt_word_t
215 BnModWord(
216     bigConst         numerator,
217     crypt_word_t     modulus
218     )
219 {
220     BN_MAX(remainder);
221     BN_VAR(mod, RADIX_BITS);
222 //
223     mod->d[0] = modulus;
224     mod->size = (modulus != 0);
225     BnDiv(NULL, remainder, numerator, mod);
226     return remainder->d[0];
227 }
```

10.2.3.2.11 Msb()

This function returns the bit number of the most significant bit of a crypt_uword_t. The number for the least significant bit of any *bigNum* value is 0. The maximum return value is RADIX_BITS - 1,

Return Value	Meaning
-1	the word was zero
n	the bit number of the most significant bit in the word

```
228 LIB_EXPORT int
229 Msb(
230     crypt_uword_t           word
231     )
232 {
233     int               retVal = -1;
234 //
235 #if RADIX_BITS == 64
236     if(word & 0xffffffff00000000) { retVal += 32; word >>= 32; }
237 #endif
238     if(word & 0xffff0000) { retVal += 16; word >>= 16; }
239     if(word & 0x0000ff00) { retVal += 8; word >>= 8; }
240     if(word & 0x000000f0) { retVal += 4; word >>= 4; }
241     if(word & 0x0000000c) { retVal += 2; word >>= 2; }
242     if(word & 0x00000002) { retVal += 1; word >>= 1; }
243     return retVal + (int)word;
244 }
```

10.2.3.2.12 BnMsb()

This function returns the number of the MSb of a *bigNum* value.

Return Value	Meaning
-1	the word was zero or *bn* was NULL
n	the bit number of the most significant bit in the word

```
245 LIB_EXPORT int
246 BnMsb(
247     bigConst              bn
248     )
```

 Copyright © TCG 2006-2020

```
249 {
250     // If the value is NULL, or the size is zero then treat as zero and return -1
251     if(bn != NULL && bn->size > 0)
252     {
253         int          retVal = Msb(bn->d[bn->size - 1]);
254         retVal += (int)(bn->size - 1) * RADIX_BITS;
255         return retVal;
256     }
257     else
258         return -1;
259 }
```

10.2.3.2.13 BnSizeInBits()

This function returns the number of bits required to hold a number. It is one greater than the Msb.

```
260 LIB_EXPORT unsigned
261 BnSizeInBits(
262     bigConst                 n
263     )
264 {
265     int     bits = BnMsb(n) + 1;
266 //
267     return bits < 0? 0 : (unsigned)bits;
268 }
```

10.2.3.2.14 BnSetWord()

Change the value of a bignum_t to a word value.

```
269 LIB_EXPORT bigNum
270 BnSetWord(
271     bigNum               n,
272     crypt_uword_t        w
273     )
274 {
275     if(n != NULL)
276     {
277         pAssert(n->allocated > 1);
278         n->d[0] = w;
279         BnSetTop(n, (w != 0) ? 1 : 0);
280     }
281     return n;
282 }
```

10.2.3.2.15 BnSetBit()

This function will SET a bit in a *bigNum*. Bit 0 is the least-significant bit in the 0th digit_t. The function always return TRUE

```
283 LIB_EXPORT BOOL
284 BnSetBit(
285     bigNum           bn,        // IN/OUT: big number to modify
286     unsigned int     bitNum     // IN: Bit number to SET
287     )
288 {
289     crypt_uword_t            offset = bitNum / RADIX_BITS;
290     pAssert(bn->allocated * RADIX_BITS >= bitNum);
291     // Grow the number if necessary to set the bit.
292     while(bn->size <= offset)
293         bn->d[bn->size++] = 0;
```


 Copyright © TCG 2006-2020

```
294       bn->d[offset] |= ((crypt_uword_t)1 << RADIX_MOD(bitNum));
295       return TRUE;
296   }
```

10.2.3.2.16 BnTestBit()

This function is used to check to see if a bit is SET in a bignum_t. The 0th bit is the LSb of d[0].

Return Value	Meaning
TRUE(1)	the bit is set
FALSE(0)	the bit is not set or the number is out of range

```
297   LIB_EXPORT BOOL
298   BnTestBit(
299       bigNum                  bn,         // IN: number to check
300       unsigned int            bitNum      // IN: bit to test
301       )
302   {
303       crypt_uword_t            offset = RADIX_DIV(bitNum);
304   //
305       if(bn->size > offset)
306           return ((bn->d[offset] & (((crypt_uword_t)1) << RADIX_MOD(bitNum))) != 0);
307       else
308           return FALSE;
309   }
```

10.2.3.2.17 BnMaskBits()

This function is used to mask off high order bits of a big number. The returned value will have no more than *maskBit* bits set.

NOTE: There is a requirement that unused words of a bignum_t are set to zero.

Return Value	Meaning
TRUE(1)	result masked
FALSE(0)	the input was not as large as the mask

```
310   LIB_EXPORT BOOL
311   BnMaskBits(
312       bigNum              bn,         // IN/OUT: number to mask
313       crypt_uword_t       maskBit     // IN: the bit number for the mask.
314       )
315   {
316       crypt_uword_t       finalSize;
317       BOOL                retVal;
318
319       finalSize = BITS_TO_CRYPT_WORDS(maskBit);
320       retVal = (finalSize <= bn->allocated);
321       if(retVal && (finalSize > 0))
322       {
323           crypt_uword_t    mask;
324           mask = ~((crypt_uword_t)0) >> RADIX_MOD(maskBit);
325           bn->d[finalSize - 1] &= mask;
326       }
327       BnSetTop(bn, finalSize);
328       return retVal;
329   }
```

10.2.3.2.18 BnShiftRight()

This function will shift a *bigNum* to the right by the *shiftAmount*. This function always returns TRUE.

```
330 LIB_EXPORT BOOL
331 BnShiftRight(
332     bigNum          result,
333     bigConst        toShift,
334     uint32_t        shiftAmount
335     )
336 {
337     uint32_t        offset = (shiftAmount >> RADIX_LOG2);
338     uint32_t        i;
339     uint32_t        shiftIn;
340     crypt_uword_t   finalSize;
341 //
342     shiftAmount = shiftAmount & RADIX_MASK;
343     shiftIn = RADIX_BITS - shiftAmount;
344
345     // The end size is toShift->size - offset less one additional
346     // word if the shiftAmount would make the upper word == 0
347     if(toShift->size > offset)
348     {
349         finalSize = toShift->size - offset;
350         finalSize -= (toShift->d[toShift->size - 1] >> shiftAmount) == 0 ? 1 : 0;
351     }
352     else
353         finalSize = 0;
354
355     pAssert(finalSize <= result->allocated);
356     if(finalSize != 0)
357     {
358         for(i = 0; i < finalSize; i++)
359         {
360             result->d[i] = (toShift->d[i + offset] >> shiftAmount)
361                 | (toShift->d[i + offset + 1] << shiftIn);
362         }
363         if(offset == 0)
364             result->d[i] = toShift->d[i] >> shiftAmount;
365     }
366     BnSetTop(result, finalSize);
367     return TRUE;
368 }
```

10.2.3.2.19 BnGetRandomBits()

This function gets random bits for use in various places. To make sure that the number is generated in a portable format, it is created as a TPM2B and then converted to the internal format.

One consequence of the generation scheme is that, if the number of bits requested is not a multiple of 8, then the high-order bits are set to zero. This would come into play when generating a 521-bit ECC key. A 66-byte (528-bit) value is generated an the high order 7 bits are masked off (CLEAR).

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
369 LIB_EXPORT BOOL
370 BnGetRandomBits(
371     bigNum          n,
372     size_t          bits,
```

 Copyright © TCG 2006-2020

```
373         RAND_STATE       *rand
374     )
375     {
376         // Since this could be used for ECC key generation using the extra bits method,
377         // make sure that the value is large enough
378         TPM2B_TYPE(LARGEST, LARGEST_NUMBER + 8);
379         TPM2B_LARGEST    large;
380     //
381         large.b.size = (UINT16)BITS_TO_BYTES(bits);
382         if(DRBG_Generate(rand, large.t.buffer, large.t.size) == large.t.size)
383         {
384             if(BnFrom2B(n, &large.b) != NULL)
385             {
386                 if(BnMaskBits(n, (crypt_uword_t)bits))
387                     return TRUE;
388             }
389         }
390         return FALSE;
391     }
```

10.2.3.2.20 BnGenerateRandomInRange()

This function is used to generate a random number r in the range 1 <= r < limit. The function gets a random number of bits that is the size of limit. There is some some probability that the returned number is going to be greater than or equal to the limit. If it is, try again. There is no more than 50% chance that the next number is also greater, so try again. We keep trying until we get a value that meets the criteria. Since limit is very often a number with a LOT of high order ones, this rarely would need a second try.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure (*limit* is too small)

```
392     LIB_EXPORT BOOL
393     BnGenerateRandomInRange(
394         bigNum           dest,
395         bigConst         limit,
396         RAND_STATE      *rand
397         )
398     {
399         size_t   bits = BnSizeInBits(limit);
400     //
401         if(bits < 2)
402         {
403             BnSetWord(dest, 0);
404             return FALSE;
405         }
406         else
407         {
408             while(BnGetRandomBits(dest, bits, rand)
409                   && (BnEqualZero(dest) || (BnUnsignedCmp(dest, limit) >= 0)));
410         }
411         return !g_inFailureMode;
412     }
```

10.2.4 BnMemory.c

10.2.4.1 Introduction

This file contains the memory setup functions used by the *bigNum* functions in CryptoEngine()

10.2.4.2 Includes

```
1   #include "Tpm.h"
```

10.2.4.3 Functions

10.2.4.3.1 BnSetTop()

This function is used when the size of a bignum_t is changed. It makes sure that the unused words are set to zero and that any significant words of zeros are eliminated from the used size indicator.

```
2   LIB_EXPORT bigNum
3   BnSetTop(
4       bigNum           bn,         // IN/OUT: number to clean
5       crypt_uword_t    top         // IN: the new top
6       )
7   {
8       if(bn != NULL)
9       {
10          pAssert(top <= bn->allocated);
11          // If forcing the size to be decreased, make sure that the words being
12          // discarded are being set to 0
13          while(bn->size > top)
14              bn->d[--bn->size] = 0;
15          bn->size = top;
16          // Now make sure that the words that are left are 'normalized' (no high-order
17          // words of zero.
18          while((bn->size > 0) && (bn->d[bn->size - 1] == 0))
19              bn->size -= 1;
20      }
21      return bn;
22  }
```

10.2.4.3.2 BnClearTop()

This function will make sure that all unused words are zero.

```
23  LIB_EXPORT bigNum
24  BnClearTop(
25      bigNum          bn
26      )
27  {
28      crypt_uword_t       i;
29  //
30      if(bn != NULL)
31      {
32          for(i = bn->size; i < bn->allocated; i++)
33              bn->d[i] = 0;
34          while((bn->size > 0) && (bn->d[bn->size] == 0))
35              bn->size -= 1;
36      }
37      return bn;
38  }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.2.4.3.3 BnInitializeWord()

This function is used to initialize an allocated *bigNum* with a word value. The *bigNum* does not have to be allocated with a single word.

```
39  LIB_EXPORT bigNum
40  BnInitializeWord(
41      bigNum          bn,         // IN:
42      crypt_uword_t   allocated,  // IN:
43      crypt_uword_t   word        // IN:
44      )
45  {
46      bn->allocated = allocated;
47      bn->size = (word != 0);
48      bn->d[0] = word;
49      while(allocated > 1)
50          bn->d[--allocated] = 0;
51      return bn;
52  }
```

10.2.4.3.4 BnInit()

This function initializes a stack allocated bignum_t. It initializes *allocated* and *size* and zeros the words of *d*.

```
53  LIB_EXPORT bigNum
54  BnInit(
55      bigNum              bn,
56      crypt_uword_t       allocated
57      )
58  {
59      if(bn != NULL)
60      {
61          bn->allocated = allocated;
62          bn->size = 0;
63          while(allocated != 0)
64              bn->d[--allocated] = 0;
65      }
66      return bn;
67  }
```

10.2.4.3.5 BnCopy()

Function to copy a bignum_t. If the output is NULL, then nothing happens. If the input is NULL, the output is set to zero.

```
68  LIB_EXPORT BOOL
69  BnCopy(
70      bigNum          out,
71      bigConst        in
72      )
73  {
74      if(in == out)
75          BnSetTop(out, BnGetSize(out));
76      else if(out != NULL)
77      {
78          if(in != NULL)
79          {
80              unsigned int        i;
81              pAssert(BnGetAllocated(out) >= BnGetSize(in));
82              for(i = 0; i < BnGetSize(in); i++)
83                  out->d[i] = in->d[i];
```

```
84              BnSetTop(out, BnGetSize(in));
85          }
86          else
87              BnSetTop(out, 0);
88      }
89      return TRUE;
90  }
91  #if ALG_ECC
```

10.2.4.3.6 BnPointCopy()

Function to copy a bn point.

```
92  LIB_EXPORT BOOL
93  BnPointCopy(
94      bigPoint                 pOut,
95      pointConst               pIn
96      )
97  {
98      return BnCopy(pOut->x, pIn->x)
99          && BnCopy(pOut->y, pIn->y)
100         && BnCopy(pOut->z, pIn->z);
101 }
```

10.2.4.3.7 BnInitializePoint()

This function is used to initialize a point structure with the addresses of the coordinates.

```
102 LIB_EXPORT bn_point_t *
103 BnInitializePoint(
104     bigPoint             p,     // OUT: structure to receive pointers
105     bigNum               x,     // IN: x coordinate
106     bigNum               y,     // IN: y coordinate
107     bigNum               z      // IN: x coordinate
108     )
109 {
110     p->x = x;
111     p->y = y;
112     p->z = z;
113     BnSetWord(z, 1);
114     return p;
115 }
116 #endif // ALG_ECC
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.2.5 CryptCmac.c

10.2.5.1 Introduction

This file contains the implementation of the message authentication codes based on a symmetric block cipher. These functions only use the single block encryption functions of the selected symmetric cryptographic library.

10.2.5.2 Includes, Defines, and Typedefs

```
1   #define _CRYPT_HASH_C_
2   #include "Tpm.h"
3   #include "CryptSym.h"
4   #if ALG_CMAC
```

10.2.5.3 Functions

10.2.5.3.1 CryptCmacStart()

This is the function to start the CMAC sequence operation. It initializes the dispatch functions for the data and end operations for CMAC and initializes the parameters that are used for the processing of data, including the key, key size and block cipher algorithm.

```
5   UINT16
6   CryptCmacStart(
7       SMAC_STATE          *state,
8       TPMU_PUBLIC_PARMS   *keyParms,
9       TPM_ALG_ID           macAlg,
10      TPM2B               *key
11  )
12  {
13      tpmCmacState_t      *cState = &state->state.cmac;
14      TPMT_SYM_DEF_OBJECT *def = &keyParms->symDetail.sym;
15  //
16      if(macAlg != TPM_ALG_CMAC)
17          return 0;
18      // set up the encryption algorithm and parameters
19      cState->symAlg = def->algorithm;
20      cState->keySizeBits = def->keyBits.sym;
21      cState->iv.t.size = CryptGetSymmetricBlockSize(def->algorithm,
22                                                     def->keyBits.sym);
23      MemoryCopy2B(&cState->symKey.b, key, sizeof(cState->symKey.t.buffer));
24
25      // Set up the dispatch methods for the CMAC
26      state->smacMethods.data = CryptCmacData;
27      state->smacMethods.end = CryptCmacEnd;
28      return cState->iv.t.size;
29  }
```

10.2.5.3.2 CryptCmacData()

This function is used to add data to the CMAC sequence computation. The function will XOR new data into the IV. If the buffer is full, and there is additional input data, the data is encrypted into the IV buffer, the new data is then XOR into the IV. When the data runs out, the function returns without encrypting even if the buffer is full. The last data block of a sequence will not be encrypted until the call to CryptCmacEnd(). This is to allow the proper subkey to be computed and applied before the last block is encrypted.

```
30  void
31  CryptCmacData(
32      SMAC_STATES          *state,
33      UINT32                size,
34      const BYTE           *buffer
35  )
36  {
37      tpmCmacState_t           *cmacState = &state->cmac;
38      TPM_ALG_ID                algorithm = cmacState->symAlg;
39      BYTE                     *key = cmacState->symKey.t.buffer;
40      UINT16                    keySizeInBits = cmacState->keySizeBits;
41      tpmCryptKeySchedule_t     keySchedule;
42      TpmCryptSetSymKeyCall_t   encrypt;
43  //
44      SELECT(ENCRYPT);
45      while(size > 0)
46      {
47          if(cmacState->bcount == cmacState->iv.t.size)
48          {
49              ENCRYPT(&keySchedule, cmacState->iv.t.buffer, cmacState->iv.t.buffer);
50              cmacState->bcount = 0;
51          }
52          for(;(size > 0) && (cmacState->bcount < cmacState->iv.t.size);
53              size--, cmacState->bcount++)
54          {
55              cmacState->iv.t.buffer[cmacState->bcount] ^= *buffer++;
56          }
57      }
58  }
```

10.2.5.3.3 CryptCmacEnd()

This is the completion function for the CMAC. It does padding, if needed, and selects the subkey to be applied before the last block is encrypted.

```
59  UINT16
60  CryptCmacEnd(
61      SMAC_STATES              *state,
62      UINT32                    outSize,
63      BYTE                     *outBuffer
64  )
65  {
66      tpmCmacState_t           *cState = &state->cmac;
67      // Need to set algorithm, key, and keySizeInBits in the local context so that
68      // the SELECT and ENCRYPT macros will work here
69      TPM_ALG_ID                algorithm = cState->symAlg;
70      BYTE                     *key = cState->symKey.t.buffer;
71      UINT16                    keySizeInBits = cState->keySizeBits;
72      tpmCryptKeySchedule_t     keySchedule;
73      TpmCryptSetSymKeyCall_t   encrypt;
74      TPM2B_IV                  subkey = {{0, {0}}};
75      BOOL                      xorVal;
76      UINT16                    i;
77
78      subkey.t.size = cState->iv.t.size;
79      // Encrypt a block of zero
80      SELECT(ENCRYPT);
81      ENCRYPT(&keySchedule, subkey.t.buffer, subkey.t.buffer);
82
83      // shift left by 1 and XOR with 0x0...87 if the MSb was 0
84      xorVal = ((subkey.t.buffer[0] & 0x80) == 0) ? 0 : 0x87;
85      ShiftLeft(&subkey.b);
86      subkey.t.buffer[subkey.t.size - 1] ^= xorVal;
87      // this is a sanity check to make sure that the algorithm is working properly.
```

 Copyright © TCG 2006-2020

```
88       // remove this check when debug is done
89       pAssert(cState->bcount <= cState->iv.t.size);
90       // If the buffer is full then no need to compute subkey 2.
91       if(cState->bcount < cState->iv.t.size)
92       {
93           //Pad the data
94           cState->iv.t.buffer[cState->bcount++] ^= 0x80;
95           // The rest of the data is a pad of zero which would simply be XORed
96           // with the iv value so nothing to do...
97           // Now compute K2
98           xorVal = ((subkey.t.buffer[0] & 0x80) == 0) ? 0 : 0x87;
99           ShiftLeft(&subkey.b);
100          subkey.t.buffer[subkey.t.size - 1] ^= xorVal;
101      }
102      // XOR the subkey into the IV
103      for(i = 0; i < subkey.t.size; i++)
104          cState->iv.t.buffer[i] ^= subkey.t.buffer[i];
105      ENCRYPT(&keySchedule, cState->iv.t.buffer, cState->iv.t.buffer);
106      i = (UINT16)MIN(cState->iv.t.size, outSize);
107      MemoryCopy(outBuffer, cState->iv.t.buffer, i);
108
109      return i;
110  }
111  #endif
  8
```

10.2.6 CryptUtil.c

10.2.6.1 Introduction

This module contains the interfaces to the CryptoEngine() and provides miscellaneous cryptographic functions in support of the TPM.

10.2.6.2 Includes

```
1     #include "Tpm.h"
```

10.2.6.3 Hash/HMAC Functions

10.2.6.3.1 CryptHmacSign()

Sign a digest using an HMAC key. This an HMAC of a digest, not an HMAC of a message.

Error Returns	Meaning
TPM_RC_HASH	not a valid hash

```
2     static TPM_RC
3     CryptHmacSign(
4         TPMT_SIGNATURE       *signature,      // OUT: signature
5         OBJECT               *signKey,        // IN: HMAC key sign the hash
6         TPM2B_DIGEST         *hashData        // IN: hash to be signed
7         )
8     {
9         HMAC_STATE       hmacState;
10        UINT32           digestSize;
11
12        digestSize = CryptHmacStart2B(&hmacState, signature->signature.any.hashAlg,
13                                      &signKey->sensitive.sensitive.bits.b);
14        CryptDigestUpdate2B(&hmacState.hashState, &hashData->b);
15        CryptHmacEnd(&hmacState, digestSize,
16                     (BYTE *)&signature->signature.hmac.digest);
17        return TPM_RC_SUCCESS;
18    }
```

10.2.6.3.2 CryptHMACVerifySignature()

This function will verify a signature signed by a HMAC key. Note that a caller needs to prepare *signature* with the signature algorithm (TPM_ALG_HMAC) and the hash algorithm to use. This function then builds a signature of that type.

Error Returns	Meaning
TPM_RC_SCHEME	not the proper scheme for this key type
TPM_RC_SIGNATURE	if invalid input or signature is not genuine

```
19    static TPM_RC
20    CryptHMACVerifySignature(
21        OBJECT               *signKey,        // IN: HMAC key signed the hash
22        TPM2B_DIGEST         *hashData,       // IN: digest being verified
23        TPMT_SIGNATURE       *signature       // IN: signature to be verified
24        )
25    {
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
26      TPMT_SIGNATURE            test;
27      TPMT_KEYEDHASH_SCHEME    *keyScheme =
28                              &signKey->publicArea.parameters.keyedHashDetail.scheme;
29  //
30      if((signature->sigAlg != ALG_HMAC_VALUE)
31         || (signature->signature.hmac.hashAlg == ALG_NULL_VALUE))
32          return TPM_RC_SCHEME;
33      // This check is not really needed for verification purposes. However, it does
34      // prevent someone from trying to validate a signature using a weaker hash
35      // algorithm than otherwise allowed by the key. That is, a key with a scheme
36      // other than TMP_ALG_NULL can only be used to validate signatures that have
37      // a matching scheme.
38      if((keyScheme->scheme != ALG_NULL_VALUE)
39         && ((keyScheme->scheme != signature->sigAlg)
40             || (keyScheme->details.hmac.hashAlg
41                 != signature->signature.any.hashAlg)))
42          return TPM_RC_SIGNATURE;
43      test.sigAlg = signature->sigAlg;
44      test.signature.hmac.hashAlg = signature->signature.hmac.hashAlg;
45
46      CryptHmacSign(&test, signKey, hashData);
47
48      // Compare digest
49      if(!MemoryEqual(&test.signature.hmac.digest,
50                      &signature->signature.hmac.digest,
51                      CryptHashGetDigestSize(signature->signature.any.hashAlg)))
52          return TPM_RC_SIGNATURE;
53
54      return TPM_RC_SUCCESS;
55  }
```

10.2.6.3.3 CryptGenerateKeyedHash()

This function creates a *keyedHash* object.

Error Returns	Meaning
TPM_RC_NO_RESULT	cannot get values from random number generator
TPM_RC_SIZE	sensitive data size is larger than allowed for the scheme

```
56  static TPM_RC
57  CryptGenerateKeyedHash(
58      TPMT_PUBLIC              *publicArea,        // IN/OUT: the public area template
59                                                   //     for the new key.
60      TPMT_SENSITIVE           *sensitive,         // OUT: sensitive area
61      TPMS_SENSITIVE_CREATE    *sensitiveCreate,   // IN: sensitive creation data
62      RAND_STATE               *rand               // IN: "entropy" source
63      )
64  {
65      TPMT_KEYEDHASH_SCHEME    *scheme;
66      TPM_ALG_ID                hashAlg;
67      UINT16                    digestSize;
68
69      scheme = &publicArea->parameters.keyedHashDetail.scheme;
70
71      if(publicArea->type != ALG_KEYEDHASH_VALUE)
72          return TPM_RC_FAILURE;
73
74      // Pick the limiting hash algorithm
75      if(scheme->scheme == ALG_NULL_VALUE)
76          hashAlg = publicArea->nameAlg;
77      else if(scheme->scheme == ALG_XOR_VALUE)
78          hashAlg = scheme->details.xor.hashAlg;
```

```
79         else
80             hashAlg = scheme->details.hmac.hashAlg;
81         digestSize = CryptHashGetDigestSize(hashAlg);
82
83         // if this is a signing or a decryption key, then the limit
84         // for the data size is the block size of the hash. This limit
85         // is set because larger values have lower entropy because of the
86         // HMAC function. The lower limit is 1/2 the size of the digest
87         //
88         //If the user provided the key, check that it is a proper size
89         if(sensitiveCreate->data.t.size != 0)
90         {
91             if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
92                || IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
93             {
94                 if(sensitiveCreate->data.t.size > CryptHashGetBlockSize(hashAlg))
95                     return TPM_RC_SIZE;
96     #if 0   // May make this a FIPS-mode requirement
97                 if(sensitiveCreate->data.t.size < (digestSize / 2))
98                     return TPM_RC_SIZE;
99     #endif
100            }
101            // If this is a data blob, then anything that will get past the unmarshaling
102            // is OK
103            MemoryCopy2B(&sensitive->sensitive.bits.b, &sensitiveCreate->data.b,
104                         sizeof(sensitive->sensitive.bits.t.buffer));
105        }
106        else
107        {
108            // The TPM is going to generate the data so set the size to be the
109            // size of the digest of the algorithm
110            sensitive->sensitive.bits.t.size =
111                DRBG_Generate(rand, sensitive->sensitive.bits.t.buffer, digestSize);
112            if(sensitive->sensitive.bits.t.size == 0)
113                return (g_inFailureMode) ? TPM_RC_FAILURE : TPM_RC_NO_RESULT;
114        }
115        return TPM_RC_SUCCESS;
116    }
```

10.2.6.3.4 CryptIsSchemeAnonymous()

This function is used to test a scheme to see if it is an anonymous scheme The only anonymous scheme is ECDAA. ECDAA can be used to do things like U-Prove.

```
117    BOOL
118    CryptIsSchemeAnonymous(
119        TPM_ALG_ID          scheme          // IN: the scheme algorithm to test
120        )
121    {
122        return scheme == ALG_ECDAA_VALUE;
123    }
```

10.2.6.4 Symmetric Functions

10.2.6.4.1 ParmDecryptSym()

This function performs parameter decryption using symmetric block cipher.

```
124    void
125    ParmDecryptSym(
126        TPM_ALG_ID          symAlg,         // IN: the symmetric algorithm
127        TPM_ALG_ID          hash,           // IN: hash algorithm for KDFa
```

```
128         UINT16             keySizeInBits, // IN: the key size in bits
129         TPM2B             *key,           // IN: KDF HMAC key
130         TPM2B             *nonceCaller,   // IN: nonce caller
131         TPM2B             *nonceTpm,      // IN: nonce TPM
132         UINT32             dataSize,      // IN: size of parameter buffer
133         BYTE              *data           // OUT: buffer to be decrypted
134         )
135     {
136         // KDF output buffer
137         // It contains parameters for the CFB encryption
138         // From MSB to LSB, they are the key and iv
139         BYTE               symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
140         // Symmetric key size in byte
141         UINT16             keySize = (keySizeInBits + 7) / 8;
142         TPM2B_IV           iv;
143
144         iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
145         // If there is decryption to do...
146         if(iv.t.size > 0)
147         {
148             // Generate key and iv
149             CryptKDFa(hash, key, CFB_KEY, nonceCaller, nonceTpm,
150                       keySizeInBits + (iv.t.size * 8), symParmString, NULL, FALSE);
151             MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size);
152
153             CryptSymmetricDecrypt(data, symAlg, keySizeInBits, symParmString,
154                                   &iv, ALG_CFB_VALUE, dataSize, data);
155         }
156         return;
157     }
```

10.2.6.4.2 ParmEncryptSym()

This function performs parameter encryption using symmetric block cipher.

```
158     void
159     ParmEncryptSym(
160         TPM_ALG_ID         symAlg,        // IN: symmetric algorithm
161         TPM_ALG_ID         hash,          // IN: hash algorithm for KDFa
162         UINT16             keySizeInBits, // IN: symmetric key size in bits
163         TPM2B             *key,           // IN: KDF HMAC key
164         TPM2B             *nonceCaller,   // IN: nonce caller
165         TPM2B             *nonceTpm,      // IN: nonce TPM
166         UINT32             dataSize,      // IN: size of parameter buffer
167         BYTE              *data           // OUT: buffer to be encrypted
168         )
169     {
170         // KDF output buffer
171         // It contains parameters for the CFB encryption
172         BYTE               symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
173
174         // Symmetric key size in bytes
175         UINT16             keySize = (keySizeInBits + 7) / 8;
176
177         TPM2B_IV           iv;
178
179         iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
180         // See if there is any encryption to do
181         if(iv.t.size > 0)
182         {
183             // Generate key and iv
184             CryptKDFa(hash, key, CFB_KEY, nonceTpm, nonceCaller,
185                       keySizeInBits + (iv.t.size * 8), symParmString, NULL, FALSE);
186             MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size);
```

```
187
188             CryptSymmetricEncrypt(data, symAlg, keySizeInBits, symParmString, &iv,
189                                   ALG_CFB_VALUE, dataSize, data);
190         }
191         return;
192     }
```

10.2.6.4.3 CryptGenerateKeySymmetric()

This function generates a symmetric cipher key. The derivation process is determined by the type of the provided *rand*

Error Returns	Meaning
TPM_RC_NO_RESULT	cannot get a random value
TPM_RC_KEY_SIZE	key size in the public area does not match the size in the sensitive creation area
TPM_RC_KEY	provided key value is not allowed

```
193 static TPM_RC
194 CryptGenerateKeySymmetric(
195     TPMT_PUBLIC             *publicArea,        // IN/OUT: The public area template
196                                                 //     for the new key.
197     TPMT_SENSITIVE          *sensitive,         // OUT: sensitive area
198     TPMS_SENSITIVE_CREATE   *sensitiveCreate,   // IN: sensitive creation data
199     RAND_STATE              *rand               // IN: the "entropy" source for
200     )
201 {
202     UINT16           keyBits = publicArea->parameters.symDetail.sym.keyBits.sym;
203     TPM_RC           result;
204 //
205     // only do multiples of RADIX_BITS
206     if((keyBits % RADIX_BITS) != 0)
207         return TPM_RC_KEY_SIZE;
208     // If this is not a new key, then the provided key data must be the right size
209     if(sensitiveCreate->data.t.size != 0)
210     {
211         result = CryptSymKeyValidate(&publicArea->parameters.symDetail.sym,
212                                      (TPM2B_SYM_KEY *)&sensitiveCreate->data);
213         if(result == TPM_RC_SUCCESS)
214             MemoryCopy2B(&sensitive->sensitive.sym.b, &sensitiveCreate->data.b,
215                          sizeof(sensitive->sensitive.sym.t.buffer));
216     }
217 #if ALG_TDES
218     else if(publicArea->parameters.symDetail.sym.algorithm == ALG_TDES_VALUE)
219     {
220         result = CryptGenerateKeyDes(publicArea, sensitive, rand);
221     }
222 #endif
223     else
224     {
225         sensitive->sensitive.sym.t.size =
226             DRBG_Generate(rand, sensitive->sensitive.sym.t.buffer,
227                           BITS_TO_BYTES(keyBits));
228         if(g_inFailureMode)
229             result = TPM_RC_FAILURE;
230         else if(sensitive->sensitive.sym.t.size == 0)
231             result = TPM_RC_NO_RESULT;
232         else
233             result = TPM_RC_SUCCESS;
234     }
235     return result;
236 }
```

10.2.6.4.4 CryptXORObfuscation()

This function implements XOR obfuscation. It should not be called if the hash algorithm is not implemented. The only return value from this function is TPM_RC_SUCCESS.

```
237  void
238  CryptXORObfuscation(
239      TPM_ALG_ID        hash,           // IN: hash algorithm for KDF
240      TPM2B            *key,            // IN: KDF key
241      TPM2B            *contextU,       // IN: contextU
242      TPM2B            *contextV,       // IN: contextV
243      UINT32            dataSize,       // IN: size of data buffer
244      BYTE             *data            // IN/OUT: data to be XORed in place
245      )
246  {
247      BYTE              mask[MAX_DIGEST_SIZE]; // Allocate a digest sized buffer
248      BYTE             *pm;
249      UINT32            i;
250      UINT32            counter = 0;
251      UINT16            hLen = CryptHashGetDigestSize(hash);
252      UINT32            requestSize = dataSize * 8;
253      INT32             remainBytes = (INT32)dataSize;
254
255      pAssert((key != NULL) && (data != NULL) && (hLen != 0));
256
257      // Call KDFa to generate XOR mask
258      for(; remainBytes > 0; remainBytes -= hLen)
259      {
260          // Make a call to KDFa to get next iteration
261          CryptKDFa(hash, key, XOR_KEY, contextU, contextV,
262                    requestSize, mask, &counter, TRUE);
263
264      // XOR next piece of the data
265          pm = mask;
266          for(i = hLen < remainBytes ? hLen : remainBytes; i > 0; i--)
267              *data++ ^= *pm++;
268      }
269      return;
270  }
```

10.2.6.5 Initialization and shut down

10.2.6.5.1 CryptInit()

This function is called when the TPM receives a _TPM_Init() indication.

NOTE: The hash algorithms do not have to be tested, they just need to be available. They have to be tested before the TPM can accept HMAC authorization or return any result that relies on a hash algorithm.

Return Value	Meaning
TRUE(1)	initializations succeeded
FALSE(0)	initialization failed and caller should place the TPM into Failure Mode

```
271  BOOL
272  CryptInit(
273      void
274      )
275  {
276      BOOL          ok;
277      // Initialize the vector of implemented algorithms
```

 Copyright © TCG 2006-2020

```
278     AlgorithmGetImplementedVector(&g_implementedAlgorithms);
279
280     // Indicate that all test are necessary
281     CryptInitializeToTest();
282
283     // Do any library initializations that are necessary. If any fails,
284     // the caller should go into failure mode;
285     ok = SupportLibInit();
286     ok = ok && CryptSymInit();
287     ok = ok && CryptRandInit();
288     ok = ok && CryptHashInit();
289 #if ALG_RSA
290     ok = ok && CryptRsaInit();
291 #endif // ALG_RSA
292 #if ALG_ECC
293     ok = ok && CryptEccInit();
294 #endif // ALG_ECC
295     return ok;
296 }
```

10.2.6.5.2 CryptStartup()

This function is called by TPM2_Startup() to initialize the functions in this cryptographic library and in the provided CryptoLibrary(). This function and CryptUtilInit() are both provided so that the implementation may move the initialization around to get the best interaction.

Return Value	Meaning
TRUE(1)	startup succeeded
FALSE(0)	startup failed and caller should place the TPM into Failure Mode

```
297 BOOL
298 CryptStartup(
299     STARTUP_TYPE     type            // IN: the startup type
300     )
301 {
302     BOOL            OK;
303     NOT_REFERENCED(type);
304
305     OK = CryptSymStartup() && CryptRandStartup() && CryptHashStartup()
306 #if ALG_RSA
307         &&  CryptRsaStartup()
308 #endif // ALG_RSA
309 #if ALG_ECC
310         &&  CryptEccStartup()
311 #endif // ALG_ECC
312         ;
313 #if ALG_ECC
314     // Don't directly check for SU_RESET because that is the default
315     if(OK && (type != SU_RESTART) && (type != SU_RESUME))
316     {
317     // If the shutdown was orderly, then the values recovered from NV will
318     // be OK to use.
319         // Get a new  random commit nonce
320         gr.commitNonce.t.size = sizeof(gr.commitNonce.t.buffer);
321         CryptRandomGenerate(gr.commitNonce.t.size, gr.commitNonce.t.buffer);
322         // Reset the counter and commit array
323         gr.commitCounter = 0;
324         MemorySet(gr.commitArray, 0, sizeof(gr.commitArray));
325     }
326 #endif // ALG_ECC
327     return OK;
328 }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.2.6.6 Algorithm-Independent Functions

10.2.6.6.1 Introduction

These functions are used generically when a function of a general type (e.g., symmetric encryption) is required. The functions will modify the parameters as required to interface to the indicated algorithms.

10.2.6.6.2 CryptIsAsymAlgorithm()

This function indicates if an algorithm is an asymmetric algorithm.

Return Value	Meaning
TRUE(1)	if it is an asymmetric algorithm
FALSE(0)	if it is not an asymmetric algorithm

```
329  BOOL
330  CryptIsAsymAlgorithm(
331      TPM_ALG_ID       algID          // IN: algorithm ID
332      )
333  {
334      switch(algID)
335      {
336  #if ALG_RSA
337          case ALG_RSA_VALUE:
338  #endif
339  #if ALG_ECC
340          case ALG_ECC_VALUE:
341  #endif
342              return TRUE;
343              break;
344          default:
345              break;
346      }
347      return FALSE;
348  }
```

10.2.6.6.3 CryptSecretEncrypt()

This function creates a secret value and its associated secret structure using an asymmetric algorithm.

This function is used by TPM2_Rewrap() TPM2_MakeCredential(), and TPM2_Duplicate().

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*keyHandle* does not reference a valid decryption key
TPM_RC_KEY	invalid ECC key (public point is not on the curve)
TPM_RC_SCHEME	RSA key with an unsupported padding scheme
TPM_RC_VALUE	numeric value of the data to be decrypted is greater than the RSA key modulus

```
349  TPM_RC
350  CryptSecretEncrypt(
351      OBJECT                  *encryptKey,     // IN: encryption key object
352      const TPM2B             *label,          // IN: a null-terminated string as L
353      TPM2B_DATA              *data,           // OUT: secret value
354      TPM2B_ENCRYPTED_SECRET  *secret          // OUT: secret structure
355      )
```

 Copyright © TCG 2006-2020

```
356 {
357     TPMT_RSA_DECRYPT         scheme;
358     TPM_RC                   result = TPM_RC_SUCCESS;
359 //
360     if(data == NULL || secret == NULL)
361         return TPM_RC_FAILURE;
362
363     // The output secret value has the size of the digest produced by the nameAlg.
364     data->t.size = CryptHashGetDigestSize(encryptKey->publicArea.nameAlg);
365     // The encryption scheme is OAEP using the nameAlg of the encrypt key.
366     scheme.scheme = ALG_OAEP_VALUE;
367     scheme.details.anySig.hashAlg = encryptKey->publicArea.nameAlg;
368
369     if(!IS_ATTRIBUTE(encryptKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
370         return TPM_RC_ATTRIBUTES;
371     switch(encryptKey->publicArea.type)
372     {
373 #if ALG_RSA
374         case ALG_RSA_VALUE:
375         {
376             // Create secret data from RNG
377             CryptRandomGenerate(data->t.size, data->t.buffer);
378
379             // Encrypt the data by RSA OAEP into encrypted secret
380             result = CryptRsaEncrypt((TPM2B_PUBLIC_KEY_RSA *)secret, &data->b,
381                                      encryptKey, &scheme, label, NULL);
382         }
383         break;
384 #endif // ALG_RSA
385
386 #if ALG_ECC
387         case ALG_ECC_VALUE:
388         {
389             TPMS_ECC_POINT      eccPublic;
390             TPM2B_ECC_PARAMETER eccPrivate;
391             TPMS_ECC_POINT      eccSecret;
392             BYTE                *buffer = secret->t.secret;
393
394             // Need to make sure that the public point of the key is on the
395             // curve defined by the key.
396             if(!CryptEccIsPointOnCurve(
397                 encryptKey->publicArea.parameters.eccDetail.curveID,
398                 &encryptKey->publicArea.unique.ecc))
399                 result = TPM_RC_KEY;
400             else
401             {
402                 // Call crypto engine to create an auxiliary ECC key
403                 // We assume crypt engine initialization should always success.
404                 // Otherwise, TPM should go to failure mode.
405
406                 CryptEccNewKeyPair(&eccPublic, &eccPrivate,
407                                    encryptKey->publicArea.parameters.eccDetail.curveID);
408                 // Marshal ECC public to secret structure. This will be used by the
409                 // recipient to decrypt the secret with their private key.
410                 secret->t.size = TPMS_ECC_POINT_Marshal(&eccPublic, &buffer, NULL);
411
412                 // Compute ECDH shared secret which is R = [d]Q where d is the
413                 // private part of the ephemeral key and Q is the public part of a
414                 // TPM key. TPM_RC_KEY error return from CryptComputeECDHSecret
415                 // because the auxiliary ECC key is just created according to the
416                 // parameters of input ECC encrypt key.
417                 if(CryptEccPointMultiply(&eccSecret,
418                                 encryptKey->publicArea.parameters.eccDetail.curveID,
419                                 &encryptKey->publicArea.unique.ecc, &eccPrivate,
420                                 NULL, NULL)
421                    != TPM_RC_SUCCESS)
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
422                         result = TPM_RC_KEY;
423                     else
424                     {
425                         // The secret value is computed from Z using KDFe as:
426                         // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
427                         // Where:
428                         //  HashID  the nameAlg of the decrypt key
429                         //  Z    the x coordinate (Px) of the product (P) of the point
430                         //       (Q) of the secret and the private x coordinate (de,V)
431                         //       of the decryption key
432                         //  Use a null-terminated string containing "SECRET"
433                         //  PartyUInfo  the x coordinate of the point in the secret
434                         //              (Qe,U )
435                         //  PartyVInfo  the x coordinate of the public key (Qs,V )
436                         //  bits    the number of bits in the digest of HashID
437                         // Retrieve seed from KDFe
438                         CryptKDFe(encryptKey->publicArea.nameAlg, &eccSecret.x.b,
439                                   label, &eccPublic.x.b,
440                                   &encryptKey->publicArea.unique.ecc.x.b,
441                                   data->t.size * 8, data->t.buffer);
442                     }
443                 }
444             }
445             break;
446 #endif // ALG_ECC
447         default:
448             FAIL(FATAL_ERROR_INTERNAL);
449             break;
450     }
451     return result;
452 }
```

10.2.6.6.4 CryptSecretDecrypt()

Decrypt a secret value by asymmetric (or symmetric) algorithm This function is used for ActivateCredential() and Import for asymmetric decryption, and StartAuthSession() for both asymmetric and symmetric decryption process

Error Returns	Meaning
TPM_RC_ATTRIBUTES	RSA key is not a decryption key
TPM_RC_BINDING	Invalid RSA key (public and private parts are not cryptographically bound.
TPM_RC_ECC_POINT	ECC point in the secret is not on the curve
TPM_RC_INSUFFICIENT	failed to retrieve ECC point from the secret
TPM_RC_NO_RESULT	multiplication resulted in ECC point at infinity
TPM_RC_SIZE	data to decrypt is not of the same size as RSA key
TPM_RC_VALUE	For RSA key, numeric value of the encrypted data is greater than the modulus, or the recovered data is larger than the output buffer. For *keyedHash* or symmetric key, the secret is larger than the size of the digest produced by the name algorithm.
TPM_RC_FAILURE	internal error

```
453 TPM_RC
454 CryptSecretDecrypt(
455     OBJECT                  *decryptKey,     // IN: decrypt key
456     TPM2B_NONCE             *nonceCaller,    // IN: nonceCaller.  It is needed for
457                                              //     symmetric decryption.  For
458                                              //     asymmetric decryption, this
```

```
459                                                 //      parameter is NULL
460     const TPM2B                 *label,         // IN: a value for L
461     TPM2B_ENCRYPTED_SECRET      *secret,        // IN: input secret
462     TPM2B_DATA                  *data           // OUT: decrypted secret value
463     )
464 {
465     TPM_RC      result = TPM_RC_SUCCESS;
466
467     // Decryption for secret
468     switch(decryptKey->publicArea.type)
469     {
470 #if ALG_RSA
471         case ALG_RSA_VALUE:
472         {
473             TPMT_RSA_DECRYPT        scheme;
474             TPMT_RSA_SCHEME         *keyScheme
475                 = &decryptKey->publicArea.parameters.rsaDetail.scheme;
476             UINT16                   digestSize;
477
478             scheme = *(TPMT_RSA_DECRYPT *)keyScheme;
479             // If the key scheme is ALG_NULL_VALUE, set the scheme to OAEP and
480             // set the algorithm to the name algorithm.
481             if(scheme.scheme == ALG_NULL_VALUE)
482             {
483                 // Use OAEP scheme
484                 scheme.scheme = ALG_OAEP_VALUE;
485                 scheme.details.oaep.hashAlg = decryptKey->publicArea.nameAlg;
486             }
487             // use the digestSize as an indicator of whether or not the scheme
488             // is using a supported hash algorithm.
489             // Note: depending on the scheme used for encryption, a hashAlg might
490             // not be needed. However, the return value has to have some upper
491             // limit on the size. In this case, it is the size of the digest of the
492             // hash algorithm. It is checked after the decryption is done but, there
493             // is no point in doing the decryption if the size is going to be
494             // 'wrong' anyway.
495             digestSize = CryptHashGetDigestSize(scheme.details.oaep.hashAlg);
496             if(scheme.scheme != ALG_OAEP_VALUE || digestSize == 0)
497                 return TPM_RC_SCHEME;
498
499             // Set the output buffer capacity
500             data->t.size = sizeof(data->t.buffer);
501
502             // Decrypt seed by RSA OAEP
503             result = CryptRsaDecrypt(&data->b, &secret->b,
504                                      decryptKey, &scheme, label);
505             if((result == TPM_RC_SUCCESS) && (data->t.size > digestSize))
506                 result = TPM_RC_VALUE;
507         }
508         break;
509 #endif // ALG_RSA
510 #if ALG_ECC
511         case ALG_ECC_VALUE:
512         {
513             TPMS_ECC_POINT       eccPublic;
514             TPMS_ECC_POINT       eccSecret;
515             BYTE                *buffer = secret->t.secret;
516             INT32                size = secret->t.size;
517
518             // Retrieve ECC point from secret buffer
519             result = TPMS_ECC_POINT_Unmarshal(&eccPublic, &buffer, &size);
520             if(result == TPM_RC_SUCCESS)
521             {
522                 result = CryptEccPointMultiply(&eccSecret,
523                               decryptKey->publicArea.parameters.eccDetail.curveID,
524                               &eccPublic, &decryptKey->sensitive.sensitive.ecc,
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
525                             NULL, NULL);
526                 if(result == TPM_RC_SUCCESS)
527                 {
528                     // Set the size of the "recovered" secret value to be the size
529                     // of the digest produced by the nameAlg.
530                     data->t.size =
531                         CryptHashGetDigestSize(decryptKey->publicArea.nameAlg);
532
533                 // The secret value is computed from Z using KDFe as:
534                 // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
535                 // Where:
536                 //  HashID -- the nameAlg of the decrypt key
537                 //  Z --  the x coordinate (Px) of the product (P) of the point
538                 //        (Q) of the secret and the private x coordinate (de,V)
539                 //        of the decryption key
540                 //  Use -- a null-terminated string containing "SECRET"
541                 //  PartyUInfo -- the x coordinate of the point in the secret
542                 //              (Qe,U )
543                 //  PartyVInfo -- the x coordinate of the public key (Qs,V )
544                 //  bits -- the number of bits in the digest of HashID
545                 // Retrieve seed from KDFe
546                     CryptKDFe(decryptKey->publicArea.nameAlg, &eccSecret.x.b, label,
547                               &eccPublic.x.b,
548                               &decryptKey->publicArea.unique.ecc.x.b,
549                               data->t.size * 8, data->t.buffer);
550                 }
551             }
552         }
553         break;
554 #endif // ALG_ECC
555 #if     !ALG_KEYEDHASH
556 #   error   "KEYEDHASH support is required"
557 #endif
558         case ALG_KEYEDHASH_VALUE:
559             // The seed size can not be bigger than the digest size of nameAlg
560             if(secret->t.size >
561                CryptHashGetDigestSize(decryptKey->publicArea.nameAlg))
562                 result = TPM_RC_VALUE;
563             else
564             {
565                 // Retrieve seed by XOR Obfuscation:
566                 //    seed = XOR(secret, hash, key, nonceCaller, nullNonce)
567                 //    where:
568                 //    secret  the secret parameter from the TPM2_StartAuthHMAC
569                 //            command that contains the seed value
570                 //    hash    nameAlg  of tpmKey
571                 //    key     the key or data value in the object referenced by
572                 //            entityHandle in the TPM2_StartAuthHMAC command
573                 //    nonceCaller the parameter from the TPM2_StartAuthHMAC command
574                 //    nullNonce   a zero-length nonce
575                 // XOR Obfuscation in place
576                 CryptXORObfuscation(decryptKey->publicArea.nameAlg,
577                                     &decryptKey->sensitive.sensitive.bits.b,
578                                     &nonceCaller->b, NULL,
579                                     secret->t.size, secret->t.secret);
580                 // Copy decrypted seed
581                 MemoryCopy2B(&data->b, &secret->b, sizeof(data->t.buffer));
582             }
583             break;
584         case ALG_SYMCIPHER_VALUE:
585         {
586             TPM2B_IV                iv = {{0}};
587             TPMT_SYM_DEF_OBJECT     *symDef;
588             // The seed size can not be bigger than the digest size of nameAlg
589             if(secret->t.size >
590                CryptHashGetDigestSize(decryptKey->publicArea.nameAlg))
```

```
591                 result = TPM_RC_VALUE;
592             else
593             {
594                 symDef = &decryptKey->publicArea.parameters.symDetail.sym;
595                 iv.t.size = CryptGetSymmetricBlockSize(symDef->algorithm,
596                                                        symDef->keyBits.sym);
597                 if(iv.t.size == 0)
598                     return TPM_RC_FAILURE;
599                 if(nonceCaller->t.size >= iv.t.size)
600                 {
601                     MemoryCopy(iv.t.buffer, nonceCaller->t.buffer, iv.t.size);
602                 }
603                 else
604                 {
605                     if(nonceCaller->t.size > sizeof(iv.t.buffer))
606                         return TPM_RC_FAILURE;
607                     MemoryCopy(iv.b.buffer, nonceCaller->t.buffer,
608                                       nonceCaller->t.size);
609                 }
610                 // make sure secret will fit
611                 if(secret->t.size > data->t.size)
612                     return TPM_RC_FAILURE;
613                 data->t.size = secret->t.size;
614                 // CFB decrypt, using nonceCaller as iv
615                 CryptSymmetricDecrypt(data->t.buffer, symDef->algorithm,
616                                       symDef->keyBits.sym,
617                                       decryptKey->sensitive.sensitive.sym.t.buffer,
618                                       &iv, ALG_CFB_VALUE, secret->t.size,
619                                       secret->t.secret);
620             }
621         }
622         break;
623         default:
624             FAIL(FATAL_ERROR_INTERNAL);
625             break;
626     }
627     return result;
628 }
```

10.2.6.6.5 CryptParameterEncryption()

This function does in-place encryption of a response parameter.

```
629 void
630 CryptParameterEncryption(
631     TPM_HANDLE       handle,            // IN: encrypt session handle
632     TPM2B           *nonceCaller,       // IN: nonce caller
633     UINT16           leadingSizeInByte, // IN: the size of the leading size field in
634                                         //     bytes
635     TPM2B_AUTH      *extraKey,          // IN: additional key material other than
636                                         //     sessionAuth
637     BYTE            *buffer             // IN/OUT: parameter buffer to be encrypted
638     )
639 {
640     SESSION     *session = SessionGet(handle);  // encrypt session
641     TPM2B_TYPE(TEMP_KEY, (sizeof(extraKey->t.buffer)
642                           + sizeof(session->sessionKey.t.buffer)));
643     TPM2B_TEMP_KEY        key;               // encryption key
644     UINT32               cipherSize = 0;    // size of cipher text
645 //
646     // Retrieve encrypted data size.
647     if(leadingSizeInByte == 2)
648     {
649         // Extract the first two bytes as the size field as the data size
```

```
650         // encrypt
651         cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
652         // advance the buffer
653         buffer = &buffer[2];
654     }
655 #ifdef      TPM4B
656     else if(leadingSizeInByte == 4)
657     {
658         // use the first four bytes to indicate the number of bytes to encrypt
659         cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
660         //advance pointer
661         buffer = &buffer[4];
662     }
663 #endif
664     else
665     {
666         FAIL(FATAL_ERROR_INTERNAL);
667     }
668
669     // Compute encryption key by concatenating sessionKey with extra key
670     MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
671     MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
672
673     if(session->symmetric.algorithm == ALG_XOR_VALUE)
674
675         // XOR parameter encryption formulation:
676         //    XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
677         CryptXORObfuscation(session->authHashAlg, &(key.b),
678                             &(session->nonceTPM.b),
679                             nonceCaller, cipherSize, buffer);
680     else
681         ParmEncryptSym(session->symmetric.algorithm, session->authHashAlg,
682                        session->symmetric.keyBits.aes, &(key.b),
683                        nonceCaller, &(session->nonceTPM.b),
684                        cipherSize, buffer);
685     return;
686 }
```

10.2.6.6.6 CryptParameterDecryption()

This function does in-place decryption of a command parameter.

Error Returns	Meaning
TPM_RC_SIZE	The number of bytes in the input buffer is less than the number of bytes to be decrypted.

```
687 TPM_RC
688 CryptParameterDecryption(
689     TPM_HANDLE       handle,             // IN: encrypted session handle
690     TPM2B           *nonceCaller,        // IN: nonce caller
691     UINT32           bufferSize,         // IN: size of parameter buffer
692     UINT16           leadingSizeInByte,  // IN: the size of the leading size field in
693                                          //     byte
694     TPM2B_AUTH      *extraKey,           // IN: the authValue
695     BYTE            *buffer              // IN/OUT: parameter buffer to be decrypted
696     )
697 {
698     SESSION         *session = SessionGet(handle);  // encrypt session
699     // The HMAC key is going to be the concatenation of the session key and any
700     // additional key material (like the authValue). The size of both of these
701     // is the size of the buffer which can contain a TPMT_HA.
702     TPM2B_TYPE(HMAC_KEY, (sizeof(extraKey->t.buffer)
703                          + sizeof(session->sessionKey.t.buffer)));
```

```
704     TPM2B_HMAC_KEY          key;            // decryption key
705     UINT32                  cipherSize = 0; // size of cipher text
706 //
707     // Retrieve encrypted data size.
708     if(leadingSizeInByte == 2)
709     {
710         // The first two bytes of the buffer are the size of the
711         // data to be decrypted
712         cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
713         buffer = &buffer[2];    // advance the buffer
714     }
715 #ifdef  TPM4B
716     else if(leadingSizeInByte == 4)
717     {
718         // the leading size is four bytes so get the four byte size field
719         cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
720         buffer = &buffer[4];    //advance pointer
721     }
722 #endif
723     else
724     {
725         FAIL(FATAL_ERROR_INTERNAL);
726     }
727     if(cipherSize > bufferSize)
728         return TPM_RC_SIZE;
729
730     // Compute decryption key by concatenating sessionAuth with extra input key
731     MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
732     MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
733
734     if(session->symmetric.algorithm == ALG_XOR_VALUE)
735         // XOR parameter decryption formulation:
736         //    XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
737         // Call XOR obfuscation function
738         CryptXORObfuscation(session->authHashAlg, &key.b, nonceCaller,
739                             &(session->nonceTPM.b), cipherSize, buffer);
740     else
741         // Assume that it is one of the symmetric block ciphers.
742         ParmDecryptSym(session->symmetric.algorithm, session->authHashAlg,
743                        session->symmetric.keyBits.sym,
744                        &key.b, nonceCaller, &session->nonceTPM.b,
745                        cipherSize, buffer);
746
747     return TPM_RC_SUCCESS;
748 }
```

10.2.6.6.7 CryptComputeSymmetricUnique()

This function computes the unique field in public area for symmetric objects.

```
749 void
750 CryptComputeSymmetricUnique(
751     TPMT_PUBLIC     *publicArea,    // IN: the object's public area
752     TPMT_SENSITIVE  *sensitive,     // IN: the associated sensitive area
753     TPM2B_DIGEST    *unique         // OUT: unique buffer
754     )
755 {
756     // For parents (symmetric and derivation), use an HMAC to compute
757     // the 'unique' field
758     if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
759        && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt))
760     {
761         // Unique field is HMAC(sensitive->seedValue, sensitive->sensitive)
762         HMAC_STATE      hmacState;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
763             unique->b.size = CryptHmacStart2B(&hmacState, publicArea->nameAlg,
764                                               &sensitive->seedValue.b);
765             CryptDigestUpdate2B(&hmacState.hashState,
766                                 &sensitive->sensitive.any.b);
767             CryptHmacEnd2B(&hmacState, &unique->b);
768         }
769         else
770         {
771             HASH_STATE  hashState;
772             // Unique := Hash(sensitive->seedValue || sensitive->sensitive)
773             unique->t.size = CryptHashStart(&hashState, publicArea->nameAlg);
774             CryptDigestUpdate2B(&hashState, &sensitive->seedValue.b);
775             CryptDigestUpdate2B(&hashState, &sensitive->sensitive.any.b);
776             CryptHashEnd2B(&hashState, &unique->b);
777         }
778         return;
779     }
```

10.2.6.6.8 CryptCreateObject()

This function creates an object. For an asymmetric key, it will create a key pair and, for a parent key, a seed value for child protections.

For an symmetric object, (TPM_ALG_SYMCIPHER or TPM_ALG_KEYEDHASH), it will create a secret key if the caller did not provide one. It will create a random secret seed value that is hashed with the secret value to create the public unique value.

publicArea, *sensitive*, and *sensitiveCreate* are the only required parameters and are the only ones that are used by TPM2_Create(). The other parameters are optional and are used when the generated Object needs to be deterministic. This is the case for both Primary Objects and Derived Objects.

When a seed value is provided, a RAND_STATE will be populated and used for all operations in the object generation that require a random number. In the simplest case, TPM2_CreatePrimary() will use *seed*, *label* and *context* with context being the hash of the template. If the Primary Object is in the Endorsement hierarchy, it will also populate *proof* with *ehProof*.

For derived keys, *seed* will be the secret value from the parent, *label* and *context* will be set according to the parameters of TPM2_CreateLoaded() and *hashAlg* will be set which causes the RAND_STATE to be a KDF generator.

Error Returns	Meaning
TPM_RC_KEY	a provided key is not an allowed value
TPM_RC_KEY_SIZE	key size in the public area does not match the size in the sensitive creation area for a symmetric key
TPM_RC_NO_RESULT	unable to get random values (only in derivation)
TPM_RC_RANGE	for an RSA key, the exponent is not supported
TPM_RC_SIZE	sensitive data size is larger than allowed for the scheme for a keyed hash object
TPM_RC_VALUE	exponent is not prime or could not find a prime using the provided parameters for an RSA key; unsupported name algorithm for an ECC key

```
780     TPM_RC
781     CryptCreateObject(
782         OBJECT                  *object,            // IN: new object structure pointer
783         TPMS_SENSITIVE_CREATE   *sensitiveCreate,   // IN: sensitive creation
784         RAND_STATE              *rand               // IN: the random number generator
785                                                     //      to use
786         )
```

```
787 {
788     TPMT_PUBLIC             *publicArea = &object->publicArea;
789     TPMT_SENSITIVE          *sensitive = &object->sensitive;
790     TPM_RC                   result = TPM_RC_SUCCESS;
791 //
792     // Set the sensitive type for the object
793     sensitive->sensitiveType = publicArea->type;
794
795     // For all objects, copy the initial authorization data
796     sensitive->authValue = sensitiveCreate->userAuth;
797
798     // If the TPM is the source of the data, set the size of the provided data to
799     // zero so that there's no confusion about what to do.
800     if(IS_ATTRIBUTE(publicArea->objectAttributes,
801                     TPMA_OBJECT, sensitiveDataOrigin))
802         sensitiveCreate->data.t.size = 0;
803
804     // Generate the key and unique fields for the asymmetric keys and just the
805     // sensitive value for symmetric object
806     switch(publicArea->type)
807     {
808 #if ALG_RSA
809         // Create RSA key
810         case ALG_RSA_VALUE:
811             // RSA uses full object so that it has a place to put the private
812             // exponent
813             result = CryptRsaGenerateKey(publicArea, sensitive, rand);
814             break;
815 #endif // ALG_RSA
816
817 #if ALG_ECC
818         // Create ECC key
819         case ALG_ECC_VALUE:
820             result = CryptEccGenerateKey(publicArea, sensitive, rand);
821             break;
822 #endif // ALG_ECC
823         case ALG_SYMCIPHER_VALUE:
824             result = CryptGenerateKeySymmetric(publicArea, sensitive,
825                                                sensitiveCreate, rand);
826             break;
827         case ALG_KEYEDHASH_VALUE:
828             result = CryptGenerateKeyedHash(publicArea, sensitive,
829                                             sensitiveCreate, rand);
830             break;
831         default:
832             FAIL(FATAL_ERROR_INTERNAL);
833             break;
834     }
835     if(result != TPM_RC_SUCCESS)
836         return result;
837 // Create the sensitive seed value
838     // If this is a primary key in the endorsement hierarchy, stir the DRBG state
839     // This implementation uses both shProof and ehProof to make sure that there
840     // is no leakage of either.
841     if(object->attributes.primary && object->attributes.epsHierarchy)
842     {
843         DRBG_AdditionalData((DRBG_STATE *)rand, &gp.shProof.b);
844         DRBG_AdditionalData((DRBG_STATE *)rand, &gp.ehProof.b);
845     }
846     // Generate a seedValue that is the size of the digest produced by nameAlg
847     sensitive->seedValue.t.size =
848         DRBG_Generate(rand, sensitive->seedValue.t.buffer,
849                       CryptHashGetDigestSize(publicArea->nameAlg));
850     if(g_inFailureMode)
851         return TPM_RC_FAILURE;
852     else if(sensitive->seedValue.t.size == 0)
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
853             return TPM_RC_NO_RESULT;
854         // For symmetric objects, need to compute the unique value for the public area
855         if(publicArea->type == ALG_SYMCIPHER_VALUE
856            || publicArea->type == ALG_KEYEDHASH_VALUE)
857         {
858             CryptComputeSymmetricUnique(publicArea, sensitive, &publicArea->unique.sym);
859         }
860         else
861         {
862             // if this is an asymmetric key and it isn't a parent, then
863             // get rid of the seed.
864             if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign)
865                || !IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
866                 memset(&sensitive->seedValue, 0, sizeof(sensitive->seedValue));
867         }
868         // Compute the name
869         PublicMarshalAndComputeName(publicArea, &object->name);
870         return result;
871     }
```

10.2.6.6.9 CryptGetSignHashAlg()

Get the hash algorithm of signature from a TPMT_SIGNATURE structure. It assumes the signature is not NULL This is a function for easy access

```
872     TPMI_ALG_HASH
873     CryptGetSignHashAlg(
874         TPMT_SIGNATURE   *auth            // IN: signature
875         )
876     {
877         if(auth->sigAlg == ALG_NULL_VALUE)
878             FAIL(FATAL_ERROR_INTERNAL);
879
880         // Get authHash algorithm based on signing scheme
881         switch(auth->sigAlg)
882         {
883     #if ALG_RSA
884         // If RSA is supported, both RSASSA and RSAPSS are required
885     #   if !defined ALG_RSASSA_VALUE || !defined ALG_RSAPSS_VALUE
886     #       error "RSASSA and RSAPSS are required for RSA"
887     #   endif
888             case ALG_RSASSA_VALUE:
889                 return auth->signature.rsassa.hash;
890             case ALG_RSAPSS_VALUE:
891                 return auth->signature.rsapss.hash;
892     #endif // ALG_RSA
893
894     #if ALG_ECC
895         // If ECC is defined, ECDSA is mandatory
896     #   if !ALG_ECDSA
897     #       error "ECDSA is requried for ECC"
898     #   endif
899             case ALG_ECDSA_VALUE:
900         // SM2 and ECSCHNORR are optional
901
902     #   if      ALG_SM2
903             case ALG_SM2_VALUE:
904     #   endif
905     #   if   ALG_ECSCHNORR
906             case ALG_ECSCHNORR_VALUE:
907     #   endif
908             //all ECC signatures look the same
909                 return auth->signature.ecdsa.hash;
910
```

```
911  #   if  ALG_ECDAA
912          // Don't know how to verify an ECDAA signature
913          case ALG_ECDAA_VALUE:
914              break;
915  #   endif
916
917  #endif // ALG_ECC
918
919          case ALG_HMAC_VALUE:
920              return auth->signature.hmac.hashAlg;
921
922          default:
923              break;
924      }
925      return ALG_NULL_VALUE;
926  }
```

10.2.6.6.10 CryptIsSplitSign()

This function us used to determine if the signing operation is a split signing operation that required a TPM2_Commit().

```
927  BOOL
928  CryptIsSplitSign(
929      TPM_ALG_ID       scheme          // IN: the algorithm selector
930      )
931  {
932      switch(scheme)
933      {
934  #   if  ALG_ECDAA
935          case ALG_ECDAA_VALUE:
936              return TRUE;
937              break;
938  #   endif   // ALG_ECDAA
939          default:
940              return FALSE;
941              break;
942      }
943  }
```

10.2.6.6.11 CryptIsAsymSignScheme()

This function indicates if a scheme algorithm is a sign algorithm.

```
944  BOOL
945  CryptIsAsymSignScheme(
946      TPMI_ALG_PUBLIC          publicType,        // IN: Type of the object
947      TPMI_ALG_ASYM_SCHEME     scheme             // IN: the scheme
948      )
949  {
950      BOOL            isSignScheme = TRUE;
951
952      switch(publicType)
953      {
954  #if ALG_RSA
955          case ALG_RSA_VALUE:
956              switch(scheme)
957              {
958  #   if !ALG_RSASSA   || !ALG_RSAPSS
959  #       error "RSASSA and PSAPSS required if RSA used."
960  #   endif
961                  case ALG_RSASSA_VALUE:
962                  case ALG_RSAPSS_VALUE:
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
963                     break;
964                 default:
965                     isSignScheme = FALSE;
966                     break;
967             }
968             break;
969 #endif // ALG_RSA
970
971 #if ALG_ECC
972         // If ECC is implemented ECDSA is required
973         case ALG_ECC_VALUE:
974             switch(scheme)
975             {
976                 // Support for ECDSA is required for ECC
977                 case ALG_ECDSA_VALUE:
978 #if ALG_ECDAA // ECDAA is optional
979                 case ALG_ECDAA_VALUE:
980 #endif
981 #if ALG_ECSCHNORR // Schnorr is also optional
982                 case ALG_ECSCHNORR_VALUE:
983 #endif
984 #if ALG_SM2 // SM2 is optional
985                 case ALG_SM2_VALUE:
986 #endif
987                     break;
988                 default:
989                     isSignScheme = FALSE;
990                     break;
991             }
992             break;
993 #endif // ALG_ECC
994         default:
995             isSignScheme = FALSE;
996             break;
997     }
998     return isSignScheme;
999 }
```

10.2.6.6.12 CryptIsAsymDecryptScheme()

This function indicate if a scheme algorithm is a decrypt algorithm.

```
1000 BOOL
1001 CryptIsAsymDecryptScheme(
1002     TPMI_ALG_PUBLIC          publicType,        // IN: Type of the object
1003     TPMI_ALG_ASYM_SCHEME     scheme             // IN: the scheme
1004     )
1005 {
1006     BOOL        isDecryptScheme = TRUE;
1007
1008     switch(publicType)
1009     {
1010 #if ALG_RSA
1011         case ALG_RSA_VALUE:
1012             switch(scheme)
1013             {
1014                 case ALG_RSAES_VALUE:
1015                 case ALG_OAEP_VALUE:
1016                     break;
1017                 default:
1018                     isDecryptScheme = FALSE;
1019                     break;
1020             }
1021             break;
```

```
1022 #endif // ALG_RSA
1023
1024 #if ALG_ECC
1025         // If ECC is implemented ECDH is required
1026         case ALG_ECC_VALUE:
1027             switch(scheme)
1028             {
1029 #if  !ALG_ECDH
1030 #   error "ECDH is required for ECC"
1031 #endif
1032                 case ALG_ECDH_VALUE:
1033 #if ALG_SM2
1034                 case ALG_SM2_VALUE:
1035 #endif
1036 #if ALG_ECMQV
1037                 case ALG_ECMQV_VALUE:
1038 #endif
1039                     break;
1040                 default:
1041                     isDecryptScheme = FALSE;
1042                     break;
1043             }
1044             break;
1045 #endif // ALG_ECC
1046         default:
1047             isDecryptScheme = FALSE;
1048             break;
1049     }
1050     return isDecryptScheme;
1051 }
```

10.2.6.6.13 CryptSelectSignScheme()

This function is used by the attestation and signing commands. It implements the rules for selecting the signature scheme to use in signing. This function requires that the signing key either be TPM_RH_NULL or be loaded.

If a default scheme is defined in object, the default scheme should be chosen, otherwise, the input scheme should be chosen. In the case that both object and input scheme has a non-NULL scheme algorithm, if the schemes are compatible, the input scheme will be chosen.

This function should not be called if '*signObject->publicArea*.type' == ALG_SYMCIPHER.

Return Value	Meaning
TRUE(1)	scheme selected
FALSE(0)	both *scheme* and key's default scheme are empty; or *scheme* is empty while key's default scheme requires explicit input scheme (split signing); or non-empty default key scheme differs from *scheme*

```
1052 BOOL
1053 CryptSelectSignScheme(
1054     OBJECT              *signObject,    // IN: signing key
1055     TPMT_SIG_SCHEME     *scheme         // IN/OUT: signing scheme
1056     )
1057 {
1058     TPMT_SIG_SCHEME     *objectScheme;
1059     TPMT_PUBLIC         *publicArea;
1060     BOOL                 OK;
1061
1062     // If the signHandle is TPM_RH_NULL, then the NULL scheme is used, regardless
1063     // of the setting of scheme
1064     if(signObject == NULL)
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1065        {
1066            OK = TRUE;
1067            scheme->scheme = ALG_NULL_VALUE;
1068            scheme->details.any.hashAlg = ALG_NULL_VALUE;
1069        }
1070        else
1071        {
1072            // assignment to save typing.
1073            publicArea = &signObject->publicArea;
1074
1075            // A symmetric cipher can be used to encrypt and decrypt but it can't
1076            // be used for signing
1077            if(publicArea->type == ALG_SYMCIPHER_VALUE)
1078                return FALSE;
1079            // Point to the scheme object
1080            if(CryptIsAsymAlgorithm(publicArea->type))
1081                objectScheme =
1082                (TPMT_SIG_SCHEME *)&publicArea->parameters.asymDetail.scheme;
1083            else
1084                objectScheme =
1085                (TPMT_SIG_SCHEME *)&publicArea->parameters.keyedHashDetail.scheme;
1086
1087        // If the object doesn't have a default scheme, then use the
1088        // input scheme.
1089            if(objectScheme->scheme == ALG_NULL_VALUE)
1090            {
1091                // Input and default can't both be NULL
1092                OK = (scheme->scheme != ALG_NULL_VALUE);
1093                // Assume that the scheme is compatible with the key. If not,
1094                // an error will be generated in the signing operation.
1095            }
1096            else if(scheme->scheme == ALG_NULL_VALUE)
1097            {
1098                // input scheme is NULL so use default
1099
1100                // First, check to see if the default requires that the caller
1101                // provided scheme data
1102                OK = !CryptIsSplitSign(objectScheme->scheme);
1103                if(OK)
1104                {
1105                    // The object has a scheme and the input is TPM_ALG_NULL so copy
1106                    // the object scheme as the final scheme. It is better to use a
1107                    // structure copy than a copy of the individual fields.
1108                    *scheme = *objectScheme;
1109                }
1110            }
1111            else
1112            {
1113                // Both input and object have scheme selectors
1114                // If the scheme and the hash are not the same then...
1115                // NOTE: the reason that there is no copy here is that the input
1116                // might contain extra data for a split signing scheme and that
1117                // data is not in the object so, it has to be preserved.
1118                OK = (objectScheme->scheme == scheme->scheme)
1119                    && (objectScheme->details.any.hashAlg
1120                        == scheme->details.any.hashAlg);
1121            }
1122        }
1123        return OK;
1124    }
```

10.2.6.6.14 CryptSign()

Sign a digest with asymmetric key or HMAC. This function is called by attestation commands and the generic TPM2_Sign() command. This function checks the key scheme and digest size. It does not check if the sign operation is allowed for restricted key. It should be checked before the function is called. The function will assert if the key is not a signing key.

Error Returns	Meaning
TPM_RC_SCHEME	*signScheme* is not compatible with the signing key type
TPM_RC_VALUE	*digest* value is greater than the modulus of *signHandle* or size of *hashData* does not match hash algorithm in*signScheme* (for an RSA key); invalid commit status or failed to generate r value (for an ECC key)

```
1125   TPM_RC
1126   CryptSign(
1127       OBJECT              *signKey,       // IN: signing key
1128       TPMT_SIG_SCHEME     *signScheme,    // IN: sign scheme.
1129       TPM2B_DIGEST        *digest,        // IN: The digest being signed
1130       TPMT_SIGNATURE      *signature      // OUT: signature
1131       )
1132   {
1133       TPM_RC               result = TPM_RC_SCHEME;
1134
1135       // Initialize signature scheme
1136       signature->sigAlg = signScheme->scheme;
1137
1138       // If the signature algorithm is TPM_ALG_NULL or the signing key is NULL,
1139       // then we are done
1140       if((signature->sigAlg == ALG_NULL_VALUE) || (signKey == NULL))
1141           return TPM_RC_SUCCESS;
1142
1143       // Initialize signature hash
1144       // Note: need to do the check for TPM_ALG_NULL first because the null scheme
1145       // doesn't have a hashAlg member.
1146       signature->signature.any.hashAlg = signScheme->details.any.hashAlg;
1147
1148       // perform sign operation based on different key type
1149       switch(signKey->publicArea.type)
1150       {
1151   #if ALG_RSA
1152           case ALG_RSA_VALUE:
1153               result = CryptRsaSign(signature, signKey, digest, NULL);
1154               break;
1155   #endif // ALG_RSA
1156   #if ALG_ECC
1157           case ALG_ECC_VALUE:
1158               // The reason that signScheme is passed to CryptEccSign but not to the
1159               // other signing methods is that the signing for ECC may be split and
1160               // need the 'r' value that is in the scheme but not in the signature.
1161               result = CryptEccSign(signature, signKey, digest,
1162                                     (TPMT_ECC_SCHEME *)signScheme, NULL);
1163               break;
1164   #endif // ALG_ECC
1165           case ALG_KEYEDHASH_VALUE:
1166               result = CryptHmacSign(signature, signKey, digest);
1167               break;
1168           default:
1169               FAIL(FATAL_ERROR_INTERNAL);
1170               break;
1171       }
1172       return result;
1173   }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.2.6.6.15 CryptValidateSignature()

This function is used to verify a signature. It is called by TPM2_VerifySignature() and TPM2_PolicySigned().

Since this operation only requires use of a public key, no consistency checks are necessary for the key to signature type because a caller can load any public key that they like with any scheme that they like. This routine simply makes sure that the signature is correct, whatever the type.

Error Returns	Meaning
TPM_RC_SIGNATURE	the signature is not genuine
TPM_RC_SCHEME	the scheme is not supported
TPM_RC_HANDLE	an HMAC key was selected but the private part of the key is not loaded

```
1174 TPM_RC
1175 CryptValidateSignature(
1176     TPMI_DH_OBJECT   keyHandle,     // IN: The handle of sign key
1177     TPM2B_DIGEST    *digest,        // IN: The digest being validated
1178     TPMT_SIGNATURE  *signature      // IN: signature
1179     )
1180 {
1181     // NOTE: HandleToObject will either return a pointer to a loaded object or
1182     // will assert. It will never return a non-valid value. This makes it save
1183     // to initialize 'publicArea' with the return value from HandleToObject()
1184     // without checking it first.
1185     OBJECT              *signObject = HandleToObject(keyHandle);
1186     TPMT_PUBLIC         *publicArea = &signObject->publicArea;
1187     TPM_RC               result = TPM_RC_SCHEME;
1188
1189     // The input unmarshaling should prevent any input signature from being
1190     // a NULL signature, but just in case
1191     if(signature->sigAlg == ALG_NULL_VALUE)
1192         return TPM_RC_SIGNATURE;
1193
1194     switch(publicArea->type)
1195     {
1196 #if ALG_RSA
1197         case ALG_RSA_VALUE:
1198         {
1199     //
1200             // Call RSA code to verify signature
1201             result = CryptRsaValidateSignature(signature, signObject, digest);
1202             break;
1203         }
1204 #endif // ALG_RSA
1205
1206 #if ALG_ECC
1207         case ALG_ECC_VALUE:
1208             result = CryptEccValidateSignature(signature, signObject, digest);
1209             break;
1210 #endif // ALG_ECC
1211
1212         case ALG_KEYEDHASH_VALUE:
1213             if(signObject->attributes.publicOnly)
1214                 result = TPM_RCS_HANDLE;
1215             else
1216                 result = CryptHMACVerifySignature(signObject, digest, signature);
1217             break;
1218         default:
1219             break;
1220     }
```

 Copyright © TCG 2006-2020

```
1221        return result;
1222    }
```

10.2.6.6.16 CryptGetTestResult

This function returns the results of a self-test function.

NOTE: the behavior in this function is NOT the correct behavior for a real TPM implementation. An artificial behavior is placed here due to the limitation of a software simulation environment. For the correct behavior, consult the part 3 specification for TPM2_GetTestResult().

```
1223    TPM_RC
1224    CryptGetTestResult(
1225        TPM2B_MAX_BUFFER    *outData        // OUT: test result data
1226        )
1227    {
1228        outData->t.size = 0;
1229        return TPM_RC_SUCCESS;
1230    }
```

10.2.6.6.17 CryptValidateKeys()

This function is used to verify that the key material of and object is valid. For a *publicOnly* object, the key is verified for size and, if it is an ECC key, it is verified to be on the specified curve. For a key with a sensitive area, the binding between the public and private parts of the key are verified. If the *nameAlg* of the key is TPM_ALG_NULL, then the size of the sensitive area is verified but the public portion is not verified, unless the key is an RSA key. For an RSA key, the reason for loading the sensitive area is to use it. The only way to use a private RSA key is to compute the private exponent. To compute the private exponent, the public modulus is used.

Error Returns	Meaning
TPM_RC_BINDING	the public and private parts are not cryptographically bound
TPM_RC_HASH	cannot have a *publicOnly* key with *nameAlg* of TPM_ALG_NULL
TPM_RC_KEY	the public unique is not valid
TPM_RC_KEY_SIZE	the private area key is not valid
TPM_RC_TYPE	the types of the sensitive and private parts do not match

```
1231    TPM_RC
1232    CryptValidateKeys(
1233        TPMT_PUBLIC      *publicArea,
1234        TPMT_SENSITIVE   *sensitive,
1235        TPM_RC            blamePublic,
1236        TPM_RC            blameSensitive
1237        )
1238    {
1239        TPM_RC                result;
1240        UINT16                keySizeInBytes;
1241        UINT16                digestSize = CryptHashGetDigestSize(publicArea->nameAlg);
1242        TPMU_PUBLIC_PARMS    *params = &publicArea->parameters;
1243        TPMU_PUBLIC_ID       *unique = &publicArea->unique;
1244
1245        if(sensitive != NULL)
1246        {
1247            // Make sure that the types of the public and sensitive are compatible
1248            if(publicArea->type != sensitive->sensitiveType)
1249                return TPM_RCS_TYPE + blameSensitive;
1250            // Make sure that the authValue is not bigger than allowed
1251            // If there is no name algorithm, then the size just needs to be less than
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
1252            // the maximum size of the buffer used for authorization. That size check
1253            // was made during unmarshaling of the sensitive area
1254            if((sensitive->authValue.t.size) > digestSize && (digestSize > 0))
1255                return TPM_RCS_SIZE + blameSensitive;
1256        }
1257        switch(publicArea->type)
1258        {
1259    #if ALG_RSA
1260            case ALG_RSA_VALUE:
1261                keySizeInBytes = BITS_TO_BYTES(params->rsaDetail.keyBits);
1262
1263                // Regardless of whether there is a sensitive area, the public modulus
1264                // needs to have the correct size. Otherwise, it can't be used for
1265                // any public key operation nor can it be used to compute the private
1266                // exponent.
1267                // NOTE: This implementation only supports key sizes that are multiples
1268                // of 1024 bits which means that the MSb of the 0th byte will always be
1269                // SET in any prime and in the public modulus.
1270                if((unique->rsa.t.size != keySizeInBytes)
1271                    || (unique->rsa.t.buffer[0] < 0x80))
1272                    return TPM_RCS_KEY + blamePublic;
1273                if(params->rsaDetail.exponent != 0
1274                   && params->rsaDetail.exponent < 7)
1275                    return TPM_RCS_VALUE + blamePublic;
1276                if(sensitive != NULL)
1277                {
1278                    // If there is a sensitive area, it has to be the correct size
1279                    // including having the correct high order bit SET.
1280                    if(((sensitive->sensitive.rsa.t.size * 2) != keySizeInBytes)
1281                       || (sensitive->sensitive.rsa.t.buffer[0] < 0x80))
1282                        return TPM_RCS_KEY_SIZE + blameSensitive;
1283                }
1284                break;
1285    #endif
1286    #if ALG_ECC
1287            case ALG_ECC_VALUE:
1288            {
1289                TPMI_ECC_CURVE      curveId;
1290                curveId = params->eccDetail.curveID;
1291                keySizeInBytes = BITS_TO_BYTES(CryptEccGetKeySizeForCurve(curveId));
1292                if(sensitive == NULL)
1293                {
1294                    // Validate the public key size
1295                    if(unique->ecc.x.t.size != keySizeInBytes
1296                       || unique->ecc.y.t.size != keySizeInBytes)
1297                        return TPM_RCS_KEY + blamePublic;
1298                    if(publicArea->nameAlg != ALG_NULL_VALUE)
1299                    {
1300                        if(!CryptEccIsPointOnCurve(curveId, &unique->ecc))
1301                            return TPM_RCS_ECC_POINT + blamePublic;
1302                    }
1303                }
1304                else
1305                {
1306                    // If the nameAlg is TPM_ALG_NULL, then only verify that the
1307                    // private part of the key is OK.
1308                    if(!CryptEccIsValidPrivateKey(&sensitive->sensitive.ecc,
1309                                                  curveId))
1310                            return TPM_RCS_KEY_SIZE;
1311                    if(publicArea->nameAlg != ALG_NULL_VALUE)
1312                    {
1313                    // Full key load, verify that the public point belongs to the
1314                    // private key.
1315                        TPMS_ECC_POINT          toCompare;
1316                        result = CryptEccPointMultiply(&toCompare, curveId, NULL,
1317                                                       &sensitive->sensitive.ecc,
```

```
1318                                                     NULL, NULL);
1319                        if(result != TPM_RC_SUCCESS)
1320                            return TPM_RCS_BINDING;
1321                        else
1322                        {
1323                        // Make sure that the private key generated the public key.
1324                        // The input values and the values produced by the point
1325                        // multiply may not be the same size so adjust the computed
1326                        // value to match the size of the input value by adding or
1327                        // removing zeros.
1328                            AdjustNumberB(&toCompare.x.b, unique->ecc.x.t.size);
1329                            AdjustNumberB(&toCompare.y.b, unique->ecc.y.t.size);
1330                            if(!MemoryEqual2B(&unique->ecc.x.b, &toCompare.x.b)
1331                               || !MemoryEqual2B(&unique->ecc.y.b, &toCompare.y.b))
1332                                return TPM_RCS_BINDING;
1333                        }
1334                    }
1335                }
1336                break;
1337            }
1338 #endif
1339        default:
1340            // Checks for SYMCIPHER and KEYEDHASH are largely the same
1341            // If public area has a nameAlg, then validate the public area size
1342            // and if there is also a sensitive area, validate the binding
1343
1344            // For consistency, if the object is public-only just make sure that
1345            // the unique field is consistent with the name algorithm
1346            if(sensitive == NULL)
1347            {
1348                if(unique->sym.t.size != digestSize)
1349                    return TPM_RCS_KEY + blamePublic;
1350            }
1351            else
1352            {
1353                // Make sure that the key size in the sensitive area is consistent.
1354                if(publicArea->type == ALG_SYMCIPHER_VALUE)
1355                {
1356                    result = CryptSymKeyValidate(&params->symDetail.sym,
1357                                                 &sensitive->sensitive.sym);
1358                    if(result != TPM_RC_SUCCESS)
1359                        return result + blameSensitive;
1360                }
1361                else
1362                {
1363                    // For a keyed hash object, the key has to be less than the
1364                    // smaller of the block size of the hash used in the scheme or
1365                    // 128 bytes. The worst case value is limited by the
1366                    // unmarshaling code so the only thing left to be checked is
1367                    // that it does not exceed the block size of the hash.
1368                    // by the hash algorithm of the scheme.
1369                    TPMT_KEYEDHASH_SCHEME       *scheme;
1370                    UINT16                       maxSize;
1371                    scheme = &params->keyedHashDetail.scheme;
1372                    if(scheme->scheme == ALG_XOR_VALUE)
1373                    {
1374                        maxSize = CryptHashGetBlockSize(scheme->details.xor.hashAlg);
1375                    }
1376                    else if(scheme->scheme == ALG_HMAC_VALUE)
1377                    {
1378                        maxSize = CryptHashGetBlockSize(scheme->details.hmac.hashAlg);
1379                    }
1380                    else if(scheme->scheme == ALG_NULL_VALUE)
1381                    {
1382                        // Not signing or xor so must be a data block
1383                        maxSize = 128;
```

```
1384                    }
1385                    else
1386                        return TPM_RCS_SCHEME + blamePublic;
1387                    if(sensitive->sensitive.bits.t.size > maxSize)
1388                        return TPM_RCS_KEY_SIZE + blameSensitive;
1389                }
1390                // If there is a nameAlg, check the binding
1391                if(publicArea->nameAlg != ALG_NULL_VALUE)
1392                {
1393                    TPM2B_DIGEST            compare;
1394                    if(sensitive->seedValue.t.size != digestSize)
1395                        return TPM_RCS_KEY_SIZE + blameSensitive;
1396
1397                    CryptComputeSymmetricUnique(publicArea, sensitive, &compare);
1398                    if(!MemoryEqual2B(&unique->sym.b, &compare.b))
1399                        return TPM_RC_BINDING;
1400                }
1401            }
1402            break;
1403    }
1404    // For a parent, need to check that the seedValue is the correct size for
1405    // protections. It should be at least half the size of the nameAlg
1406    if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
1407       && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
1408       && sensitive != NULL
1409       && publicArea->nameAlg != ALG_NULL_VALUE)
1410    {
1411        if((sensitive->seedValue.t.size < (digestSize / 2))
1412           || (sensitive->seedValue.t.size > digestSize))
1413            return TPM_RCS_SIZE + blameSensitive;
1414    }
1415    return TPM_RC_SUCCESS;
1416 }
```

10.2.6.6.18 CryptSelectMac()

This function is used to set the MAC scheme based on the key parameters and the input scheme.

Error Returns	Meaning
TPM_RC_SCHEME	the scheme is not a valid mac scheme
TPM_RC_TYPE	the input key is not a type that supports a mac
TPM_RC_VALUE	the input scheme and the key scheme are not compatible

```
1417 TPM_RC
1418 CryptSelectMac(
1419     TPMT_PUBLIC             *publicArea,
1420     TPMI_ALG_MAC_SCHEME     *inMac
1421 )
1422 {
1423     TPM_ALG_ID              macAlg = ALG_NULL_VALUE;
1424     switch(publicArea->type)
1425     {
1426         case ALG_KEYEDHASH_VALUE:
1427         {
1428             // Local value to keep lines from getting too long
1429             TPMT_KEYEDHASH_SCHEME   *scheme;
1430             scheme = &publicArea->parameters.keyedHashDetail.scheme;
1431             // Expect that the scheme is either HMAC or NULL
1432             if(scheme->scheme != ALG_NULL_VALUE)
1433                 macAlg = scheme->details.hmac.hashAlg;
1434             break;
1435         }
```

```
1436            case ALG_SYMCIPHER_VALUE:
1437            {
1438                TPMT_SYM_DEF_OBJECT      *scheme;
1439                scheme = &publicArea->parameters.symDetail.sym;
1440                // Expect that the scheme is either valid symmetric cipher or NULL
1441                if(scheme->algorithm != ALG_NULL_VALUE)
1442                    macAlg = scheme->mode.sym;
1443                break;
1444            }
1445            default:
1446                return TPM_RCS_TYPE;
1447        }
1448        // If the input value is not TPM_ALG_NULL ...
1449        if(*inMac != ALG_NULL_VALUE)
1450        {
1451            // ... then either the scheme in the key must be TPM_ALG_NULL or the input
1452            // value must match
1453            if((macAlg != ALG_NULL_VALUE) && (*inMac != macAlg))
1454                return TPM_RCS_VALUE;
1455        }
1456        else
1457        {
1458            // Since the input value is TPM_ALG_NULL, then the key value can't be
1459            // TPM_ALG_NULL
1460            if(macAlg == ALG_NULL_VALUE)
1461                return TPM_RCS_VALUE;
1462            *inMac = macAlg;
1463        }
1464        if(!CryptMacIsValidForKey(publicArea->type, *inMac, FALSE))
1465            return TPM_RCS_SCHEME;
1466        return TPM_RC_SUCCESS;
1467    }
```

10.2.6.6.19 CryptMacIsValidForKey()

Check to see if the key type is compatible with the mac type

```
1468    BOOL
1469    CryptMacIsValidForKey(
1470        TPM_ALG_ID          keyType,
1471        TPM_ALG_ID          macAlg,
1472        BOOL                flag
1473    )
1474    {
1475        switch(keyType)
1476        {
1477            case ALG_KEYEDHASH_VALUE:
1478                return CryptHashIsValidAlg(macAlg, flag);
1479                break;
1480            case ALG_SYMCIPHER_VALUE:
1481                return CryptSmacIsValidAlg(macAlg, flag);
1482                break;
1483            default:
1484                break;
1485        }
1486        return FALSE;
1487    }
```

10.2.6.6.20 CryptSmacIsValidAlg()

This function is used to test if an algorithm is a supported SMAC algorithm. It needs to be updated as new algorithms are added.

```
1488 BOOL
1489 CryptSmacIsValidAlg(
1490     TPM_ALG_ID      alg,
1491     BOOL            FLAG        // IN: Indicates if TPM_ALG_NULL is valid
1492 )
1493 {
1494     switch (alg)
1495     {
1496 #if ALG_CMAC
1497         case ALG_CMAC_VALUE:
1498             return TRUE;
1499             break;
1500 #endif
1501         case ALG_NULL_VALUE:
1502             return FLAG;
1503             break;
1504         default:
1505             return FALSE;
1506     }
1507 }
```

10.2.6.6.21 CryptSymModeIsValid()

Function checks to see if an algorithm ID is a valid, symmetric block cipher mode for the TPM. If *flag* is SET, them TPM_ALG_NULL is a valid mode. not include the modes used for SMAC

```
1508 BOOL
1509 CryptSymModeIsValid(
1510     TPM_ALG_ID          mode,
1511     BOOL                flag
1512 )
1513 {
1514     switch(mode)
1515     {
1516 #if ALG_CTR
1517         case ALG_CTR_VALUE:
1518 #endif // ALG_CTR
1519 #if ALG_OFB
1520         case ALG_OFB_VALUE:
1521 #endif // ALG_OFB
1522 #if ALG_CBC
1523         case ALG_CBC_VALUE:
1524 #endif // ALG_CBC
1525 #if ALG_CFB
1526         case ALG_CFB_VALUE:
1527 #endif // ALG_CFB
1528 #if ALG_ECB
1529         case ALG_ECB_VALUE:
1530 #endif // ALG_ECB
1531             return TRUE;
1532         case ALG_NULL_VALUE:
1533             return flag;
1534             break;
1535         default:
1536             break;
1537     }
1538     return FALSE;
1539 }
```

 Copyright © TCG 2006-2020

10.2.7 CryptSelfTest.c

10.2.7.1 Introduction

The functions in this file are designed to support self-test of cryptographic functions in the TPM. The TPM allows the user to decide whether to run self-test on a demand basis or to run all the self-tests before proceeding.

The self-tests are controlled by a set of bit vectors. The *g_untestedDecryptionAlgorithms* vector has a bit for each decryption algorithm that needs to be tested and *g_untestedEncryptionAlgorithms* has a bit for each encryption algorithm that needs to be tested. Before an algorithm is used, the appropriate vector is checked (indexed using the algorithm ID). If the bit is 1, then the test function should be called.

For more information, see TpmSelfTests().txt

```
1   #include "Tpm.h"
```

10.2.7.2 Functions

10.2.7.2.1 RunSelfTest()

Local function to run self-test

```
2   static TPM_RC
3   CryptRunSelfTests(
4       ALGORITHM_VECTOR    *toTest         // IN: the vector of the algorithms to test
5       )
6   {
7       TPM_ALG_ID           alg;
8
9       // For each of the algorithms that are in the toTestVecor, need to run a
10      // test
11      for(alg = TPM_ALG_FIRST; alg <= TPM_ALG_LAST; alg++)
12      {
13          if(TEST_BIT(alg, *toTest))
14          {
15              TPM_RC          result = CryptTestAlgorithm(alg, toTest);
16              if(result != TPM_RC_SUCCESS)
17                  return result;
18          }
19      }
20      return TPM_RC_SUCCESS;
21  }
```

10.2.7.2.2 CryptSelfTest()

This function is called to start/complete a full self-test. If *fullTest* is NO, then only the untested algorithms will be run. If *fullTest* is YES, then *g_untestedDecryptionAlgorithms* is reinitialized and then all tests are run. This implementation of the reference design does not support processing outside the framework of a TPM command. As a consequence, this command does not complete until all tests are done. Since this can take a long time, the TPM will check after each test to see if the command is canceled. If so, then the TPM will returned TPM_RC_CANCELLED. To continue with the self-tests, call TPM2_SelfTest(*fullTest* == No) and the TPM will complete the testing.

 Copyright © TCG 2006-2020

Error Returns	Meaning
TPM_RC_CANCELED	if the command is canceled

```
22  LIB_EXPORT
23  TPM_RC
24  CryptSelfTest(
25      TPMI_YES_NO      fullTest       // IN: if full test is required
26      )
27  {
28  #if SIMULATION
29      if(g_forceFailureMode)
30          FAIL(FATAL_ERROR_FORCED);
31  #endif
32
33      // If the caller requested a full test, then reset the to test vector so that
34      // all the tests will be run
35      if(fullTest == YES)
36      {
37          MemoryCopy(g_toTest,
38                     g_implementedAlgorithms,
39                     sizeof(g_toTest));
40      }
41      return CryptRunSelfTests(&g_toTest);
42  }
```

10.2.7.2.3 CryptIncrementalSelfTest()

This function is used to perform an incremental self-test. This implementation will perform the *toTest* values before returning. That is, it assumes that the TPM cannot perform background tasks between commands.

This command may be canceled. If it is, then there is no return result. However, this command can be run again and the incremental progress will not be lost.

Error Returns	Meaning
TPM_RC_CANCELED	processing of this command was canceled
TPM_RC_TESTING	if *toTest* list is not empty
TPM_RC_VALUE	an algorithm in the *toTest* list is not implemented

```
43  TPM_RC
44  CryptIncrementalSelfTest(
45      TPML_ALG            *toTest,        // IN: list of algorithms to be tested
46      TPML_ALG            *toDoList       // OUT: list of algorithms needing test
47      )
48  {
49      ALGORITHM_VECTOR     toTestVector = {0};
50      TPM_ALG_ID           alg;
51      UINT32               i;
52
53      pAssert(toTest != NULL && toDoList != NULL);
54      if(toTest->count > 0)
55      {
56          // Transcribe the toTest list into the toTestVector
57          for(i = 0; i < toTest->count; i++)
58          {
59              alg = toTest->algorithms[i];
60
61              // make sure that the algorithm value is not out of range
62              if((alg > TPM_ALG_LAST) || !TEST_BIT(alg, g_implementedAlgorithms))
```

```
63                  return TPM_RC_VALUE;
64              SET_BIT(alg, toTestVector);
65          }
66          // Run the test
67          if(CryptRunSelfTests(&toTestVector) == TPM_RC_CANCELED)
68              return TPM_RC_CANCELED;
69      }
70      // Fill in the toDoList with the algorithms that are still untested
71      toDoList->count = 0;
72
73      for(alg = TPM_ALG_FIRST;
74      toDoList->count < MAX_ALG_LIST_SIZE && alg <= TPM_ALG_LAST;
75          alg++)
76      {
77          if(TEST_BIT(alg, g_toTest))
78              toDoList->algorithms[toDoList->count++] = alg;
79      }
80      return TPM_RC_SUCCESS;
81  }
```

10.2.7.2.4 CryptInitializeToTest()

This function will initialize the data structures for testing all the algorithms.

```
82  void
83  CryptInitializeToTest(
84      void
85      )
86  {
87      // Indicate that nothing has been tested
88      memset(&g_cryptoSelfTestState, 0, sizeof(g_cryptoSelfTestState));
89
90      // Copy the implemented algorithm vector
91      MemoryCopy(g_toTest, g_implementedAlgorithms, sizeof(g_toTest));
92
93      // Setting the algorithm to null causes the test function to just clear
94      // out any algorithms for which there is no test.
95      CryptTestAlgorithm(TPM_ALG_ERROR, &g_toTest);
96
97      return;
98  }
```

10.2.7.2.5 CryptTestAlgorithm()

Only point of contact with the actual self tests. If a self-test fails, there is no return and the TPM goes into failure mode. The call to TestAlgorithm() uses an algorithm selector and a bit vector. When the test is run, the corresponding bit in *toTest* and in *g_toTest* is CLEAR. If *toTest* is NULL, then only the bit in *g_toTest* is CLEAR. There is a special case for the call to TestAlgorithm(). When *alg* is ALG_ERROR, TestAlgorithm() will CLEAR any bit in *toTest* for which it has no test. This allows the knowledge about which algorithms have test to be accessed through the interface that provides the test.

Error Returns	Meaning
TPM_RC_CANCELED	test was canceled

```
99  LIB_EXPORT
100 TPM_RC
101 CryptTestAlgorithm(
102     TPM_ALG_ID          alg,
103     ALGORITHM_VECTOR   *toTest
104     )
105 {
```


 Copyright © TCG 2006-2020

```
106         TPM_RC                   result;
107     #if SELF_TEST
108         result = TestAlgorithm(alg, toTest);
109     #else
110         // If this is an attempt to determine the algorithms for which there is a
111         // self test, pretend that all of them do. We do that by not clearing any
112         // of the algorithm bits. When/if this function is called to run tests, it
113         // will over report. This can be changed so that any call to check on which
114         // algorithms have tests, 'toTest' can be cleared.
115         if(alg != TPM_ALG_ERROR)
116         {
117             CLEAR_BIT(alg, g_toTest);
118             if(toTest != NULL)
119                 CLEAR_BIT(alg, *toTest);
120         }
121         result = TPM_RC_SUCCESS;
122     #endif
123         return result;
124     }
```

10.2.8 CryptEccData.c

```
1    #include "Tpm.h"
2    #include "OIDs.h"
```

This file contains the ECC curve data. The format of the data depends on the setting of USE_BN_ECC_DATA. If it is defined, then the TPM's BigNum() format is used. Otherwise, it is kept in TPM2B format. The purpose of having the data in BigNum() format is so that it does not have to be reformatted before being used by the crypto library.

```
3    #if ALG_ECC
4    #if USE_BN_ECC_DATA
5    #       define TO_ECC_64                        TO_CRYPT_WORD_64
6    #       define TO_ECC_56(a, b, c, d, e, f, g)   TO_ECC_64(0, a, b, c, d, e, f, g)
7    #       define TO_ECC_48(a, b, c, d, e, f)      TO_ECC_64(0, 0, a, b, c, d, e, f)
8    #       define TO_ECC_40(a, b, c, d, e)         TO_ECC_64(0, 0, 0, a, b, c, d, e)
9    #   if RADIX_BITS > 32
10   #       define TO_ECC_32(a, b, c, d)            TO_ECC_64(0, 0, 0, 0, a, b, c, d)
11   #       define TO_ECC_24(a, b, c)               TO_ECC_64(0, 0, 0, 0, 0, a, b, c)
12   #       define TO_ECC_16(a, b)                  TO_ECC_64(0, 0, 0, 0, 0, 0, a, b)
13   #       define TO_ECC_8(a)                      TO_ECC_64(0, 0, 0, 0, 0, 0, 0, a)
14   #   else // RADIX_BITS == 32
15   #       define TO_ECC_32                        BIG_ENDIAN_BYTES_TO_UINT32
16   #       define TO_ECC_24(a, b, c)               TO_ECC_32(0, a, b, c)
17   #       define TO_ECC_16(a, b)                  TO_ECC_32(0, 0, a, b)
18   #       define TO_ECC_8(a)                      TO_ECC_32(0, 0, 0, a)
19   #   endif
20   #else // TPM2B_
21   #   define TO_ECC_64(a, b, c, d, e, f, g, h) a, b, c, d, e, f, g, h
22   #   define TO_ECC_56(a, b, c, d, e, f, g)    a, b, c, d, e, f, g
23   #   define TO_ECC_48(a, b, c, d, e, f)       a, b, c, d, e, f
24   #   define TO_ECC_40(a, b, c, d, e)          a, b, c, d, e
25   #   define TO_ECC_32(a, b, c, d)             a, b, c, d
26   #   define TO_ECC_24(a, b, c)                a, b, c
27   #   define TO_ECC_16(a, b)                   a, b
28   #   define TO_ECC_8(a)                       a
29   #endif
30   #if USE_BN_ECC_DATA
31   #define BN_MIN_ALLOC(bytes)                                                        \
32               (BYTES_TO_CRYPT_WORDS(bytes) == 0) ? 1 : BYTES_TO_CRYPT_WORDS(bytes)
33   # define ECC_CONST(NAME, bytes, initializer)                                       \
34       const struct {                                                                 \
35           crypt_uword_t   allocate, size, d[BN_MIN_ALLOC(bytes)];                    \
36           } NAME = {BN_MIN_ALLOC(bytes), BYTES_TO_CRYPT_WORDS(bytes),{initializer}}
37   ECC_CONST(ECC_ZERO, 0, 0);
38   #else
39   # define ECC_CONST(NAME, bytes, initializer)                                       \
40          const TPM2B_##bytes##_BYTE_VALUE NAME = {bytes, {initializer}}
```

Have to special case ECC_ZERO

```
41   TPM2B_BYTE_VALUE(1);
42   TPM2B_1_BYTE_VALUE ECC_ZERO = {1, {0}};
43
44   #endif
45
46   ECC_CONST(ECC_ONE, 1, 1);
47   #if !USE_BN_ECC_DATA
48   TPM2B_BYTE_VALUE(24);
49   #define TO_ECC_192(a, b, c)  a, b, c
50   TPM2B_BYTE_VALUE(28);
51   #define TO_ECC_224(a, b, c, d)   a, b, c, d
52   TPM2B_BYTE_VALUE(32);
```

```
 53  #define TO_ECC_256(a, b, c, d)   a, b, c, d
 54  TPM2B_BYTE_VALUE(48);
 55  #define TO_ECC_384(a, b, c, d, e, f)     a, b, c, d, e, f
 56  TPM2B_BYTE_VALUE(66);
 57  #define TO_ECC_528(a, b, c, d, e, f, g, h, i)    a, b, c, d, e, f, g, h, i
 58  TPM2B_BYTE_VALUE(80);
 59  #define TO_ECC_640(a, b, c, d, e, f, g, h, i, j)     a, b, c, d, e, f, g, h, i, j
 60  #else
 61  #define TO_ECC_192(a, b, c)  c, b, a
 62  #define TO_ECC_224(a, b, c, d)   d, c, b, a
 63  #define TO_ECC_256(a, b, c, d)   d, c, b, a
 64  #define TO_ECC_384(a, b, c, d, e, f)     f, e, d, c, b, a
 65  #define TO_ECC_528(a, b, c, d, e, f, g, h, i)    i, h, g, f, e, d, c, b, a
 66  #define TO_ECC_640(a, b, c, d, e, f, g, h, i, j)     j, i, h, g, f, e, d, c, b, a
 67  #endif // !USE_BN_ECC_DATA
 68  #if ECC_NIST_P192
 69  ECC_CONST(NIST_P192_p, 24, TO_ECC_192(
 70      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
 71      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
 72      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
 73  ECC_CONST(NIST_P192_a, 24, TO_ECC_192(
 74      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
 75      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
 76      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC)));
 77  ECC_CONST(NIST_P192_b, 24, TO_ECC_192(
 78      TO_ECC_64(0x64, 0x21, 0x05, 0x19, 0xE5, 0x9C, 0x80, 0xE7),
 79      TO_ECC_64(0x0F, 0xA7, 0xE9, 0xAB, 0x72, 0x24, 0x30, 0x49),
 80      TO_ECC_64(0xFE, 0xB8, 0xDE, 0xEC, 0xC1, 0x46, 0xB9, 0xB1)));
 81  ECC_CONST(NIST_P192_gX, 24, TO_ECC_192(
 82      TO_ECC_64(0x18, 0x8D, 0xA8, 0x0E, 0xB0, 0x30, 0x90, 0xF6),
 83      TO_ECC_64(0x7C, 0xBF, 0x20, 0xEB, 0x43, 0xA1, 0x88, 0x00),
 84      TO_ECC_64(0xF4, 0xFF, 0x0A, 0xFD, 0x82, 0xFF, 0x10, 0x12)));
 85  ECC_CONST(NIST_P192_gY, 24, TO_ECC_192(
 86      TO_ECC_64(0x07, 0x19, 0x2B, 0x95, 0xFF, 0xC8, 0xDA, 0x78),
 87      TO_ECC_64(0x63, 0x10, 0x11, 0xED, 0x6B, 0x24, 0xCD, 0xD5),
 88      TO_ECC_64(0x73, 0xF9, 0x77, 0xA1, 0x1E, 0x79, 0x48, 0x11)));
 89  ECC_CONST(NIST_P192_n, 24, TO_ECC_192(
 90      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
 91      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x99, 0xDE, 0xF8, 0x36),
 92      TO_ECC_64(0x14, 0x6B, 0xC9, 0xB1, 0xB4, 0xD2, 0x28, 0x31)));
 93  #define NIST_P192_h         ECC_ONE
 94  #define NIST_P192_gZ        ECC_ONE
 95  #if USE_BN_ECC_DATA
 96      const ECC_CURVE_DATA NIST_P192 = {
 97          (bigNum)&NIST_P192_p, (bigNum)&NIST_P192_n, (bigNum)&NIST_P192_h,
 98          (bigNum)&NIST_P192_a, (bigNum)&NIST_P192_b,
 99          {(bigNum)&NIST_P192_gX, (bigNum)&NIST_P192_gY, (bigNum)&NIST_P192_gZ}};
100  #else
101      const ECC_CURVE_DATA NIST_P192 = {
102          &NIST_P192_p.b, &NIST_P192_n.b, &NIST_P192_h.b,
103          &NIST_P192_a.b, &NIST_P192_b.b,
104          {&NIST_P192_gX.b, &NIST_P192_gY.b, &NIST_P192_gZ.b}};
105
106  #endif // USE_BN_ECC_DATA
107
108  #endif // ECC_NIST_P192
109
110
111  #if ECC_NIST_P224
112  ECC_CONST(NIST_P224_p, 28, TO_ECC_224(
113      TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
114      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
115      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
116      TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01)));
117  ECC_CONST(NIST_P224_a, 28, TO_ECC_224(
```

 Copyright © TCG 2006-2020

```
118         TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
119         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
120         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
121         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE)));
122     ECC_CONST(NIST_P224_b, 28, TO_ECC_224(
123         TO_ECC_32(0xB4, 0x05, 0x0A, 0x85),
124         TO_ECC_64(0x0C, 0x04, 0xB3, 0xAB, 0xF5, 0x41, 0x32, 0x56),
125         TO_ECC_64(0x50, 0x44, 0xB0, 0xB7, 0xD7, 0xBF, 0xD8, 0xBA),
126         TO_ECC_64(0x27, 0x0B, 0x39, 0x43, 0x23, 0x55, 0xFF, 0xB4)));
127     ECC_CONST(NIST_P224_gX, 28, TO_ECC_224(
128         TO_ECC_32(0xB7, 0x0E, 0x0C, 0xBD),
129         TO_ECC_64(0x6B, 0xB4, 0xBF, 0x7F, 0x32, 0x13, 0x90, 0xB9),
130         TO_ECC_64(0x4A, 0x03, 0xC1, 0xD3, 0x56, 0xC2, 0x11, 0x22),
131         TO_ECC_64(0x34, 0x32, 0x80, 0xD6, 0x11, 0x5C, 0x1D, 0x21)));
132     ECC_CONST(NIST_P224_gY, 28, TO_ECC_224(
133         TO_ECC_32(0xBD, 0x37, 0x63, 0x88),
134         TO_ECC_64(0xB5, 0xF7, 0x23, 0xFB, 0x4C, 0x22, 0xDF, 0xE6),
135         TO_ECC_64(0xCD, 0x43, 0x75, 0xA0, 0x5A, 0x07, 0x47, 0x64),
136         TO_ECC_64(0x44, 0xD5, 0x81, 0x99, 0x85, 0x00, 0x7E, 0x34)));
137     ECC_CONST(NIST_P224_n, 28, TO_ECC_224(
138         TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
139         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
140         TO_ECC_64(0xFF, 0xFF, 0x16, 0xA2, 0xE0, 0xB8, 0xF0, 0x3E),
141         TO_ECC_64(0x13, 0xDD, 0x29, 0x45, 0x5C, 0x5C, 0x2A, 0x3D)));
142     #define NIST_P224_h         ECC_ONE
143     #define NIST_P224_gZ        ECC_ONE
144     #if USE_BN_ECC_DATA
145         const ECC_CURVE_DATA NIST_P224 = {
146              (bigNum)&NIST_P224_p, (bigNum)&NIST_P224_n, (bigNum)&NIST_P224_h,
147             (bigNum)&NIST_P224_a, (bigNum)&NIST_P224_b,
148             {(bigNum)&NIST_P224_gX, (bigNum)&NIST_P224_gY, (bigNum)&NIST_P224_gZ}};
149     #else
150         const ECC_CURVE_DATA NIST_P224 = {
151             &NIST_P224_p.b, &NIST_P224_n.b, &NIST_P224_h.b,
152             &NIST_P224_a.b, &NIST_P224_b.b,
153             {&NIST_P224_gX.b, &NIST_P224_gY.b, &NIST_P224_gZ.b}};
154
155     #endif // USE_BN_ECC_DATA
156
157     #endif // ECC_NIST_P224
158
159
160     #if ECC_NIST_P256
161     ECC_CONST(NIST_P256_p, 32, TO_ECC_256(
162         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x01),
163         TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00),
164         TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF),
165         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
166     ECC_CONST(NIST_P256_a, 32, TO_ECC_256(
167         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x01),
168         TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00),
169         TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF),
170         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC)));
171     ECC_CONST(NIST_P256_b, 32, TO_ECC_256(
172         TO_ECC_64(0x5A, 0xC6, 0x35, 0xD8, 0xAA, 0x3A, 0x93, 0xE7),
173         TO_ECC_64(0xB3, 0xEB, 0xBD, 0x55, 0x76, 0x98, 0x86, 0xBC),
174         TO_ECC_64(0x65, 0x1D, 0x06, 0xB0, 0xCC, 0x53, 0xB0, 0xF6),
175         TO_ECC_64(0x3B, 0xCE, 0x3C, 0x3E, 0x27, 0xD2, 0x60, 0x4B)));
176     ECC_CONST(NIST_P256_gX, 32, TO_ECC_256(
177         TO_ECC_64(0x6B, 0x17, 0xD1, 0xF2, 0xE1, 0x2C, 0x42, 0x47),
178         TO_ECC_64(0xF8, 0xBC, 0xE6, 0xE5, 0x63, 0xA4, 0x40, 0xF2),
179         TO_ECC_64(0x77, 0x03, 0x7D, 0x81, 0x2D, 0xEB, 0x33, 0xA0),
180         TO_ECC_64(0xF4, 0xA1, 0x39, 0x45, 0xD8, 0x98, 0xC2, 0x96)));
181     ECC_CONST(NIST_P256_gY, 32, TO_ECC_256(
182         TO_ECC_64(0x4F, 0xE3, 0x42, 0xE2, 0xFE, 0x1A, 0x7F, 0x9B),
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
183         TO_ECC_64(0x8E, 0xE7, 0xEB, 0x4A, 0x7C, 0x0F, 0x9E, 0x16),
184         TO_ECC_64(0x2B, 0xCE, 0x33, 0x57, 0x6B, 0x31, 0x5E, 0xCE),
185         TO_ECC_64(0xCB, 0xB6, 0x40, 0x68, 0x37, 0xBF, 0x51, 0xF5)));
186     ECC_CONST(NIST_P256_n, 32, TO_ECC_256(
187         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
188         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
189         TO_ECC_64(0xBC, 0xE6, 0xFA, 0xAD, 0xA7, 0x17, 0x9E, 0x84),
190         TO_ECC_64(0xF3, 0xB9, 0xCA, 0xC2, 0xFC, 0x63, 0x25, 0x51)));
191     #define NIST_P256_h         ECC_ONE
192     #define NIST_P256_gZ        ECC_ONE
193     #if USE_BN_ECC_DATA
194         const ECC_CURVE_DATA NIST_P256 = {
195              (bigNum)&NIST_P256_p, (bigNum)&NIST_P256_n, (bigNum)&NIST_P256_h,
196             (bigNum)&NIST_P256_a, (bigNum)&NIST_P256_b,
197             {(bigNum)&NIST_P256_gX, (bigNum)&NIST_P256_gY, (bigNum)&NIST_P256_gZ}};
198     #else
199         const ECC_CURVE_DATA NIST_P256 = {
200             &NIST_P256_p.b, &NIST_P256_n.b, &NIST_P256_h.b,
201             &NIST_P256_a.b, &NIST_P256_b.b,
202             {&NIST_P256_gX.b, &NIST_P256_gY.b, &NIST_P256_gZ.b}};
203
204     #endif // USE_BN_ECC_DATA
205
206     #endif // ECC_NIST_P256
207
208
209     #if ECC_NIST_P384
210     ECC_CONST(NIST_P384_p, 48, TO_ECC_384(
211         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
212         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
213         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
214         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
215         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
216         TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF)));
217     ECC_CONST(NIST_P384_a, 48, TO_ECC_384(
218         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
219         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
220         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
221         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
222         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
223         TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFC)));
224     ECC_CONST(NIST_P384_b, 48, TO_ECC_384(
225         TO_ECC_64(0xB3, 0x31, 0x2F, 0xA7, 0xE2, 0x3E, 0xE7, 0xE4),
226         TO_ECC_64(0x98, 0x8E, 0x05, 0x6B, 0xE3, 0xF8, 0x2D, 0x19),
227         TO_ECC_64(0x18, 0x1D, 0x9C, 0x6E, 0xFE, 0x81, 0x41, 0x12),
228         TO_ECC_64(0x03, 0x14, 0x08, 0x8F, 0x50, 0x13, 0x87, 0x5A),
229         TO_ECC_64(0xC6, 0x56, 0x39, 0x8D, 0x8A, 0x2E, 0xD1, 0x9D),
230         TO_ECC_64(0x2A, 0x85, 0xC8, 0xED, 0xD3, 0xEC, 0x2A, 0xEF)));
231     ECC_CONST(NIST_P384_gX, 48, TO_ECC_384(
232         TO_ECC_64(0xAA, 0x87, 0xCA, 0x22, 0xBE, 0x8B, 0x05, 0x37),
233         TO_ECC_64(0x8E, 0xB1, 0xC7, 0x1E, 0xF3, 0x20, 0xAD, 0x74),
234         TO_ECC_64(0x6E, 0x1D, 0x3B, 0x62, 0x8B, 0xA7, 0x9B, 0x98),
235         TO_ECC_64(0x59, 0xF7, 0x41, 0xE0, 0x82, 0x54, 0x2A, 0x38),
236         TO_ECC_64(0x55, 0x02, 0xF2, 0x5D, 0xBF, 0x55, 0x29, 0x6C),
237         TO_ECC_64(0x3A, 0x54, 0x5E, 0x38, 0x72, 0x76, 0x0A, 0xB7)));
238     ECC_CONST(NIST_P384_gY, 48, TO_ECC_384(
239         TO_ECC_64(0x36, 0x17, 0xDE, 0x4A, 0x96, 0x26, 0x2C, 0x6F),
240         TO_ECC_64(0x5D, 0x9E, 0x98, 0xBF, 0x92, 0x92, 0xDC, 0x29),
241         TO_ECC_64(0xF8, 0xF4, 0x1D, 0xBD, 0x28, 0x9A, 0x14, 0x7C),
242         TO_ECC_64(0xE9, 0xDA, 0x31, 0x13, 0xB5, 0xF0, 0xB8, 0xC0),
243         TO_ECC_64(0x0A, 0x60, 0xB1, 0xCE, 0x1D, 0x7E, 0x81, 0x9D),
244         TO_ECC_64(0x7A, 0x43, 0x1D, 0x7C, 0x90, 0xEA, 0x0E, 0x5F)));
245     ECC_CONST(NIST_P384_n, 48, TO_ECC_384(
246         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
247         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
248         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
249         TO_ECC_64(0xC7, 0x63, 0x4D, 0x81, 0xF4, 0x37, 0x2D, 0xDF),
250         TO_ECC_64(0x58, 0x1A, 0x0D, 0xB2, 0x48, 0xB0, 0xA7, 0x7A),
251         TO_ECC_64(0xEC, 0xEC, 0x19, 0x6A, 0xCC, 0xC5, 0x29, 0x73)));
252     #define NIST_P384_h         ECC_ONE
253     #define NIST_P384_gZ        ECC_ONE
254     #if USE_BN_ECC_DATA
255         const ECC_CURVE_DATA NIST_P384 = {
256              (bigNum)&NIST_P384_p, (bigNum)&NIST_P384_n, (bigNum)&NIST_P384_h,
257             (bigNum)&NIST_P384_a, (bigNum)&NIST_P384_b,
258             {(bigNum)&NIST_P384_gX, (bigNum)&NIST_P384_gY, (bigNum)&NIST_P384_gZ}};
259     #else
260         const ECC_CURVE_DATA NIST_P384 = {
261             &NIST_P384_p.b, &NIST_P384_n.b, &NIST_P384_h.b,
262             &NIST_P384_a.b, &NIST_P384_b.b,
263             {&NIST_P384_gX.b, &NIST_P384_gY.b, &NIST_P384_gZ.b}};
264
265     #endif // USE_BN_ECC_DATA
266
267     #endif // ECC_NIST_P384
268
269
270     #if ECC_NIST_P521
271     ECC_CONST(NIST_P521_p, 66, TO_ECC_528(
272         TO_ECC_16(0x01, 0xFF),
273         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
274         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
275         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
276         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
277         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
278         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
279         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
280         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
281     ECC_CONST(NIST_P521_a, 66, TO_ECC_528(
282         TO_ECC_16(0x01, 0xFF),
283         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
284         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
285         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
286         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
287         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
288         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
289         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
290         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC)));
291     ECC_CONST(NIST_P521_b, 66, TO_ECC_528(
292         TO_ECC_16(0x00, 0x51),
293         TO_ECC_64(0x95, 0x3E, 0xB9, 0x61, 0x8E, 0x1C, 0x9A, 0x1F),
294         TO_ECC_64(0x92, 0x9A, 0x21, 0xA0, 0xB6, 0x85, 0x40, 0xEE),
295         TO_ECC_64(0xA2, 0xDA, 0x72, 0x5B, 0x99, 0xB3, 0x15, 0xF3),
296         TO_ECC_64(0xB8, 0xB4, 0x89, 0x91, 0x8E, 0xF1, 0x09, 0xE1),
297         TO_ECC_64(0x56, 0x19, 0x39, 0x51, 0xEC, 0x7E, 0x93, 0x7B),
298         TO_ECC_64(0x16, 0x52, 0xC0, 0xBD, 0x3B, 0xB1, 0xBF, 0x07),
299         TO_ECC_64(0x35, 0x73, 0xDF, 0x88, 0x3D, 0x2C, 0x34, 0xF1),
300         TO_ECC_64(0xEF, 0x45, 0x1F, 0xD4, 0x6B, 0x50, 0x3F, 0x00)));
301     ECC_CONST(NIST_P521_gX, 66, TO_ECC_528(
302         TO_ECC_16(0x00, 0xC6),
303         TO_ECC_64(0x85, 0x8E, 0x06, 0xB7, 0x04, 0x04, 0xE9, 0xCD),
304         TO_ECC_64(0x9E, 0x3E, 0xCB, 0x66, 0x23, 0x95, 0xB4, 0x42),
305         TO_ECC_64(0x9C, 0x64, 0x81, 0x39, 0x05, 0x3F, 0xB5, 0x21),
306         TO_ECC_64(0xF8, 0x28, 0xAF, 0x60, 0x6B, 0x4D, 0x3D, 0xBA),
307         TO_ECC_64(0xA1, 0x4B, 0x5E, 0x77, 0xEF, 0xE7, 0x59, 0x28),
308         TO_ECC_64(0xFE, 0x1D, 0xC1, 0x27, 0xA2, 0xFF, 0xA8, 0xDE),
309         TO_ECC_64(0x33, 0x48, 0xB3, 0xC1, 0x85, 0x6A, 0x42, 0x9B),
310         TO_ECC_64(0xF9, 0x7E, 0x7E, 0x31, 0xC2, 0xE5, 0xBD, 0x66)));
311     ECC_CONST(NIST_P521_gY, 66, TO_ECC_528(
312         TO_ECC_16(0x01, 0x18),
```

```
313         TO_ECC_64(0x39, 0x29, 0x6A, 0x78, 0x9A, 0x3B, 0xC0, 0x04),
314         TO_ECC_64(0x5C, 0x8A, 0x5F, 0xB4, 0x2C, 0x7D, 0x1B, 0xD9),
315         TO_ECC_64(0x98, 0xF5, 0x44, 0x49, 0x57, 0x9B, 0x44, 0x68),
316         TO_ECC_64(0x17, 0xAF, 0xBD, 0x17, 0x27, 0x3E, 0x66, 0x2C),
317         TO_ECC_64(0x97, 0xEE, 0x72, 0x99, 0x5E, 0xF4, 0x26, 0x40),
318         TO_ECC_64(0xC5, 0x50, 0xB9, 0x01, 0x3F, 0xAD, 0x07, 0x61),
319         TO_ECC_64(0x35, 0x3C, 0x70, 0x86, 0xA2, 0x72, 0xC2, 0x40),
320         TO_ECC_64(0x88, 0xBE, 0x94, 0x76, 0x9F, 0xD1, 0x66, 0x50)));
321     ECC_CONST(NIST_P521_n, 66, TO_ECC_528(
322         TO_ECC_16(0x01, 0xFF),
323         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
324         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
325         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
326         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFA),
327         TO_ECC_64(0x51, 0x86, 0x87, 0x83, 0xBF, 0x2F, 0x96, 0x6B),
328         TO_ECC_64(0x7F, 0xCC, 0x01, 0x48, 0xF7, 0x09, 0xA5, 0xD0),
329         TO_ECC_64(0x3B, 0xB5, 0xC9, 0xB8, 0x89, 0x9C, 0x47, 0xAE),
330         TO_ECC_64(0xBB, 0x6F, 0xB7, 0x1E, 0x91, 0x38, 0x64, 0x09)));
331     #define NIST_P521_h         ECC_ONE
332     #define NIST_P521_gZ        ECC_ONE
333     #if USE_BN_ECC_DATA
334         const ECC_CURVE_DATA NIST_P521 = {
335             (bigNum)&NIST_P521_p, (bigNum)&NIST_P521_n, (bigNum)&NIST_P521_h,
336             (bigNum)&NIST_P521_a, (bigNum)&NIST_P521_b,
337             {(bigNum)&NIST_P521_gX, (bigNum)&NIST_P521_gY, (bigNum)&NIST_P521_gZ}};
338     #else
339         const ECC_CURVE_DATA NIST_P521 = {
340             &NIST_P521_p.b, &NIST_P521_n.b, &NIST_P521_h.b,
341             &NIST_P521_a.b, &NIST_P521_b.b,
342             {&NIST_P521_gX.b, &NIST_P521_gY.b, &NIST_P521_gZ.b}};
343
344     #endif // USE_BN_ECC_DATA
345
346     #endif // ECC_NIST_P521
347
348
349     #if ECC_BN_P256
350     ECC_CONST(BN_P256_p, 32, TO_ECC_256(
351         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xF0, 0xCD),
352         TO_ECC_64(0x46, 0xE5, 0xF2, 0x5E, 0xEE, 0x71, 0xA4, 0x9F),
353         TO_ECC_64(0x0C, 0xDC, 0x65, 0xFB, 0x12, 0x98, 0x0A, 0x82),
354         TO_ECC_64(0xD3, 0x29, 0x2D, 0xDB, 0xAE, 0xD3, 0x30, 0x13)));
355     #define BN_P256_a           ECC_ZERO
356     ECC_CONST(BN_P256_b, 1, TO_ECC_8(3));
357     #define BN_P256_gX          ECC_ONE
358     ECC_CONST(BN_P256_gY, 1, TO_ECC_8(2));
359     ECC_CONST(BN_P256_n, 32, TO_ECC_256(
360         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xF0, 0xCD),
361         TO_ECC_64(0x46, 0xE5, 0xF2, 0x5E, 0xEE, 0x71, 0xA4, 0x9E),
362         TO_ECC_64(0x0C, 0xDC, 0x65, 0xFB, 0x12, 0x99, 0x92, 0x1A),
363         TO_ECC_64(0xF6, 0x2D, 0x53, 0x6C, 0xD1, 0x0B, 0x50, 0x0D)));
364     #define BN_P256_h           ECC_ONE
365     #define BN_P256_gZ          ECC_ONE
366     #if USE_BN_ECC_DATA
367         const ECC_CURVE_DATA BN_P256 = {
368             (bigNum)&BN_P256_p, (bigNum)&BN_P256_n, (bigNum)&BN_P256_h,
369             (bigNum)&BN_P256_a, (bigNum)&BN_P256_b,
370             {(bigNum)&BN_P256_gX, (bigNum)&BN_P256_gY, (bigNum)&BN_P256_gZ}};
371     #else
372         const ECC_CURVE_DATA BN_P256 = {
373             &BN_P256_p.b, &BN_P256_n.b, &BN_P256_h.b,
374             &BN_P256_a.b, &BN_P256_b.b,
375             {&BN_P256_gX.b, &BN_P256_gY.b, &BN_P256_gZ.b}};
376
377     #endif // USE_BN_ECC_DATA
```

```
378
379  #endif // ECC_BN_P256
380
381
382  #if ECC_BN_P638
383  ECC_CONST(BN_P638_p, 80, TO_ECC_640(
384      TO_ECC_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D),
385      TO_ECC_64(0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
386      TO_ECC_64(0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E),
387      TO_ECC_64(0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F),
388      TO_ECC_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
389      TO_ECC_64(0xC0, 0x00, 0x86, 0x52, 0x00, 0x21, 0xE5, 0x5B),
390      TO_ECC_64(0xFF, 0xFF, 0xF5, 0x1F, 0xFF, 0xF4, 0xEB, 0x80),
391      TO_ECC_64(0x00, 0x00, 0x00, 0x4C, 0x80, 0x01, 0x5A, 0xCD),
392      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xEC, 0xE0),
393      TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x67)));
394  #define BN_P638_a            ECC_ZERO
395  ECC_CONST(BN_P638_b, 2, TO_ECC_16(0x01,0x01));
396  ECC_CONST(BN_P638_gX, 80, TO_ECC_640(
397      TO_ECC_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D),
398      TO_ECC_64(0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
399      TO_ECC_64(0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E),
400      TO_ECC_64(0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F),
401      TO_ECC_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
402      TO_ECC_64(0xC0, 0x00, 0x86, 0x52, 0x00, 0x21, 0xE5, 0x5B),
403      TO_ECC_64(0xFF, 0xFF, 0xF5, 0x1F, 0xFF, 0xF4, 0xEB, 0x80),
404      TO_ECC_64(0x00, 0x00, 0x00, 0x4C, 0x80, 0x01, 0x5A, 0xCD),
405      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xEC, 0xE0),
406      TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x66)));
407  ECC_CONST(BN_P638_gY, 1, TO_ECC_8(0x10));
408  ECC_CONST(BN_P638_n, 80, TO_ECC_640(
409      TO_ECC_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D),
410      TO_ECC_64(0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
411      TO_ECC_64(0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E),
412      TO_ECC_64(0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F),
413      TO_ECC_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
414      TO_ECC_64(0x60, 0x00, 0x86, 0x55, 0x00, 0x21, 0xE5, 0x55),
415      TO_ECC_64(0xFF, 0xFF, 0xF5, 0x4F, 0xFF, 0xF4, 0xEA, 0xC0),
416      TO_ECC_64(0x00, 0x00, 0x00, 0x49, 0x80, 0x01, 0x54, 0xD9),
417      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xED, 0xA0),
418      TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x61)));
419  #define BN_P638_h            ECC_ONE
420  #define BN_P638_gZ           ECC_ONE
421  #if USE_BN_ECC_DATA
422      const ECC_CURVE_DATA BN_P638 = {
423           (bigNum)&BN_P638_p, (bigNum)&BN_P638_n, (bigNum)&BN_P638_h,
424          (bigNum)&BN_P638_a, (bigNum)&BN_P638_b,
425          {(bigNum)&BN_P638_gX, (bigNum)&BN_P638_gY, (bigNum)&BN_P638_gZ}};
426  #else
427      const ECC_CURVE_DATA BN_P638 = {
428          &BN_P638_p.b, &BN_P638_n.b, &BN_P638_h.b,
429          &BN_P638_a.b, &BN_P638_b.b,
430          {&BN_P638_gX.b, &BN_P638_gY.b, &BN_P638_gZ.b}};
431
432  #endif // USE_BN_ECC_DATA
433
434  #endif // ECC_BN_P638
435
436
437  #if ECC_SM2_P256
438  ECC_CONST(SM2_P256_p, 32, TO_ECC_256(
439      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
440      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
441      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
442      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
443  ECC_CONST(SM2_P256_a, 32, TO_ECC_256(
444      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
445      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
446      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
447      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC)));
448  ECC_CONST(SM2_P256_b, 32, TO_ECC_256(
449      TO_ECC_64(0x28, 0xE9, 0xFA, 0x9E, 0x9D, 0x9F, 0x5E, 0x34),
450      TO_ECC_64(0x4D, 0x5A, 0x9E, 0x4B, 0xCF, 0x65, 0x09, 0xA7),
451      TO_ECC_64(0xF3, 0x97, 0x89, 0xF5, 0x15, 0xAB, 0x8F, 0x92),
452      TO_ECC_64(0xDD, 0xBC, 0xBD, 0x41, 0x4D, 0x94, 0x0E, 0x93)));
453  ECC_CONST(SM2_P256_gX, 32, TO_ECC_256(
454      TO_ECC_64(0x32, 0xC4, 0xAE, 0x2C, 0x1F, 0x19, 0x81, 0x19),
455      TO_ECC_64(0x5F, 0x99, 0x04, 0x46, 0x6A, 0x39, 0xC9, 0x94),
456      TO_ECC_64(0x8F, 0xE3, 0x0B, 0xBF, 0xF2, 0x66, 0x0B, 0xE1),
457      TO_ECC_64(0x71, 0x5A, 0x45, 0x89, 0x33, 0x4C, 0x74, 0xC7)));
458  ECC_CONST(SM2_P256_gY, 32, TO_ECC_256(
459      TO_ECC_64(0xBC, 0x37, 0x36, 0xA2, 0xF4, 0xF6, 0x77, 0x9C),
460      TO_ECC_64(0x59, 0xBD, 0xCE, 0xE3, 0x6B, 0x69, 0x21, 0x53),
461      TO_ECC_64(0xD0, 0xA9, 0x87, 0x7C, 0xC6, 0x2A, 0x47, 0x40),
462      TO_ECC_64(0x02, 0xDF, 0x32, 0xE5, 0x21, 0x39, 0xF0, 0xA0)));
463  ECC_CONST(SM2_P256_n, 32, TO_ECC_256(
464      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
465      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
466      TO_ECC_64(0x72, 0x03, 0xDF, 0x6B, 0x21, 0xC6, 0x05, 0x2B),
467      TO_ECC_64(0x53, 0xBB, 0xF4, 0x09, 0x39, 0xD5, 0x41, 0x23)));
468  #define SM2_P256_h           ECC_ONE
469  #define SM2_P256_gZ          ECC_ONE
470  #if USE_BN_ECC_DATA
471      const ECC_CURVE_DATA SM2_P256 = {
472           (bigNum)&SM2_P256_p, (bigNum)&SM2_P256_n, (bigNum)&SM2_P256_h,
473          (bigNum)&SM2_P256_a, (bigNum)&SM2_P256_b,
474          {(bigNum)&SM2_P256_gX, (bigNum)&SM2_P256_gY, (bigNum)&SM2_P256_gZ}};
475  #else
476      const ECC_CURVE_DATA SM2_P256 = {
477          &SM2_P256_p.b, &SM2_P256_n.b, &SM2_P256_h.b,
478          &SM2_P256_a.b, &SM2_P256_b.b,
479          {&SM2_P256_gX.b, &SM2_P256_gY.b, &SM2_P256_gZ.b}};
480
481  #endif // USE_BN_ECC_DATA
482
483  #endif // ECC_SM2_P256
484
485
486  #define comma
487  const ECC_CURVE   eccCurves[] = {
488  #if ECC_NIST_P192
489      comma
490      {TPM_ECC_NIST_P192,
491      192,
492      {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA256_VALUE}}},
493      {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
494      &NIST_P192,
495      OID_ECC_NIST_P192
496      CURVE_NAME("NIST_P192")}
497  #   undef comma
498  #   define comma ,
499  #endif // ECC_NIST_P192
500  #if ECC_NIST_P224
501      comma
502      {TPM_ECC_NIST_P224,
503      224,
504      {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA256_VALUE}}},
505      {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
506      &NIST_P224,
507      OID_ECC_NIST_P224
```

 Copyright © TCG 2006-2020

```
508         CURVE_NAME("NIST_P224")}
509     #   undef comma
510     #   define comma ,
511     #endif // ECC_NIST_P224
512     #if ECC_NIST_P256
513         comma
514         {TPM_ECC_NIST_P256,
515         256,
516         {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA256_VALUE}}},
517         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
518         &NIST_P256,
519         OID_ECC_NIST_P256
520         CURVE_NAME("NIST_P256")}
521     #   undef comma
522     #   define comma ,
523     #endif // ECC_NIST_P256
524     #if ECC_NIST_P384
525         comma
526         {TPM_ECC_NIST_P384,
527         384,
528         {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA384_VALUE}}},
529         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
530         &NIST_P384,
531         OID_ECC_NIST_P384
532         CURVE_NAME("NIST_P384")}
533     #   undef comma
534     #   define comma ,
535     #endif // ECC_NIST_P384
536     #if ECC_NIST_P521
537         comma
538         {TPM_ECC_NIST_P521,
539         521,
540         {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA512_VALUE}}},
541         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
542         &NIST_P521,
543         OID_ECC_NIST_P521
544         CURVE_NAME("NIST_P521")}
545     #   undef comma
546     #   define comma ,
547     #endif // ECC_NIST_P521
548     #if ECC_BN_P256
549         comma
550         {TPM_ECC_BN_P256,
551         256,
552         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
553         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
554         &BN_P256,
555         OID_ECC_BN_P256
556         CURVE_NAME("BN_P256")}
557     #   undef comma
558     #   define comma ,
559     #endif // ECC_BN_P256
560     #if ECC_BN_P638
561         comma
562         {TPM_ECC_BN_P638,
563         638,
564         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
565         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
566         &BN_P638,
567         OID_ECC_BN_P638
568         CURVE_NAME("BN_P638")}
569     #   undef comma
570     #   define comma ,
571     #endif // ECC_BN_P638
572     #if ECC_SM2_P256
573         comma
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
574         {TPM_ECC_SM2_P256,
575         256,
576         {ALG_KDF1_SP800_56A_VALUE, {{ALG_SM3_256_VALUE}}},
577         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
578         &SM2_P256,
579         OID_ECC_SM2_P256
580         CURVE_NAME("SM2_P256")}
581     #   undef comma
582     #   define comma ,
583     #endif // ECC_SM2_P256
584     };
585     #endif // TPM_ALG_ECC
```

10.2.9 CryptDes.c

10.2.9.1 Introduction

This file contains the extra functions required for TDES.

10.2.9.2 Includes, Defines, and Typedefs

```
1   #include "Tpm.h"
2   #if ALG_TDES
3   #define DES_NUM_WEAK 64
4   const UINT64 DesWeakKeys[DES_NUM_WEAK] = {
5       0x0101010101010101ULL, 0xFEFEFEFEFEFEFEFEULL,
6       0xE0E0E0E0F1F1F1F1ULL, 0x1F1F1F1F0E0E0E0EULL,
7       0x011F011F010E010EULL, 0x1F011F010E010E01ULL,
8       0x01E001E001F101F1ULL, 0xE001E001F101F101ULL,
9       0x01FE01FE01FE01FEULL, 0xFE01FE01FE01FE01ULL,
10      0x1FE01FE00EF10EF1ULL, 0xE01FE01FF10EF10EULL,
11      0x1FFE1FFE0EFE0EFEULL, 0xFE1FFE1FFE0EFE0EULL,
12      0xE0FEE0FEF1FEF1FEULL, 0xFEE0FEE0FEF1FEF1ULL,
13      0x01011F1F01010E0EULL, 0x1F1F01010E0E0101ULL,
14      0xE0E01F1FF1F10E0EULL, 0x0101E0E00101F1F1ULL,
15      0x1F1FE0E00E0EF1F1ULL, 0xE0E0FEFEF1F1FEFEULL,
16      0x0101FEFE0101FEFEULL, 0x1F1FFEFE0E0EFEFEULL,
17      0xE0FE011FF1FE010EULL, 0x011F1F01010E0E01ULL,
18      0x1FE001FE0EF101FEULL, 0xE0FE1F01F1FE0E01ULL,
19      0x011FE0FE010EF1FEULL, 0x1FE0E01F0EF1F10EULL,
20      0xE0FEFEE0F1FEFEF1ULL, 0x011FFEE0010EFEF1ULL,
21      0x1FE0FE010EF1FE01ULL, 0xFE0101FEFE0101FEULL,
22      0x01E01FFE01F10EFEULL, 0x1FFE01E00EFE01F1ULL,
23      0xFE011FE0FE010EF1ULL, 0xFE01E01FFE01F10EULL,
24      0x1FFEE0010EFEF101ULL, 0xFE1F01E0FE0E01F1ULL,
25      0x01E0E00101F1F101ULL, 0x1FFEFE1F0EFEFE0EULL,
26      0xFE1FE001FE0EF101ULL, 0x01E0FE1F01F1FE0EULL,
27      0xE00101E0F10101F1ULL, 0xFE1F1FFEFE0E0EFEULL,
28      0x01FE1FE001FE0EF1ULL, 0xE0011FFEF1010EFEULL,
29      0xFEE0011FFEF1010EULL, 0x01FEE01F01FEF10EULL,
30      0xE001FE1FF101FE0EULL, 0xFEE01F01FEF10E01ULL,
31      0x01FEFE0101FEFE01ULL, 0xE01F01FEF10E01FEULL,
32      0xFEE0E0FEFEF1F1FEULL, 0x1F01011F0E01010EULL,
33      0xE01F1FE0F10E0EF1ULL, 0xFEFE0101FEFE0101ULL,
34      0x1F01E0FE0E01F1FEULL, 0xE01FFE01F10EFE01ULL,
35      0xFEFE1F1FFEFE0E0EULL, 0x1F01FEE00E01FEF1ULL,
36      0xE0E00101F1F10101ULL, 0xFEFEE0E0FEFEF1F1ULL};
37
38  //*** CryptSetOddByteParity()
39  // This function sets the per byte parity of a 64-bit value. The least-significant
40  // bit is of each byte is replaced with the odd parity of the other 7 bits in the
41  // byte. With odd parity, no byte will ever be 0x00.
42  UINT64
43  CryptSetOddByteParity(
44      UINT64          k
45      )
46  {
47  #define PMASK 0x0101010101010101ULL
48      UINT64          out;
49      k |= PMASK;     // set the parity bit
50      out = k;
51      k ^= k >> 4;
52      k ^= k >> 2;
53      k ^= k >> 1;
```

```
54      k &= PMASK;       // odd parity extracted
55      out ^= k;         // out is now even parity because parity bit was already set
56      out ^= PMASK;     // out is now even parity
57      return out;
58  }
```

10.2.9.2.1 CryptDesIsWeakKey()

Check to see if a DES key is on the list of weak, semi-weak, or possibly weak keys.

Return Value	Meaning
TRUE(1)	DES key is weak
FALSE(0)	DES key is not weak

```
59  static BOOL
60  CryptDesIsWeakKey(
61      UINT64              k
62      )
63  {
64      int                i;
65  //
66      for(i = 0; i < DES_NUM_WEAK; i++)
67      {
68          if(k == DesWeakKeys[i])
69              return TRUE;
70      }
71      return FALSE;
72  }
```

10.2.9.2.2 CryptDesValidateKey()

Function to check to see if the input key is a valid DES key where the definition of valid is that none of the elements are on the list of weak, semi-weak, or possibly weak keys; and that for two keys, K1!=K2, and for three keys that K1!=K2 and K2!=K3.

```
73  BOOL
74  CryptDesValidateKey(
75      TPM2B_SYM_KEY       *desKey     // IN: key to validate
76      )
77  {
78      UINT64               k[3];
79      int                  i;
80      int                  keys = (desKey->t.size + 7) / 8;
81      BYTE                *pk = desKey->t.buffer;
82      BOOL                 ok;
83  //
84      // Note: 'keys' is the number of keys, not the maximum index for 'k'
85      ok = ((keys == 2) || (keys == 3)) && ((desKey->t.size % 8) == 0);
86      for(i = 0; ok && i < keys; pk += 8, i++)
87      {
88          k[i] = CryptSetOddByteParity(BYTE_ARRAY_TO_UINT64(pk));
89          ok = !CryptDesIsWeakKey(k[i]);
90      }
91      ok = ok && k[0] != k[1];
92      if(keys == 3)
93          ok = ok && k[1] != k[2];
94      return ok;
95  }
```

 Copyright © TCG 2006-2020

10.2.9.2.3 CryptGenerateKeyDes()

This function is used to create a DES key of the appropriate size. The key will have odd parity in the bytes.

```
96   TPM_RC
97   CryptGenerateKeyDes(
98       TPMT_PUBLIC             *publicArea,        // IN/OUT: The public area template
99                                                   //     for the new key.
100      TPMT_SENSITIVE          *sensitive,         // OUT: sensitive area
101      RAND_STATE              *rand               // IN: the "entropy" source for
102      )
103  {
104
105      // Assume that the publicArea key size has been validated and is a supported
106      // number of bits.
107      sensitive->sensitive.sym.t.size =
108              BITS_TO_BYTES(publicArea->parameters.symDetail.sym.keyBits.sym);
109      do
110      {
111          BYTE                    *pK = sensitive->sensitive.sym.t.buffer;
112          int                      i = (sensitive->sensitive.sym.t.size + 7) / 8;
113  // Use the random number generator to generate the required number of bits
114          if(DRBG_Generate(rand, pK, sensitive->sensitive.sym.t.size) == 0)
115              return TPM_RC_NO_RESULT;
116          for(; i > 0; pK += 8, i--)
117          {
118              UINT64      k = BYTE_ARRAY_TO_UINT64(pK);
119              k = CryptSetOddByteParity(k);
120              UINT64_TO_BYTE_ARRAY(k, pK);
121          }
122      } while(!CryptDesValidateKey(&sensitive->sensitive.sym));
123      return TPM_RC_SUCCESS;
124  }
125  #endif
```


 Copyright © TCG 2006-2020

10.2.10 CryptEccKeyExchange.c

10.2.10.1 Introduction

This file contains the functions that are used for the two-phase, ECC, key-exchange protocols

```
1   #include "Tpm.h"
2   #if CC_ZGen_2Phase == YES
```

10.2.10.2 Functions

```
3   #if ALG_ECMQV
```

10.2.10.2.1 avf1()

This function does the associated value computation required by MQV key exchange. Process:

a) Convert *xQ* to an integer *xqi* using the convention specified in Appendix C.3.

b) Calculate xqm = xqi mod 2^ceil(f/2) (where f = ceil(log2(n)).

c) Calculate the associate value function avf(Q) = xqm + 2ceil(f / 2) Always returns TRUE(1).

```
4   static BOOL
5   avf1(
6       bigNum                bnX,            // IN/OUT: the reduced value
7       bigNum                bnN             // IN: the order of the curve
8       )
9   {
10  // compute f = 2^(ceil(ceil(log2(n)) / 2))
11      int                       f = (BnSizeInBits(bnN) + 1) / 2;
12  // x' = 2^f + (x mod 2^f)
13      BnMaskBits(bnX, f);    // This is mod 2*2^f but it doesn't matter because
14                              // the next operation will SET the extra bit anyway
15      BnSetBit(bnX, f);
16      return TRUE;
17  }
```

10.2.10.2.2 C_2_2_MQV()

This function performs the key exchange defined in SP800-56A 6.1.1.4 Full MQV, C(2, 2, ECC MQV).

CAUTION: Implementation of this function may require use of essential claims in patents not owned by TCG members.

Points *QsB* and *QeB* are required to be on the curve of *inQsA*. The function will fail, possibly catastrophically, if this is not the case.

Error Returns	Meaning
TPM_RC_NO_RESULT	the value for *dsA* does not give a valid point on the curve

```
18  static TPM_RC
19  C_2_2_MQV(
20      TPMS_ECC_POINT          *outZ,          // OUT: the computed point
21      TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
22      TPM2B_ECC_PARAMETER     *dsA,           // IN: static private TPM key
23      TPM2B_ECC_PARAMETER     *deA,           // IN: ephemeral private TPM key
24      TPMS_ECC_POINT          *QsB,           // IN: static public party B key
25      TPMS_ECC_POINT          *QeB            // IN: ephemeral public party B key
```


 Copyright © TCG 2006-2020

```
26          )
27      {
28          CURVE_INITIALIZED(E, curveId);
29          const ECC_CURVE_DATA     *C;
30          POINT(pQeA);
31          POINT_INITIALIZED(pQeB, QeB);
32          POINT_INITIALIZED(pQsB, QsB);
33          ECC_NUM(bnTa);
34          ECC_INITIALIZED(bnDeA, deA);
35          ECC_INITIALIZED(bnDsA, dsA);
36          ECC_NUM(bnN);
37          ECC_NUM(bnXeB);
38          TPM_RC                   retVal;
39      //
40          // Parameter checks
41          if(E == NULL)
42              ERROR_RETURN(TPM_RC_VALUE);
43          pAssert(outZ != NULL && pQeB != NULL && pQsB != NULL && deA != NULL
44                  && dsA != NULL);
45          C = AccessCurveData(E);
46      // Process:
47      //  1. implicitsigA = (de,A + avf(Qe,A)ds,A ) mod n.
48      //  2. P = h(implicitsigA)(Qe,B + avf(Qe,B)Qs,B).
49      //  3. If P = O, output an error indicator.
50      //  4. Z=xP, where xP is the x-coordinate of P.
51
52          // Compute the public ephemeral key pQeA = [de,A]G
53          if((retVal = BnPointMult(pQeA, CurveGetG(C), bnDeA, NULL, NULL, E))
54             != TPM_RC_SUCCESS)
55              goto Exit;
56
57      //  1. implicitsigA = (de,A + avf(Qe,A)ds,A ) mod n.
58      //  tA := (ds,A + de,A  avf(Xe,A)) mod n    (3)
59      //  Compute 'tA' = ('deA' +  'dsA'  avf('XeA')) mod n
60          // Ta = avf(XeA);
61          BnCopy(bnTa, pQeA->x);
62          avf1(bnTa, bnN);
63          // do Ta = ds,A * Ta mod n = dsA * avf(XeA) mod n
64          BnModMult(bnTa, bnDsA, bnTa, bnN);
65          // now Ta = deA + Ta mod n =  deA + dsA * avf(XeA) mod n
66          BnAdd(bnTa, bnTa, bnDeA);
67          BnMod(bnTa, bnN);
68
69      //  2. P = h(implicitsigA)(Qe,B + avf(Qe,B)Qs,B).
70      // Put this in because almost every case of h is == 1 so skip the call when
71          // not necessary.
72          if(!BnEqualWord(CurveGetCofactor(C), 1))
73              // Cofactor is not 1 so compute Ta := Ta * h mod n
74              BnModMult(bnTa, bnTa, CurveGetCofactor(C), CurveGetOrder(C));
75
76          // Now that 'tA' is (h * 'tA' mod n)
77          // 'outZ' = (tA)(Qe,B + avf(Qe,B)Qs,B).
78
79          // first, compute XeB = avf(XeB)
80          avf1(bnXeB, bnN);
81
82          // QsB := [XeB]QsB
83          BnPointMult(pQsB, pQsB, bnXeB, NULL, NULL, E);
84          BnEccAdd(pQeB, pQeB, pQsB, E);
85
86          // QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
87          // If the result is not the point at infinity, return QeB
88          BnPointMult(pQeB, pQeB, bnTa, NULL, NULL, E);
89          if(BnEqualZero(pQeB->z))
90              ERROR_RETURN(TPM_RC_NO_RESULT);
91          // Convert BIGNUM E to TPM2B E
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
92         BnPointTo2B(outZ, pQeB, E);
93
94     Exit:
95         CURVE_FREE(E);
96         return retVal;
97     }
98     #endif // ALG_ECMQV
```

10.2.10.2.3 C_2_2_ECDH()

This function performs the two phase key exchange defined in SP800-56A, 6.1.1.2 Full Unified Model, C(2, 2, ECC CDH).

```
99     static TPM_RC
100    C_2_2_ECDH(
101        TPMS_ECC_POINT          *outZs,          // OUT: Zs
102        TPMS_ECC_POINT          *outZe,          // OUT: Ze
103        TPM_ECC_CURVE            curveId,        // IN: the curve for the computations
104        TPM2B_ECC_PARAMETER     *dsA,            // IN: static private TPM key
105        TPM2B_ECC_PARAMETER     *deA,            // IN: ephemeral private TPM key
106        TPMS_ECC_POINT          *QsB,            // IN: static public party B key
107        TPMS_ECC_POINT          *QeB             // IN: ephemeral public party B key
108        )
109    {
110        CURVE_INITIALIZED(E, curveId);
111        ECC_INITIALIZED(bnAs, dsA);
112        ECC_INITIALIZED(bnAe, deA);
113        POINT_INITIALIZED(ecBs, QsB);
114        POINT_INITIALIZED(ecBe, QeB);
115        POINT(ecZ);
116        TPM_RC            retVal;
117    //
118        // Parameter checks
119        if(E == NULL)
120            ERROR_RETURN(TPM_RC_CURVE);
121        pAssert(outZs != NULL && dsA != NULL && deA != NULL && QsB != NULL
122                && QeB != NULL);
123
124        // Do the point multiply for the Zs value ([dsA]QsB)
125        retVal = BnPointMult(ecZ, ecBs, bnAs, NULL, NULL, E);
126        if(retVal == TPM_RC_SUCCESS)
127        {
128            // Convert the Zs value.
129            BnPointTo2B(outZs, ecZ, E);
130            // Do the point multiply for the Ze value ([deA]QeB)
131            retVal = BnPointMult(ecZ, ecBe, bnAe, NULL, NULL, E);
132            if(retVal == TPM_RC_SUCCESS)
133                BnPointTo2B(outZe, ecZ, E);
134        }
135    Exit:
136        CURVE_FREE(E);
137        return retVal;
138    }
```

10.2.10.2.4 CryptEcc2PhaseKeyExchange()

This function is the dispatch routine for the EC key exchange functions that use two ephemeral and two static keys.

November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

Error Returns	Meaning
TPM_RC_SCHEME	scheme is not defined

```
139 LIB_EXPORT TPM_RC
140 CryptEcc2PhaseKeyExchange(
141     TPMS_ECC_POINT          *outZ1,         // OUT: a computed point
142     TPMS_ECC_POINT          *outZ2,         // OUT: and optional second point
143     TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
144     TPM_ALG_ID               scheme,        // IN: the key exchange scheme
145     TPM2B_ECC_PARAMETER     *dsA,           // IN: static private TPM key
146     TPM2B_ECC_PARAMETER     *deA,           // IN: ephemeral private TPM key
147     TPMS_ECC_POINT          *QsB,           // IN: static public party B key
148     TPMS_ECC_POINT          *QeB            // IN: ephemeral public party B key
149     )
150 {
151     pAssert(outZ1 != NULL
152             && dsA != NULL && deA != NULL
153             && QsB != NULL && QeB != NULL);
154
155     // Initialize the output points so that they are empty until one of the
156     // functions decides otherwise
157     outZ1->x.b.size = 0;
158     outZ1->y.b.size = 0;
159     if(outZ2 != NULL)
160     {
161         outZ2->x.b.size = 0;
162         outZ2->y.b.size = 0;
163     }
164     switch(scheme)
165     {
166         case ALG_ECDH_VALUE:
167             return C_2_2_ECDH(outZ1, outZ2, curveId, dsA, deA, QsB, QeB);
168             break;
169 #if ALG_ECMQV
170         case ALG_ECMQV_VALUE:
171             return C_2_2_MQV(outZ1, curveId, dsA, deA, QsB, QeB);
172             break;
173 #endif
174 #if ALG_SM2
175         case ALG_SM2_VALUE:
176             return SM2KeyExchange(outZ1, curveId, dsA, deA, QsB, QeB);
177             break;
178 #endif
179         default:
180             return TPM_RC_SCHEME;
181     }
182 }
183 #if ALG_SM2
```

10.2.10.2.5 ComputeWForSM2()

Compute the value for w used by SM2

```
184 static UINT32
185 ComputeWForSM2(
186     bigCurve        E
187     )
188 {
189     //  w := ceil(ceil(log2(n)) / 2) - 1
190     return (BnMsb(CurveGetOrder(AccessCurveData(E))) / 2 - 1);
191 }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.2.10.2.6 avfSm2()

This function does the associated value computation required by SM2 key exchange. This is different from the avf() in the international standards because it returns a value that is half the size of the value returned by the standard avf(). For example, if *n* is 15, *Ws* (*w* in the standard) is 2 but the *W* here is 1. This means that an input value of 14 (1110b) would return a value of 110b with the standard but 10b with the scheme in SM2.

```
192 static bigNum
193 avfSm2(
194     bigNum              bn,           // IN/OUT: the reduced value
195     UINT32              w              // IN: the value of w
196     )
197 {
198     // a)   set w := ceil(ceil(log2(n)) / 2) - 1
199     // b)   set x' := 2^w + ( x & (2^w - 1))
200     // This is just like the avf for MQV where x' = 2^w + (x mod 2^w)
201
202     BnMaskBits(bn, w);   // as with avf1, this is too big by a factor of 2 but
203                          // it doesn't matter because we SET the extra bit
204                          // anyway
205     BnSetBit(bn, w);
206     return bn;
207 }
```

10.2.10.2.7 SM2KeyExchange()

This function performs the key exchange defined in SM2. The first step is to compute *tA* = (*dsA* + *deA* avf(Xe,A)) mod *n* Then, compute the *Z* value from *outZ* = (*h tA* mod *n*) (*QsA* + [avf(*QeB.x*)](*QeB*)). The function will compute the ephemeral public key from the ephemeral private key. All points are required to be on the curve of *inQsA*. The function will fail catastrophically if this is not the case

Error Returns	Meaning
TPM_RC_NO_RESULT	the value for *dsA* does not give a valid point on the curve

```
208 LIB_EXPORT TPM_RC
209 SM2KeyExchange(
210     TPMS_ECC_POINT         *outZ,         // OUT: the computed point
211     TPM_ECC_CURVE           curveId,      // IN: the curve for the computations
212     TPM2B_ECC_PARAMETER    *dsAIn,        // IN: static private TPM key
213     TPM2B_ECC_PARAMETER    *deAIn,        // IN: ephemeral private TPM key
214     TPMS_ECC_POINT         *QsBIn,        // IN: static public party B key
215     TPMS_ECC_POINT         *QeBIn         // IN: ephemeral public party B key
216     )
217 {
218     CURVE_INITIALIZED(E, curveId);
219     const ECC_CURVE_DATA      *C;
220     ECC_INITIALIZED(dsA, dsAIn);
221     ECC_INITIALIZED(deA, deAIn);
222     POINT_INITIALIZED(QsB, QsBIn);
223     POINT_INITIALIZED(QeB, QeBIn);
224     BN_WORD_INITIALIZED(One, 1);
225     POINT(QeA);
226     ECC_NUM(XeB);
227     POINT(Z);
228     ECC_NUM(Ta);
229     UINT32                     w;
230     TPM_RC                    retVal = TPM_RC_NO_RESULT;
231 //
232     // Parameter checks
233     if(E == NULL)
```

 Copyright © TCG 2006-2020

```
234            ERROR_RETURN(TPM_RC_CURVE);
235        C = AccessCurveData(E);
236        pAssert(outZ != NULL && dsA != NULL && deA != NULL &&  QsB != NULL
237                && QeB != NULL);
238
239        // Compute the value for w
240        w = ComputeWForSM2(E);
241
242        // Compute the public ephemeral key pQeA = [de,A]G
243        if(!BnEccModMult(QeA, CurveGetG(C), deA, E))
244            goto Exit;
245
246        //  tA := (ds,A + de,A  avf(Xe,A)) mod n    (3)
247        //  Compute 'tA' = ('dsA' +  'deA'  avf('XeA')) mod n
248        // Ta = avf(XeA);
249        // do Ta = de,A * Ta = deA * avf(XeA)
250        BnMult(Ta, deA, avfSm2(QeA->x, w));
251        // now Ta = dsA + Ta =  dsA + deA * avf(XeA)
252        BnAdd(Ta, dsA, Ta);
253        BnMod(Ta, CurveGetOrder(C));
254
255        //  outZ = [h  tA mod n] (Qs,B + [avf(Xe,B)](Qe,B))  (4)
256        // Put this in because almost every case of h is == 1 so skip the call when
257        // not necessary.
258        if(!BnEqualWord(CurveGetCofactor(C), 1))
259            // Cofactor is not 1 so compute Ta := Ta * h mod n
260            BnModMult(Ta, Ta, CurveGetCofactor(C), CurveGetOrder(C));
261        // Now that 'tA' is (h * 'tA' mod n)
262        // 'outZ' = ['tA'](QsB + [avf(QeB.x)](QeB)).
263        BnCopy(XeB, QeB->x);
264        if(!BnEccModMult2(Z, QsB, One, QeB, avfSm2(XeB, w), E))
265            goto Exit;
266        // QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
267        if(!BnEccModMult(Z, Z, Ta, E))
268            goto Exit;
269        // Convert BIGNUM E to TPM2B E
270        BnPointTo2B(outZ, Z, E);
271        retVal = TPM_RC_SUCCESS;
272    Exit:
273        CURVE_FREE(E);
274        return retVal;
275    }
276    #endif
277    #endif // CC_ZGen_2Phase
```

10.2.11 CryptEccMain.c

10.2.11.1 Includes and Defines

```
1  #include "Tpm.h"
2  #if ALG_ECC
```

This version requires that the new format for ECC data be used

```
3  #if !USE_BN_ECC_DATA
4  #error "Need to SET USE_BN_ECC_DATA to YES in Implementaion.h"
5  #endif
```

10.2.11.2 Functions

```
6  #if SIMULATION
7  void
8  EccSimulationEnd(
9      void
10     )
11 {
12 #if SIMULATION
13 // put things to be printed at the end of the simulation here
14 #endif
15 }
16 #endif // SIMULATION
```

10.2.11.2.1 CryptEccInit()

This function is called at _TPM_Init()

```
17 BOOL
18 CryptEccInit(
19     void
20     )
21 {
22     return TRUE;
23 }
```

10.2.11.2.2 CryptEccStartup()

This function is called at TPM2_Startup().

```
24 BOOL
25 CryptEccStartup(
26     void
27     )
28 {
29     return TRUE;
30 }
```

10.2.11.2.3 ClearPoint2B(generic

Initialize the size values of a TPMS_ECC_POINT structure.

```
31 void
32 ClearPoint2B(
33     TPMS_ECC_POINT      *p          // IN: the point
```

 Copyright © TCG 2006-2020

```
34          )
35      {
36          if(p != NULL)
37          {
38              p->x.t.size = 0;
39              p->y.t.size = 0;
40          }
41      }
```

10.2.11.2.4 CryptEccGetParametersByCurveId()

This function returns a pointer to the curve data that is associated with the indicated *curveId*. If there is no curve with the indicated ID, the function returns NULL. This function is in this module so that it can be called by GetCurve() data.

Return Value	Meaning
NULL	curve with the indicated TPM_ECC_CURVE is not implemented
NULL	pointer to the curve data

```
42      LIB_EXPORT const ECC_CURVE *
43      CryptEccGetParametersByCurveId(
44          TPM_ECC_CURVE       curveId     // IN: the curveID
45          )
46      {
47          int           i;
48          for(i = 0; i < ECC_CURVE_COUNT; i++)
49          {
50              if(eccCurves[i].curveId == curveId)
51                  return &eccCurves[i];
52          }
53          return NULL;
54      }
```

10.2.11.2.5 CryptEccGetKeySizeForCurve()

This function returns the key size in bits of the indicated curve.

```
55      LIB_EXPORT UINT16
56      CryptEccGetKeySizeForCurve(
57          TPM_ECC_CURVE            curveId    // IN: the curve
58          )
59      {
60          const ECC_CURVE *curve = CryptEccGetParametersByCurveId(curveId);
61          UINT16          keySizeInBits;
62      //
63          keySizeInBits = (curve != NULL) ? curve->keySizeBits : 0;
64          return keySizeInBits;
65      }
```

10.2.11.2.6 GetCurveData()

This function returns the a pointer for the parameter data associated with a curve.

```
66      const ECC_CURVE_DATA *
67      GetCurveData(
68          TPM_ECC_CURVE        curveId     // IN: the curveID
69          )
70      {
71          const ECC_CURVE      *curve = CryptEccGetParametersByCurveId(curveId);
```

```
72         return (curve != NULL) ? curve->curveData : NULL;
73     }
```

10.2.11.2.7 CryptEccGetOID()

```
74     const BYTE *
75     CryptEccGetOID(
76         TPM_ECC_CURVE        curveId
77     )
78     {
79         const ECC_CURVE          *curve = CryptEccGetParametersByCurveId(curveId);
80         return (curve != NULL) ? curve->OID : NULL;
81     }
```

10.2.11.2.8 CryptEccGetCurveByIndex()

This function returns the number of the *i*-th implemented curve. The normal use would be to call this function with *i* starting at 0. When the *i* is greater than or equal to the number of implemented curves, TPM_ECC_NONE is returned.

```
82     LIB_EXPORT TPM_ECC_CURVE
83     CryptEccGetCurveByIndex(
84         UINT16               i
85         )
86     {
87         if(i >= ECC_CURVE_COUNT)
88             return TPM_ECC_NONE;
89         return eccCurves[i].curveId;
90     }
```

10.2.11.2.9 CryptEccGetParameter()

This function returns an ECC curve parameter. The parameter is selected by a single character designator from the set of "PNABXYH".

Return Value	Meaning
TRUE(1)	curve exists and parameter returned
FALSE(0)	curve does not exist or parameter selector

```
91     LIB_EXPORT BOOL
92     CryptEccGetParameter(
93         TPM2B_ECC_PARAMETER     *out,       // OUT: place to put parameter
94         char                     p,         // IN: the parameter selector
95         TPM_ECC_CURVE            curveId    // IN: the curve id
96         )
97     {
98         const ECC_CURVE_DATA    *curve = GetCurveData(curveId);
99         bigConst                 parameter = NULL;
100
101        if(curve != NULL)
102        {
103            switch(p)
104            {
105                case 'p':
106                    parameter = CurveGetPrime(curve);
107                    break;
108                case 'n':
109                    parameter = CurveGetOrder(curve);
110                    break;
```

 Copyright © TCG 2006-2020

```
111             case 'a':
112                 parameter = CurveGet_a(curve);
113                 break;
114             case 'b':
115                 parameter = CurveGet_b(curve);
116                 break;
117             case 'x':
118                 parameter = CurveGetGx(curve);
119                 break;
120             case 'y':
121                 parameter = CurveGetGy(curve);
122                 break;
123             case 'h':
124                 parameter = CurveGetCofactor(curve);
125                 break;
126             default:
127                 FAIL(FATAL_ERROR_INTERNAL);
128                 break;
129         }
130     }
131     // If not debugging and we get here with parameter still NULL, had better
132     // not try to convert so just return FALSE instead.
133     return (parameter != NULL) ? BnTo2B(parameter, &out->b, 0) : 0;
134 }
```

10.2.11.2.10 CryptCapGetECCCurve()

This function returns the list of implemented ECC curves.

Return Value	Meaning
YES	if no more ECC curve is available
NO	if there are more ECC curves not reported

```
135 TPMI_YES_NO
136 CryptCapGetECCCurve(
137     TPM_ECC_CURVE    curveID,       // IN: the starting ECC curve
138     UINT32           maxCount,      // IN: count of returned curves
139     TPML_ECC_CURVE  *curveList      // OUT: ECC curve list
140     )
141 {
142     TPMI_YES_NO       more = NO;
143     UINT16            i;
144     UINT32            count = ECC_CURVE_COUNT;
145     TPM_ECC_CURVE     curve;
146
147     // Initialize output property list
148     curveList->count = 0;
149
150     // The maximum count of curves we may return is MAX_ECC_CURVES
151     if(maxCount > MAX_ECC_CURVES) maxCount = MAX_ECC_CURVES;
152
153     // Scan the eccCurveValues array
154     for(i = 0; i < count; i++)
155     {
156         curve = CryptEccGetCurveByIndex(i);
157         // If curveID is less than the starting curveID, skip it
158         if(curve < curveID)
159             continue;
160         if(curveList->count < maxCount)
161         {
162             // If we have not filled up the return list, add more curves to
163             // it
```

 Copyright © TCG 2006-2020

```
164                 curveList->eccCurves[curveList->count] = curve;
165                 curveList->count++;
166             }
167             else
168             {
169                 // If the return list is full but we still have curves
170                 // available, report this and stop iterating
171                 more = YES;
172                 break;
173             }
174         }
175         return more;
176     }
```

10.2.11.2.11 CryptGetCurveSignScheme()

This function will return a pointer to the scheme of the curve.

```
177     const TPMT_ECC_SCHEME *
178     CryptGetCurveSignScheme(
179         TPM_ECC_CURVE     curveId        // IN: The curve selector
180         )
181     {
182         const ECC_CURVE          *curve = CryptEccGetParametersByCurveId(curveId);
183
184         if(curve != NULL)
185             return &(curve->sign);
186         else
187             return NULL;
188     }
```

10.2.11.2.12 CryptGenerateR()

This function computes the commit random value for a split signing scheme.

If *c* is NULL, it indicates that *r* is being generated for TPM2_Commit(). If *c* is not NULL, the TPM will validate that the *gr.commitArray* bit associated with the input value of *c* is SET. If not, the TPM returns FALSE and no *r* value is generated.

Return Value	Meaning
TRUE(1)	r value computed
FALSE(0)	no r value computed

```
189     BOOL
190     CryptGenerateR(
191         TPM2B_ECC_PARAMETER     *r,             // OUT: the generated random value
192         UINT16                  *c,             // IN/OUT: count value.
193         TPMI_ECC_CURVE           curveID,       // IN: the curve for the value
194         TPM2B_NAME              *name           // IN: optional name of a key to
195                                                 //     associate with 'r'
196         )
197     {
198         // This holds the marshaled g_commitCounter.
199         TPM2B_TYPE(8B, 8);
200         TPM2B_8B                 cntr = {{8,{0}}};
201         UINT32                   iterations;
202         TPM2B_ECC_PARAMETER      n;
203         UINT64                   currentCount = gr.commitCounter;
204         UINT16                   t1;
205     //
206         if(!CryptEccGetParameter(&n, 'n', curveID))
```

```
207             return FALSE;
208
209         // If this is the commit phase, use the current value of the commit counter
210         if(c != NULL)
211         {
212             // if the array bit is not set, can't use the value.
213             if(!TEST_BIT((*c & COMMIT_INDEX_MASK), gr.commitArray))
214                 return FALSE;
215
216             // If it is the sign phase, figure out what the counter value was
217             // when the commitment was made.
218             //
219             // When gr.commitArray has less than 64K bits, the extra
220             // bits of 'c' are used as a check to make sure that the
221             // signing operation is not using an out of range count value
222             t1 = (UINT16)currentCount;
223
224             // If the lower bits of c are greater or equal to the lower bits of t1
225             // then the upper bits of t1 must be one more than the upper bits
226             // of c
227             if((*c & COMMIT_INDEX_MASK) >= (t1 & COMMIT_INDEX_MASK))
228                 // Since the counter is behind, reduce the current count
229                 currentCount = currentCount - (COMMIT_INDEX_MASK + 1);
230
231             t1 = (UINT16)currentCount;
232             if((t1 & ~COMMIT_INDEX_MASK) != (*c & ~COMMIT_INDEX_MASK))
233                 return FALSE;
234             // set the counter to the value that was
235             // present when the commitment was made
236             currentCount = (currentCount & 0xffffffffffff0000) | *c;
237         }
238         // Marshal the count value to a TPM2B buffer for the KDF
239         cntr.t.size = sizeof(currentCount);
240         UINT64_TO_BYTE_ARRAY(currentCount, cntr.t.buffer);
241
242         // Now can do the KDF to create the random value for the signing operation
243         // During the creation process, we may generate an r that does not meet the
244         // requirements of the random value.
245         // want to generate a new r.
246         r->t.size = n.t.size;
247
248         for(iterations = 1; iterations < 1000000;)
249         {
250             int     i;
251             CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &gr.commitNonce.b, COMMIT_STRING,
252                       &name->b, &cntr.b, n.t.size * 8, r->t.buffer, &iterations, FALSE);
253
254             // "random" value must be less than the prime
255             if(UnsignedCompareB(r->b.size, r->b.buffer, n.t.size, n.t.buffer) >= 0)
256                 continue;
257
258             // in this implementation it is required that at least bit
259             // in the upper half of the number be set
260             for(i = n.t.size / 2; i >= 0; i--)
261                 if(r->b.buffer[i] != 0)
262                     return TRUE;
263         }
264         return FALSE;
265     }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.2.11.2.13 CryptCommit()

This function is called when the count value is committed. The *gr.commitArray* value associated with the current count value is SET and *g_commitCounter* is incremented. The low-order 16 bits of old value of the counter is returned.

```
266 UINT16
267 CryptCommit(
268     void
269     )
270 {
271     UINT16      oldCount = (UINT16)gr.commitCounter;
272     gr.commitCounter++;
273     SET_BIT(oldCount & COMMIT_INDEX_MASK, gr.commitArray);
274     return oldCount;
275 }
```

10.2.11.2.14 CryptEndCommit()

This function is called when the signing operation using the committed value is completed. It clears the gr.*commitArray* bit associated with the count value so that it can't be used again.

```
276 void
277 CryptEndCommit(
278     UINT16          c               // IN: the counter value of the commitment
279     )
280 {
281     ClearBit((c & COMMIT_INDEX_MASK), gr.commitArray, sizeof(gr.commitArray));
282 }
```

10.2.11.2.15 CryptEccGetParameters()

This function returns the ECC parameter details of the given curve.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	unsupported ECC curve ID

```
283 BOOL
284 CryptEccGetParameters(
285     TPM_ECC_CURVE                curveId,       // IN: ECC curve ID
286     TPMS_ALGORITHM_DETAIL_ECC   *parameters     // OUT: ECC parameters
287     )
288 {
289     const ECC_CURVE             *curve = CryptEccGetParametersByCurveId(curveId);
290     const ECC_CURVE_DATA        *data;
291     BOOL                         found = curve != NULL;
292
293     if(found)
294     {
295         data = curve->curveData;
296         parameters->curveID = curve->curveId;
297         parameters->keySize = curve->keySizeBits;
298         parameters->kdf = curve->kdf;
299         parameters->sign = curve->sign;
300 //        BnTo2B(data->prime, &parameters->p.b, 0);
301         BnTo2B(data->prime, &parameters->p.b, parameters->p.t.size);
302         BnTo2B(data->a, &parameters->a.b, 0);
303         BnTo2B(data->b, &parameters->b.b, 0);
304         BnTo2B(data->base.x, &parameters->gX.b, parameters->p.t.size);
```

 Copyright © TCG 2006-2020

```
305             BnTo2B(data->base.y, &parameters->gY.b, parameters->p.t.size);
306     //          BnTo2B(data->base.x, &parameters->gX.b, 0);
307     //          BnTo2B(data->base.y, &parameters->gY.b, 0);
308             BnTo2B(data->order, &parameters->n.b, 0);
309             BnTo2B(data->h, &parameters->h.b, 0);
310         }
311         return found;
312     }
```

10.2.11.2.16 BnGetCurvePrime()

This function is used to get just the prime modulus associated with a curve.

```
313     const bignum_t *
314     BnGetCurvePrime(
315         TPM_ECC_CURVE            curveId
316         )
317     {
318         const ECC_CURVE_DATA    *C = GetCurveData(curveId);
319         return (C != NULL) ? CurveGetPrime(C) : NULL;
320     }
```

10.2.11.2.17 BnGetCurveOrder()

This function is used to get just the curve order

```
321     const bignum_t *
322     BnGetCurveOrder(
323         TPM_ECC_CURVE            curveId
324         )
325     {
326         const ECC_CURVE_DATA    *C = GetCurveData(curveId);
327         return (C != NULL) ? CurveGetOrder(C) : NULL;
328     }
```

10.2.11.2.18 BnIsOnCurve()

This function checks if a point is on the curve.

```
329     BOOL
330     BnIsOnCurve(
331         pointConst                   Q,
332         const ECC_CURVE_DATA        *C
333         )
334     {
335         BN_VAR(right, (MAX_ECC_KEY_BITS * 3));
336         BN_VAR(left, (MAX_ECC_KEY_BITS * 2));
337         bigConst                    prime = CurveGetPrime(C);
338     //
339         // Show that point is on the curve y^2 = x^3 + ax + b;
340         // Or y^2 = x(x^2 + a) + b
341         // y^2
342         BnMult(left, Q->y, Q->y);
343
344         BnMod(left, prime);
345     // x^2
346         BnMult(right, Q->x, Q->x);
347
348         // x^2 + a
349         BnAdd(right, right, CurveGet_a(C));
350
```

```
351 //    BnMod(right, CurveGetPrime(C));
352     // x(x^2 + a)
353     BnMult(right, right, Q->x);
354
355     // x(x^2 + a) + b
356     BnAdd(right, right, CurveGet_b(C));
357
358     BnMod(right, prime);
359     if(BnUnsignedCmp(left, right) == 0)
360         return TRUE;
361     else
362         return FALSE;
363 }
```

10.2.11.2.19 BnIsValidPrivateEcc()

Checks that 0 < *x* < *q*

```
364 BOOL
365 BnIsValidPrivateEcc(
366     bigConst                 x,         // IN: private key to check
367     bigCurve                 E          // IN: the curve to check
368     )
369 {
370     BOOL         retVal;
371     retVal = (!BnEqualZero(x)
372               && (BnUnsignedCmp(x, CurveGetOrder(AccessCurveData(E))) < 0));
373     return retVal;
374 }
375 LIB_EXPORT BOOL
376 CryptEccIsValidPrivateKey(
377     TPM2B_ECC_PARAMETER     *d,
378     TPM_ECC_CURVE            curveId
379     )
380 {
381     BN_INITIALIZED(bnD, MAX_ECC_PARAMETER_BYTES * 8, d);
382     return !BnEqualZero(bnD) && (BnUnsignedCmp(bnD, BnGetCurveOrder(curveId)) < 0);
383 }
```

10.2.11.2.20 BnPointMul()

This function does a point multiply of the form *R* = [*d*]*S* + [*u*]Q where the parameters are *bigNum* values. If *S* is NULL and d is not NULL, then it computes *R* = [*d*]*G* + [*u*]*Q* or just *R* = [*d*]*G* if *u* and *Q* are NULL. If *skipChecks* is TRUE, then the function will not verify that the inputs are correct for the domain. This would be the case when the values were created by the CryptoEngine() code. It will return TPM_RC_NO_RESULT if the resulting point is the point at infinity.

Error Returns	Meaning
TPM_RC_NO_RESULT	result of multiplication is a point at infinity
TPM_RC_ECC_POINT	*S* or *Q* is not on the curve
TPM_RC_VALUE	*d* or *u* is not < n

```
384 TPM_RC
385 BnPointMult(
386     bigPoint             R,          // OUT: computed point
387     pointConst           S,          // IN: optional point to multiply by 'd'
388     bigConst             d,          // IN: scalar for [d]S or [d]G
389     pointConst           Q,          // IN: optional second point
390     bigConst             u,          // IN: optional second scalar
```

```
391         bigCurve                  E            // IN: curve parameters
392         )
393     {
394         BOOL                      OK;
395     //
396         TEST(TPM_ALG_ECDH);
397
398         // Need one scalar
399         OK = (d != NULL || u != NULL);
400
401         // If S is present, then d has to be present. If S is not
402         // present, then d may or may not be present
403         OK = OK && (((S == NULL) == (d == NULL)) || (d != NULL));
404
405         // either both u and Q have to be provided or neither can be provided (don't
406         // know what to do if only one is provided.
407         OK = OK && ((u == NULL) == (Q == NULL));
408
409         OK = OK && (E != NULL);
410         if(!OK)
411             return TPM_RC_VALUE;
412
413         OK = (S == NULL) || BnIsOnCurve(S, AccessCurveData(E));
414         OK = OK && ((Q == NULL) || BnIsOnCurve(Q, AccessCurveData(E)));
415         if(!OK)
416             return TPM_RC_ECC_POINT;
417
418         if((d != NULL) && (S == NULL))
419             S = CurveGetG(AccessCurveData(E));
420         // If only one scalar, don't need Shamir's trick
421         if((d == NULL) || (u == NULL))
422         {
423             if(d == NULL)
424                 OK = BnEccModMult(R, Q, u, E);
425             else
426                 OK = BnEccModMult(R, S, d, E);
427         }
428         else
429         {
430             OK = BnEccModMult2(R, S, d, Q, u, E);
431         }
432         return  (OK ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT);
433     }
```

10.2.11.2.21 BnEccGetPrivate()

This function gets random values that are the size of the key plus 64 bits. The value is reduced (mod (q - 1)) and incremented by 1 (q is the order of the curve. This produces a value (d) such that 1 <= d < q. This is the method of FIPS 186-4 Section B.4.1 "Key Pair Generation Using Extra Random Bits".

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure generating private key

```
434     BOOL
435     BnEccGetPrivate(
436         bigNum                     dOut,        // OUT: the qualified random value
437         const ECC_CURVE_DATA      *C,           // IN: curve for which the private key
438                                                 //     needs to be appropriate
439         RAND_STATE                *rand         // IN: state for DRBG
440         )
441     {
```

```
442       bigConst                  order = CurveGetOrder(C);
443       BOOL                      OK;
444       UINT32                    orderBits = BnSizeInBits(order);
445       UINT32                    orderBytes = BITS_TO_BYTES(orderBits);
446       BN_VAR(bnExtraBits, MAX_ECC_KEY_BITS + 64);
447       BN_VAR(nMinus1, MAX_ECC_KEY_BITS);
448   //
449       OK = BnGetRandomBits(bnExtraBits, (orderBytes * 8) + 64, rand);
450       OK = OK && BnSubWord(nMinus1, order, 1);
451       OK = OK && BnMod(bnExtraBits, nMinus1);
452       OK = OK && BnAddWord(dOut, bnExtraBits, 1);
453       return OK && !g_inFailureMode;
454   }
```

10.2.11.2.22 BnEccGenerateKeyPair()

This function gets a private scalar from the source of random bits and does the point multiply to get the public key.

```
455   BOOL
456   BnEccGenerateKeyPair(
457       bigNum               bnD,             // OUT: private scalar
458       bn_point_t          *ecQ,             // OUT: public point
459       bigCurve             E,               // IN: curve for the point
460       RAND_STATE          *rand             // IN: DRBG state to use
461       )
462   {
463       BOOL                 OK = FALSE;
464       // Get a private scalar
465       OK = BnEccGetPrivate(bnD, AccessCurveData(E), rand);
466
467       // Do a point multiply
468       OK = OK && BnEccModMult(ecQ, NULL, bnD, E);
469       if(!OK)
470           BnSetWord(ecQ->z, 0);
471       else
472           BnSetWord(ecQ->z, 1);
473       return OK;
474   }
```

10.2.11.2.23 CryptEccNewKeyPair

This function creates an ephemeral ECC. It is ephemeral in that is expected that the private part of the key will be discarded

```
475   LIB_EXPORT TPM_RC
476   CryptEccNewKeyPair(
477       TPMS_ECC_POINT          *Qout,      // OUT: the public point
478       TPM2B_ECC_PARAMETER     *dOut,      // OUT: the private scalar
479       TPM_ECC_CURVE            curveId    // IN: the curve for the key
480       )
481   {
482       CURVE_INITIALIZED(E, curveId);
483       POINT(ecQ);
484       ECC_NUM(bnD);
485       BOOL                    OK;
486
487       if(E == NULL)
488           return TPM_RC_CURVE;
489
490       TEST(TPM_ALG_ECDH);
491       OK = BnEccGenerateKeyPair(bnD, ecQ, E, NULL);
```

```
492         if(OK)
493         {
494             BnPointTo2B(Qout, ecQ, E);
495             BnTo2B(bnD, &dOut->b, Qout->x.t.size);
496         }
497         else
498         {
499             Qout->x.t.size = Qout->y.t.size = dOut->t.size = 0;
500         }
501         CURVE_FREE(E);
502         return OK ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT;
503     }
```

10.2.11.2.24 CryptEccPointMultiply()

This function computes *R* := [*dIn*]*G* + [*uIn*]*QIn*. Where *dIn* and *uIn* are scalars, *G* and *QIn* are points on the specified curve and *G* is the default generator of the curve.

The *xOut* and *yOut* parameters are optional and may be set to NULL if not used.

It is not necessary to provide *uIn* if *QIn* is specified but one of *uIn* and *dIn* must be provided. If *dIn* and *QIn* are specified but *uIn* is not provided, then *R* = [*dIn*]*QIn*.

If the multiply produces the point at infinity, the TPM_RC_NO_RESULT is returned.

The sizes of *xOut* and *yOut*' will be set to be the size of the degree of the curve

It is a fatal error if *dIn* and *uIn* are both unspecified (NULL) or if *Qin* or *Rout* is unspecified.

Error Returns	Meaning
TPM_RC_ECC_POINT	the point *Pin* or *Qin* is not on the curve
TPM_RC_NO_RESULT	the product point is at infinity
TPM_RC_CURVE	bad curve
TPM_RC_VALUE	*dIn* or *uIn* out of range

```
504     LIB_EXPORT TPM_RC
505     CryptEccPointMultiply(
506         TPMS_ECC_POINT      *Rout,              // OUT: the product point R
507         TPM_ECC_CURVE        curveId,           // IN: the curve to use
508         TPMS_ECC_POINT      *Pin,               // IN: first point (can be null)
509         TPM2B_ECC_PARAMETER *dIn,               // IN: scalar value for [dIn]Qin
510                                                 //     the Pin
511         TPMS_ECC_POINT      *Qin,               // IN: point Q
512         TPM2B_ECC_PARAMETER *uIn                // IN: scalar value for the multiplier
513                                                 //     of Q
514         )
515     {
516         CURVE_INITIALIZED(E, curveId);
517         POINT_INITIALIZED(ecP, Pin);
518         ECC_INITIALIZED(bnD, dIn);       // If dIn is null, then bnD is null
519         ECC_INITIALIZED(bnU, uIn);
520         POINT_INITIALIZED(ecQ, Qin);
521         POINT(ecR);
522         TPM_RC             retVal;
523     //
524         retVal = BnPointMult(ecR, ecP, bnD, ecQ, bnU, E);
525
526         if(retVal == TPM_RC_SUCCESS)
527             BnPointTo2B(Rout, ecR, E);
528         else
529             ClearPoint2B(Rout);
530         CURVE_FREE(E);
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
531     return retVal;
532 }
```

10.2.11.2.25 CryptEccIsPointOnCurve()

This function is used to test if a point is on a defined curve. It does this by checking that *y*^2 mod *p* = *x*^3 + *a***x* + *b* mod *p*.

It is a fatal error if *Q* is not specified (is NULL).

Return Value	Meaning
TRUE(1)	point is on curve
FALSE(0)	point is not on curve or curve is not supported

```
533 LIB_EXPORT BOOL
534 CryptEccIsPointOnCurve(
535     TPM_ECC_CURVE            curveId,        // IN: the curve selector
536     TPMS_ECC_POINT          *Qin            // IN: the point.
537     )
538 {
539     const ECC_CURVE_DATA    *C = GetCurveData(curveId);
540     POINT_INITIALIZED(ecQ, Qin);
541     BOOL                OK;
542 //
543     pAssert(Qin != NULL);
544     OK = (C != NULL && (BnIsOnCurve(ecQ, C)));
545     return OK;
546 }
```

10.2.11.2.26 CryptEccGenerateKey()

This function generates an ECC key pair based on the input parameters. This routine uses KDFa to produce candidate numbers. The method is according to FIPS 186-3, section B.1.2 "Key Pair Generation by Testing Candidates." According to the method in FIPS 186-3, the resulting private value *d* should be 1 <= *d* < *n* where *n* is the order of the base point.

It is a fatal error if *Qout*, *dOut*, is not provided (is NULL).

If the curve is not supported If *seed* is not provided, then a random number will be used for the key

Error Returns	Meaning
TPM_RC_CURVE	curve is not supported
TPM_RC_NO_RESULT	could not verify key with signature (FIPS only)

```
547 LIB_EXPORT TPM_RC
548 CryptEccGenerateKey(
549     TPMT_PUBLIC         *publicArea,        // IN/OUT: The public area template for
550                                             //      the new key. The public key
551                                             //      area will be replaced computed
552                                             //      ECC public key
553     TPMT_SENSITIVE      *sensitive,         // OUT: the sensitive area will be
554                                             //      updated to contain the private
555                                             //      ECC key and the symmetric
556                                             //      encryption key
557     RAND_STATE          *rand               // IN: if not NULL, the deterministic
558                                             //     RNG state
559     )
560 {
561     CURVE_INITIALIZED(E, publicArea->parameters.eccDetail.curveID);
```

```
562     ECC_NUM(bnD);
563     POINT(ecQ);
564     BOOL                    OK;
565     TPM_RC                  retVal;
566 //
567     TEST(TPM_ALG_ECDSA); // ECDSA is used to verify each key
568
569     // Validate parameters
570     if(E == NULL)
571         ERROR_RETURN(TPM_RC_CURVE);
572
573     publicArea->unique.ecc.x.t.size = 0;
574     publicArea->unique.ecc.y.t.size = 0;
575     sensitive->sensitive.ecc.t.size = 0;
576
577     OK = BnEccGenerateKeyPair(bnD, ecQ, E, rand);
578     if(OK)
579     {
580         BnPointTo2B(&publicArea->unique.ecc, ecQ, E);
581         BnTo2B(bnD, &sensitive->sensitive.ecc.b, publicArea->unique.ecc.x.t.size);
582     }
583 #if FIPS_COMPLIANT
584     // See if PWCT is required
585     if(OK && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
586     {
587         ECC_NUM(bnT);
588         ECC_NUM(bnS);
589         TPM2B_DIGEST            digest;
590 //
591         TEST(TPM_ALG_ECDSA);
592         digest.t.size = MIN(sensitive->sensitive.ecc.t.size, sizeof(digest.t.buffer));
593         // Get a random value to sign using the built in DRBG state
594         DRBG_Generate(NULL, digest.t.buffer, digest.t.size);
595         if(g_inFailureMode)
596             return TPM_RC_FAILURE;
597         BnSignEcdsa(bnT, bnS, E, bnD, &digest, NULL);
598         // and make sure that we can validate the signature
599         OK = BnValidateSignatureEcdsa(bnT, bnS, E, ecQ, &digest) == TPM_RC_SUCCESS;
600     }
601 #endif
602     retVal = (OK) ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT;
603 Exit:
604     CURVE_FREE(E);
605     return retVal;
606 }
607 #endif  // ALG_ECC
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.2.12 CryptEccSignature.c

10.2.12.1 Includes and Defines

```
1 #include "Tpm.h"
2 #include "CryptEccSignature_fp.h"
3 #if ALG_ECC
```

10.2.12.2 Utility Functions

10.2.12.2.1 EcdsaDigest()

Function to adjust the digest so that it is no larger than the order of the curve. This is used for ECDSA sign and verification.

```
4 static bigNum
5 EcdsaDigest(
6     bigNum              bnD,           // OUT: the adjusted digest
7     const TPM2B_DIGEST  *digest,       // IN: digest to adjust
8     bigConst            max            // IN: value that indicates the maximum
9                                        //     number of bits in the results
10    )
11 {
12     int                 bitsInMax = BnSizeInBits(max);
13     int                 shift;
14 //
15     if(digest == NULL)
16         BnSetWord(bnD, 0);
17     else
18     {
19         BnFromBytes(bnD, digest->t.buffer,
20                     (NUMBYTES)MIN(digest->t.size, BITS_TO_BYTES(bitsInMax)));
21         shift = BnSizeInBits(bnD) - bitsInMax;
22         if(shift > 0)
23             BnShiftRight(bnD, bnD, shift);
24     }
25     return bnD;
26 }
```

10.2.12.2.2 BnSchnorrSign()

This contains the Schnorr signature computation. It is used by both ECDSA and Schnorr signing. The result is computed as: [$s = k + r * d \pmod{n}$] where

a) s is the signature

b) k is a random value

c) r is the value to sign

d) d is the private EC key

e) n is the order of the curve

Error Returns	Meaning
TPM_RC_NO_RESULT	the result of the operation was zero or $r \pmod{n}$ is zero

```
27 static TPM_RC
28 BnSchnorrSign(
29     bigNum                  bnS,         // OUT: 's' component of the signature
```

 Copyright © TCG 2006-2020

```
30      bigConst                bnK,          // IN: a random value
31      bigNum                  bnR,          // IN: the signature 'r' value
32      bigConst                bnD,          // IN: the private key
33      bigConst                bnN           // IN: the order of the curve
34      )
35  {
36      // Need a local temp value to store the intermediate computation because product
37      // size can be larger than will fit in bnS.
38      BN_VAR(bnT1, MAX_ECC_PARAMETER_BYTES * 2 * 8);
39  //
40      // Reduce bnR without changing the input value
41      BnDiv(NULL, bnT1, bnR, bnN);
42      if(BnEqualZero(bnT1))
43          return TPM_RC_NO_RESULT;
44      // compute s = (k + r * d)(mod n)
45          // r * d
46      BnMult(bnT1, bnT1, bnD);
47          // k * r * d
48      BnAdd(bnT1, bnT1, bnK);
49          // k + r * d (mod n)
50      BnDiv(NULL, bnS, bnT1, bnN);
51      return (BnEqualZero(bnS)) ? TPM_RC_NO_RESULT : TPM_RC_SUCCESS;
52  }
```

10.2.12.3 Signing Functions

10.2.12.3.1 BnSignEcdsa()

This function implements the ECDSA signing algorithm. The method is described in the comments below.

```
53  TPM_RC
54  BnSignEcdsa(
55      bigNum                  bnR,          // OUT: 'r' component of the signature
56      bigNum                  bnS,          // OUT: 's' component of the signature
57      bigCurve                E,            // IN: the curve used in the signature
58                                            //     process
59      bigNum                  bnD,          // IN: private signing key
60      const TPM2B_DIGEST     *digest,       // IN: the digest to sign
61      RAND_STATE             *rand          // IN: used in debug of signing
62      )
63  {
64      ECC_NUM(bnK);
65      ECC_NUM(bnIk);
66      BN_VAR(bnE, MAX(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE) * 8);
67      POINT(ecR);
68      bigConst               order = CurveGetOrder(AccessCurveData(E));
69      TPM_RC                 retVal = TPM_RC_SUCCESS;
70      INT32                  tries = 10;
71      BOOL                   OK = FALSE;
72  //
73      pAssert(digest != NULL);
74      // The algorithm as described in "Suite B Implementer's Guide to FIPS
75      // 186-3(ECDSA)"
76      // 1. Use one of the routines in Appendix A.2 to generate (k, k^-1), a
77      //    per-message secret number and its inverse modulo n. Since n is prime,
78      //    the output will be invalid only if there is a failure in the RBG.
79      // 2. Compute the elliptic curve point R = [k]G = (xR, yR) using EC scalar
80      //    multiplication (see [Routines]), where G is the base point included in
81      //    the set of domain parameters.
82      // 3. Compute r = xR mod n. If r = 0, then return to Step 1. 1.
83      // 4. Use the selected hash function to compute H = Hash(M).
84      // 5. Convert the bit string H to an integer e as described in Appendix B.2.
85      // 6. Compute s = (k^-1 *  (e + d *  r)) mod q. If s = 0, return to Step 1.2.
```

```
86      // 7. Return (r, s).
87      // In the code below, q is n (that it, the order of the curve is p)
88
89      do // This implements the loop at step 6. If s is zero, start over.
90      {
91          for(; tries > 0; tries--)
92          {
93              // Step 1 and 2 -- generate an ephemeral key and the modular inverse
94              // of the private key.
95              if(!BnEccGenerateKeyPair(bnK, ecR, E, rand))
96                  continue;
97           // x coordinate is mod p.  Make it mod q
98              BnMod(ecR->x, order);
99              // Make sure that it is not zero;
100             if(BnEqualZero(ecR->x))
101                 continue;
102             // write the modular reduced version of r as part of the signature
103             BnCopy(bnR, ecR->x);
104             // Make sure that a modular inverse exists and try again if not
105             OK = (BnModInverse(bnIk, bnK, order));
106             if(OK)
107                 break;
108         }
109         if(!OK)
110             goto Exit;
111
112         EcdsaDigest(bnE, digest, order);
113
114         // now have inverse of K (bnIk), e (bnE), r (bnR),  d (bnD) and
115         // CurveGetOrder(E)
116         // Compute s = k^-1  (e + r*d)(mod q)
117         //  first do s = r*d mod q
118         BnModMult(bnS, bnR, bnD, order);
119         // s = e + s = e + r * d
120         BnAdd(bnS, bnE, bnS);
121         // s = k^(-1)s  (mod n)  = k^(-1)(e + r * d)(mod n)
122         BnModMult(bnS, bnIk, bnS, order);
123
124         // If S is zero, try again
125     } while(BnEqualZero(bnS));
126 Exit:
127     return retVal;
128 }
129 #if ALG_ECDAA
```

10.2.12.3.2 BnSignEcdaa()

This function performs $s = r + T * d \bmod q$ where

a) 'r is a random, or pseudo-random value created in the commit phase

b) *nonceK* is a TPM-generated, random value 0 < *nonceK* < *n*

c) *T* is mod *q* of **Hash**(*nonceK* || *digest*), and

d) *d* is a private key.

The signature is the tuple (*nonceK*, *s*)

Regrettably, the parameters in this function kind of collide with the parameter names used in ECSCHNORR making for a lot of confusion.

Error Returns	Meaning
TPM_RC_SCHEME	unsupported hash algorithm
TPM_RC_NO_RESULT	cannot get values from random number generator

```
130 static TPM_RC
131 BnSignEcdaa(
132     TPM2B_ECC_PARAMETER     *nonceK,        // OUT: 'nonce' component of the signature
133     bigNum                   bnS,           // OUT: 's' component of the signature
134     bigCurve                 E,             // IN: the curve used in signing
135     bigNum                   bnD,           // IN: the private key
136     const TPM2B_DIGEST      *digest,        // IN: the value to sign (mod 'q')
137     TPMT_ECC_SCHEME         *scheme,        // IN: signing scheme (contains the
138                                             //      commit count value).
139     OBJECT                  *eccKey,        // IN: The signing key
140     RAND_STATE              *rand           // IN: a random number state
141     )
142 {
143     TPM_RC                   retVal;
144     TPM2B_ECC_PARAMETER      r;
145     HASH_STATE               state;
146     TPM2B_DIGEST             T;
147     BN_MAX(bnT);
148 //
149     NOT_REFERENCED(rand);
150     if(!CryptGenerateR(&r, &scheme->details.ecdaa.count,
151                        eccKey->publicArea.parameters.eccDetail.curveID,
152                        &eccKey->name))
153         retVal = TPM_RC_VALUE;
154     else
155     {
156         // This allocation is here because 'r' doesn't have a value until
157         // CrypGenerateR() is done.
158         ECC_INITIALIZED(bnR, &r);
159         do
160         {
161             // generate nonceK such that 0 < nonceK < n
162             // use bnT as a temp.
163             if(!BnEccGetPrivate(bnT, AccessCurveData(E), rand))
164             {
165                 retVal = TPM_RC_NO_RESULT;
166                 break;
167             }
168             BnTo2B(bnT, &nonceK->b, 0);
169
170             T.t.size = CryptHashStart(&state, scheme->details.ecdaa.hashAlg);
171             if(T.t.size == 0)
172             {
173                 retVal = TPM_RC_SCHEME;
174             }
175             else
176             {
177                 CryptDigestUpdate2B(&state, &nonceK->b);
178                 CryptDigestUpdate2B(&state, &digest->b);
179                 CryptHashEnd2B(&state, &T.b);
180                 BnFrom2B(bnT, &T.b);
181                 // Watch out for the name collisions in this call!!
182                 retVal = BnSchnorrSign(bnS, bnR, bnT, bnD,
183                                        AccessCurveData(E)->order);
184             }
185         } while(retVal == TPM_RC_NO_RESULT);
186         // Because the rule is that internal state is not modified if the command
187         // fails, only end the commit if the command succeeds.
188         // NOTE that if the result of the Schnorr computation was zero
```


 Copyright © TCG 2006-2020

```
189             // it will probably not be worthwhile to run the same command again because
190             // the result will still be zero. This means that the Commit command will
191             // need to be run again to get a new commit value for the signature.
192             if(retVal == TPM_RC_SUCCESS)
193                 CryptEndCommit(scheme->details.ecdaa.count);
194         }
195         return retVal;
196     }
197     #endif // ALG_ECDAA
198     #if ALG_ECSCHNORR
```

10.2.12.3.3 SchnorrReduce()

Function to reduce a hash result if it's magnitude is too large. The size of *number* is set so that it has no more bytes of significance than *reference* value. If the resulting number can have more bits of significance than *reference*.

```
199     static void
200     SchnorrReduce(
201         TPM2B        *number,          // IN/OUT: Value to reduce
202         bigConst      reference        // IN: the reference value
203         )
204     {
205         UINT16       maxBytes = (UINT16)BITS_TO_BYTES(BnSizeInBits(reference));
206         if(number->size > maxBytes)
207             number->size = maxBytes;
208     }
```

10.2.12.3.4 SchnorrEcc()

This function is used to perform a modified Schnorr signature.

This function will generate a random value *k* and compute

a) $(xR, yR) = [k]G$

b) $r =$ **Hash**$(xR \,||\, P)(\text{mod } q)$

c) $rT =$ truncated r

d) $s = k + rT * ds \,(\text{mod } q)$

e) return the tuple rT, s

Error Returns	Meaning
TPM_RC_NO_RESULT	failure in the Schnorr sign process
TPM_RC_SCHEME	*hashAlg* can't produce zero-length digest

```
209     static TPM_RC
210     BnSignEcSchnorr(
211         bigNum                    bnR,            // OUT: 'r' component of the signature
212         bigNum                    bnS,            // OUT: 's' component of the signature
213         bigCurve                  E,              // IN: the curve used in signing
214         bigNum                    bnD,            // IN: the signing key
215         const TPM2B_DIGEST       *digest,         // IN: the digest to sign
216         TPM_ALG_ID                hashAlg,        // IN: signing scheme (contains a hash)
217         RAND_STATE               *rand            // IN: non-NULL when testing
218         )
219     {
220         HASH_STATE                hashState;
221         UINT16                    digestSize = CryptHashGetDigestSize(hashAlg);
222         TPM2B_TYPE(T, MAX(MAX_DIGEST_SIZE, MAX_ECC_KEY_BYTES));
```


 Copyright © TCG 2006-2020

```
223        TPM2B_T                     T2b;
224        TPM2B                      *e = &T2b.b;
225        TPM_RC                      retVal = TPM_RC_NO_RESULT;
226        const ECC_CURVE_DATA       *C;
227        bigConst                    order;
228        bigConst                    prime;
229        ECC_NUM(bnK);
230        POINT(ecR);
231    //
232        // Parameter checks
233        if(E == NULL)
234            ERROR_RETURN(TPM_RC_VALUE);
235        C = AccessCurveData(E);
236        order = CurveGetOrder(C);
237        prime = CurveGetOrder(C);
238
239        // If the digest does not produce a hash, then null the signature and return
240        // a failure.
241        if(digestSize == 0)
242        {
243            BnSetWord(bnR, 0);
244            BnSetWord(bnS, 0);
245            ERROR_RETURN(TPM_RC_SCHEME);
246        }
247        do
248        {
249            // Generate a random key pair
250            if(!BnEccGenerateKeyPair(bnK, ecR, E, rand))
251                break;
252            // Convert R.x to a string
253            BnTo2B(ecR->x, e, (NUMBYTES)BITS_TO_BYTES(BnSizeInBits(prime)));
254
255            // f) compute r = Hash(e || P) (mod n)
256            CryptHashStart(&hashState, hashAlg);
257            CryptDigestUpdate2B(&hashState, e);
258            CryptDigestUpdate2B(&hashState, &digest->b);
259            e->size = CryptHashEnd(&hashState, digestSize, e->buffer);
260            // Reduce the hash size if it is larger than the curve order
261            SchnorrReduce(e, order);
262            // Convert hash to number
263            BnFrom2B(bnR, e);
264            // Do the Schnorr computation
265            retVal = BnSchnorrSign(bnS, bnK, bnR, bnD, CurveGetOrder(C));
266        } while(retVal == TPM_RC_NO_RESULT);
267    Exit:
268        return retVal;
269    }
270    #endif // ALG_ECSCHNORR
271    #if ALG_SM2
272    #ifdef  _SM2_SIGN_DEBUG
```

10.2.12.3.5 BnHexEqual()

This function compares a bignum value to a hex string.

Return Value	Meaning
TRUE(1)	values equal
FALSE(0)	values not equal

```
273    static BOOL
274    BnHexEqual(
275        bigNum              bn,         //IN: big number value
```

 Copyright © TCG 2006-2020

```
276        const char       *c           //IN: character string number
277        )
278    {
279        ECC_NUM(bnC);
280        BnFromHex(bnC, c);
281        return (BnUnsignedCmp(bn, bnC) == 0);
282    }
283    #endif // _SM2_SIGN_DEBUG
```

10.2.12.3.6 BnSignEcSm2()

This function signs a digest using the method defined in SM2 Part 2. The method in the standard will add a header to the message to be signed that is a hash of the values that define the key. This then hashed with the message to produce a digest (*e*). This function signs *e*.

Error Returns	Meaning
TPM_RC_VALUE	bad curve

```
284    static TPM_RC
285    BnSignEcSm2(
286        bigNum                  bnR,        // OUT: 'r' component of the signature
287        bigNum                  bnS,        // OUT: 's' component of the signature
288        bigCurve                E,          // IN: the curve used in signing
289        bigNum                  bnD,        // IN: the private key
290        const TPM2B_DIGEST     *digest,     // IN: the digest to sign
291        RAND_STATE             *rand        // IN: random number generator (mostly for
292                                            //     debug)
293        )
294    {
295        BN_MAX_INITIALIZED(bnE, digest);    // Don't know how big digest might be
296        ECC_NUM(bnN);
297        ECC_NUM(bnK);
298        ECC_NUM(bnT);                       // temp
299        POINT(Q1);
300        bigConst                  order = (E != NULL)
301            ? CurveGetOrder(AccessCurveData(E)) : NULL;
302    //
303    #ifdef _SM2_SIGN_DEBUG
304        BnFromHex(bnE, "B524F552CD82B8B028476E005C377FB1"
305                       "9A87E6FC682D48BB5D42E3D9B9EFFE76");
306        BnFromHex(bnD, "128B2FA8BD433C6C068C8D803DFF7979"
307                       "2A519A55171B1B650C23661D15897263");
308    #endif
309        // A3: Use random number generator to generate random number 1 <= k <= n-1;
310        // NOTE: Ax: numbers are from the SM2 standard
311    loop:
312        {
313            // Get a random number 0 < k < n
314            BnGenerateRandomInRange(bnK, order, rand);
315    #ifdef _SM2_SIGN_DEBUG
316            BnFromHex(bnK, "6CB28D99385C175C94F94E934817663F"
317                           "C176D925DD72B727260DBAAE1FB2F96F");
318    #endif
319            // A4: Figure out the point of elliptic curve (x1, y1)=[k]G, and according
320            // to details specified in 4.2.7 in Part 1 of this document, transform the
321            // data type of x1 into an integer;
322            if(!BnEccModMult(Q1, NULL, bnK, E))
323                goto loop;
324            // A5: Figure out 'r' = ('e' + 'x1') mod 'n',
325            BnAdd(bnR, bnE, Q1->x);
326            BnMod(bnR, order);
327    #ifdef _SM2_SIGN_DEBUG
328            pAssert(BnHexEqual(bnR, "40F1EC59F793D9F49E09DCEF49130D41"
```

```
329                                    "94F79FB1EED2CAA55BACDB49C4E755D1"));
330 #endif
331         // if r=0 or r+k=n, return to A3;
332         if(BnEqualZero(bnR))
333             goto loop;
334         BnAdd(bnT, bnK, bnR);
335         if(BnUnsignedCmp(bnT, bnN) == 0)
336             goto loop;
337         // A6: Figure out s = ((1 + dA)^-1  (k - r  dA)) mod n,
338         // if s=0, return to A3;
339         // compute t = (1+dA)^-1
340         BnAddWord(bnT, bnD, 1);
341         BnModInverse(bnT, bnT, order);
342 #ifdef _SM2_SIGN_DEBUG
343         pAssert(BnHexEqual(bnT, "79BFCF3052C80DA7B939E0C6914A18CB"
344                                    "B2D96D8555256E83122743A7D4F5F956"));
345 #endif
346         // compute s = t * (k - r * dA) mod n
347         BnModMult(bnS, bnR, bnD, order);
348         // k - r * dA mod n = k + n - ((r * dA) mod n)
349         BnSub(bnS, order, bnS);
350         BnAdd(bnS, bnK, bnS);
351         BnModMult(bnS, bnS, bnT, order);
352 #ifdef _SM2_SIGN_DEBUG
353         pAssert(BnHexEqual(bnS, "6FC6DAC32C5D5CF10C77DFB20F7C2EB6"
354                                    "67A457872FB09EC56327A67EC7DEEBE7"));
355 #endif
356         if(BnEqualZero(bnS))
357             goto loop;
358     }
359     // A7: According to details specified in 4.2.1 in Part 1 of this document,
360     // transform the data type of r, s into bit strings, signature of message M
361     // is (r, s).
362     // This is handled by the common return code
363 #ifdef _SM2_SIGN_DEBUG
364     pAssert(BnHexEqual(bnR, "40F1EC59F793D9F49E09DCEF49130D41"
365                             "94F79FB1EED2CAA55BACDB49C4E755D1"));
366     pAssert(BnHexEqual(bnS, "6FC6DAC32C5D5CF10C77DFB20F7C2EB6"
367                             "67A457872FB09EC56327A67EC7DEEBE7"));
368 #endif
369     return TPM_RC_SUCCESS;
370 }
371 #endif // ALG_SM2
```

10.2.12.3.7 CryptEccSign()

This function is the dispatch function for the various ECC-based signing schemes. There is a bit of ugliness to the parameter passing. In order to test this, we sometime would like to use a deterministic RNG so that we can get the same signatures during testing. The easiest way to do this for most schemes is to pass in a deterministic RNG and let it return canned values during testing. There is a competing need for a canned parameter to use in ECDAA. To accommodate both needs with minimal fuss, a special type of RAND_STATE is defined to carry the address of the commit value. The setup and handling of this is not very different for the caller than what was in previous versions of the code.

Error Returns	Meaning
TPM_RC_SCHEME	*scheme* is not supported

```
372 LIB_EXPORT TPM_RC
373 CryptEccSign(
374     TPMT_SIGNATURE         *signature,      // OUT: signature
375     OBJECT                 *signKey,        // IN: ECC key to sign the hash
376     const TPM2B_DIGEST     *digest,         // IN: digest to sign
```


 Copyright © TCG 2006-2020

```
377      TPMT_ECC_SCHEME         *scheme,        // IN: signing scheme
378      RAND_STATE              *rand
379      )
380  {
381      CURVE_INITIALIZED(E, signKey->publicArea.parameters.eccDetail.curveID);
382      ECC_INITIALIZED(bnD, &signKey->sensitive.sensitive.ecc.b);
383      ECC_NUM(bnR);
384      ECC_NUM(bnS);
385      const ECC_CURVE_DATA   *C;
386      TPM_RC                  retVal = TPM_RC_SCHEME;
387  //
388      NOT_REFERENCED(scheme);
389      if(E == NULL)
390          ERROR_RETURN(TPM_RC_VALUE);
391       C = AccessCurveData(E);
392      signature->signature.ecdaa.signatureR.t.size
393          = sizeof(signature->signature.ecdaa.signatureR.t.buffer);
394      signature->signature.ecdaa.signatureS.t.size
395          = sizeof(signature->signature.ecdaa.signatureS.t.buffer);
396      TEST(signature->sigAlg);
397      switch(signature->sigAlg)
398      {
399          case ALG_ECDSA_VALUE:
400              retVal = BnSignEcdsa(bnR, bnS, E, bnD, digest, rand);
401              break;
402  #if ALG_ECDAA
403          case ALG_ECDAA_VALUE:
404              retVal = BnSignEcdaa(&signature->signature.ecdaa.signatureR, bnS, E,
405                                   bnD, digest, scheme, signKey, rand);
406              bnR = NULL;
407              break;
408  #endif
409  #if ALG_ECSCHNORR
410          case ALG_ECSCHNORR_VALUE:
411              retVal = BnSignEcSchnorr(bnR, bnS, E, bnD, digest,
412                                       signature->signature.ecschnorr.hash,
413                                       rand);
414              break;
415  #endif
416  #if ALG_SM2
417          case ALG_SM2_VALUE:
418              retVal = BnSignEcSm2(bnR, bnS, E, bnD, digest, rand);
419              break;
420  #endif
421          default:
422              break;
423      }
424      // If signature generation worked, convert the results.
425      if(retVal == TPM_RC_SUCCESS)
426      {
427          NUMBYTES     orderBytes =
428              (NUMBYTES)BITS_TO_BYTES(BnSizeInBits(CurveGetOrder(C)));
429          if(bnR != NULL)
430              BnTo2B(bnR, &signature->signature.ecdaa.signatureR.b, orderBytes);
431          if(bnS != NULL)
432              BnTo2B(bnS, &signature->signature.ecdaa.signatureS.b, orderBytes);
433      }
434  Exit:
435      CURVE_FREE(E);
436      return retVal;
437  }
438  #if ALG_ECDSA
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

10.2.12.3.8 BnValidateSignatureEcdsa()

This function validates an ECDSA signature. *rIn* and *sIn* should have been checked to make sure that they are in the range 0 < *v* < *n*

Error Returns	Meaning
TPM_RC_SIGNATURE	signature not valid

```
439  TPM_RC
440  BnValidateSignatureEcdsa(
441      bigNum                   bnR,          // IN: 'r' component of the signature
442      bigNum                   bnS,          // IN: 's' component of the signature
443      bigCurve                 E,            // IN: the curve used in the signature
444                                             //     process
445      bn_point_t              *ecQ,          // IN: the public point of the key
446      const TPM2B_DIGEST      *digest        // IN: the digest that was signed
447      )
448  {
449      // Make sure that the allocation for the digest is big enough for a maximum
450      // digest
451      BN_VAR(bnE, MAX(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE) * 8);
452      POINT(ecR);
453      ECC_NUM(bnU1);
454      ECC_NUM(bnU2);
455      ECC_NUM(bnW);
456      bigConst                 order = CurveGetOrder(AccessCurveData(E));
457      TPM_RC                   retVal = TPM_RC_SIGNATURE;
458  //
459      // Get adjusted digest
460      EcdsaDigest(bnE, digest, order);
461      // 1. If r and s are not both integers in the interval [1, n - 1], output
462      //    INVALID.
463      //  bnR  and bnS were validated by the caller
464      // 2. Use the selected hash function to compute H0 = Hash(M0).
465      // This is an input parameter
466      // 3. Convert the bit string H0 to an integer e as described in Appendix B.2.
467      // Done at entry
468      // 4. Compute w = (s')^-1 mod n, using the routine in Appendix B.1.
469      if(!BnModInverse(bnW, bnS, order))
470          goto Exit;
471      // 5. Compute u1 = (e' *   w) mod n, and compute u2 = (r' *  w) mod n.
472      BnModMult(bnU1, bnE, bnW, order);
473      BnModMult(bnU2, bnR, bnW, order);
474      // 6. Compute the elliptic curve point R = (xR, yR) = u1G+u2Q, using EC
475      //    scalar multiplication and EC addition (see [Routines]). If R is equal to
476      //    the point at infinity O, output INVALID.
477      if(BnPointMult(ecR, CurveGetG(AccessCurveData(E)), bnU1, ecQ, bnU2, E)
478         != TPM_RC_SUCCESS)
479          goto Exit;
480      // 7. Compute v = Rx mod n.
481      BnMod(ecR->x, order);
482      // 8. Compare v and r0. If v = r0, output VALID; otherwise, output INVALID
483      if(BnUnsignedCmp(ecR->x, bnR) != 0)
484          goto Exit;
485
486      retVal = TPM_RC_SUCCESS;
487  Exit:
488      return retVal;
489  }
490  #endif      // ALG_ECDSA
491  #if ALG_SM2
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.2.12.3.9 BnValidateSignatureEcSm2()

This function is used to validate an SM2 signature.

Error Returns	Meaning
TPM_RC_SIGNATURE	signature not valid

```
492 static TPM_RC
493 BnValidateSignatureEcSm2(
494     bigNum                  bnR,        // IN: 'r' component of the signature
495     bigNum                  bnS,        // IN: 's' component of the signature
496     bigCurve                E,          // IN: the curve used in the signature
497                                         //     process
498     bigPoint                ecQ,        // IN: the public point of the key
499     const TPM2B_DIGEST     *digest      // IN: the digest that was signed
500     )
501 {
502     POINT(P);
503     ECC_NUM(bnRp);
504     ECC_NUM(bnT);
505     BN_MAX_INITIALIZED(bnE, digest);
506     BOOL                    OK;
507     bigConst                order = CurveGetOrder(AccessCurveData(E));
508
509 #ifdef _SM2_SIGN_DEBUG
510     // Make sure that the input signature is the test signature
511     pAssert(BnHexEqual(bnR,
512                        "40F1EC59F793D9F49E09DCEF49130D41"
513                        "94F79FB1EED2CAA55BACDB49C4E755D1"));
514     pAssert(BnHexEqual(bnS,
515                        "6FC6DAC32C5D5CF10C77DFB20F7C2EB6"
516                        "67A457872FB09EC56327A67EC7DEEBE7"));
517 #endif
518     // b)   compute t  := (r + s) mod n
519     BnAdd(bnT, bnR, bnS);
520     BnMod(bnT, order);
521 #ifdef _SM2_SIGN_DEBUG
522     pAssert(BnHexEqual(bnT,
523                        "2B75F07ED7ECE7CCC1C8986B991F441A"
524                        "D324D6D619FE06DD63ED32E0C997C801"));
525 #endif
526     // c)   verify that t > 0
527     OK = !BnEqualZero(bnT);
528     if(!OK)
529         // set T to a value that should allow rest of the computations to run
530         // without trouble
531         BnCopy(bnT, bnS);
532     // d)   compute (x, y)  := [s]G + [t]Q
533     OK = BnEccModMult2(P, NULL, bnS, ecQ, bnT, E);
534 #ifdef  _SM2_SIGN_DEBUG
535     pAssert(OK && BnHexEqual(P->x,
536                              "110FCDA57615705D5E7B9324AC4B856D"
537                              "23E6D9188B2AE47759514657CE25D112"));
538 #endif
539     // e)   compute r' := (e + x) mod n (the x coordinate is in bnT)
540     OK = OK && BnAdd(bnRp, bnE, P->x);
541     OK = OK && BnMod(bnRp, order);
542
543     // f)   verify that r' = r
544     OK = OK && (BnUnsignedCmp(bnR, bnRp) == 0);
545
546     if(!OK)
547         return TPM_RC_SIGNATURE;
548     else
```

```
549             return TPM_RC_SUCCESS;
550     }
551     #endif  // ALG_SM2
552     #if ALG_ECSCHNORR
```

10.2.12.3.10 BnValidateSignatureEcSchnorr()

This function is used to validate an EC Schnorr signature.

Error Returns	Meaning
TPM_RC_SIGNATURE	signature not valid

```
553     static TPM_RC
554     BnValidateSignatureEcSchnorr(
555         bigNum                bnR,        // IN: 'r' component of the signature
556         bigNum                bnS,        // IN: 's' component of the signature
557         TPM_ALG_ID            hashAlg,    // IN: hash algorithm of the signature
558         bigCurve              E,          // IN: the curve used in the signature
559                                           //     process
560         bigPoint              ecQ,        // IN: the public point of the key
561         const TPM2B_DIGEST   *digest      // IN: the digest that was signed
562         )
563     {
564         BN_MAX(bnRn);
565         POINT(ecE);
566         BN_MAX(bnEx);
567         const ECC_CURVE_DATA    *C = AccessCurveData(E);
568         bigConst                 order = CurveGetOrder(C);
569         UINT16                   digestSize = CryptHashGetDigestSize(hashAlg);
570         HASH_STATE               hashState;
571         TPM2B_TYPE(BUFFER, MAX(MAX_ECC_PARAMETER_BYTES, MAX_DIGEST_SIZE));
572         TPM2B_BUFFER             Ex2 = {{sizeof(Ex2.t.buffer),{ 0 }}};
573         BOOL                     OK;
574     //
575         // E = [s]G - [r]Q
576         BnMod(bnR, order);
577         // Make -r = n - r
578         BnSub(bnRn, order, bnR);
579         // E = [s]G + [-r]Q
580         OK = BnPointMult(ecE, CurveGetG(C), bnS, ecQ, bnRn, E) == TPM_RC_SUCCESS;
581     //    // reduce the x portion of E mod q
582     //      OK = OK && BnMod(ecE->x, order);
583         // Convert to byte string
584         OK = OK && BnTo2B(ecE->x, &Ex2.b,
585                           (NUMBYTES)(BITS_TO_BYTES(BnSizeInBits(order))));
586         if(OK)
587         {
588     // Ex = h(pE.x || digest)
589             CryptHashStart(&hashState, hashAlg);
590             CryptDigestUpdate(&hashState, Ex2.t.size, Ex2.t.buffer);
591             CryptDigestUpdate(&hashState, digest->t.size, digest->t.buffer);
592             Ex2.t.size = CryptHashEnd(&hashState, digestSize, Ex2.t.buffer);
593             SchnorrReduce(&Ex2.b, order);
594             BnFrom2B(bnEx, &Ex2.b);
595             // see if Ex matches R
596             OK = BnUnsignedCmp(bnEx, bnR) == 0;
597         }
598         return (OK) ? TPM_RC_SUCCESS : TPM_RC_SIGNATURE;
599     }
600     #endif  // ALG_ECSCHNORR
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.2.12.3.11 CryptEccValidateSignature()

This function validates an EcDsa() or EcSchnorr() signature. The point *Qin* needs to have been validated to be on the curve of *curveId*.

Error Returns	Meaning
TPM_RC_SIGNATURE	not a valid signature

```
601 LIB_EXPORT TPM_RC
602 CryptEccValidateSignature(
603     TPMT_SIGNATURE          *signature,     // IN: signature to be verified
604     OBJECT                  *signKey,       // IN: ECC key signed the hash
605     const TPM2B_DIGEST      *digest         // IN: digest that was signed
606     )
607 {
608     CURVE_INITIALIZED(E, signKey->publicArea.parameters.eccDetail.curveID);
609     ECC_NUM(bnR);
610     ECC_NUM(bnS);
611     POINT_INITIALIZED(ecQ, &signKey->publicArea.unique.ecc);
612     bigConst                 order;
613     TPM_RC                   retVal;
614
615     if(E == NULL)
616         ERROR_RETURN(TPM_RC_VALUE);
617
618     order = CurveGetOrder(AccessCurveData(E));
619
620 //    // Make sure that the scheme is valid
621     switch(signature->sigAlg)
622     {
623         case ALG_ECDSA_VALUE:
624 #if ALG_ECSCHNORR
625         case ALG_ECSCHNORR_VALUE:
626 #endif
627 #if ALG_SM2
628         case ALG_SM2_VALUE:
629 #endif
630             break;
631         default:
632             ERROR_RETURN(TPM_RC_SCHEME);
633             break;
634     }
635     // Can convert r and s after determining that the scheme is an ECC scheme. If
636     // this conversion doesn't work, it means that the unmarshaling code for
637     // an ECC signature is broken.
638     BnFrom2B(bnR, &signature->signature.ecdsa.signatureR.b);
639     BnFrom2B(bnS, &signature->signature.ecdsa.signatureS.b);
640
641     // r and s have to be greater than 0 but less than the curve order
642     if(BnEqualZero(bnR) || BnEqualZero(bnS))
643         ERROR_RETURN(TPM_RC_SIGNATURE);
644     if((BnUnsignedCmp(bnS, order) >= 0)
645        || (BnUnsignedCmp(bnR, order) >= 0))
646         ERROR_RETURN(TPM_RC_SIGNATURE);
647
648     switch(signature->sigAlg)
649     {
650         case ALG_ECDSA_VALUE:
651             retVal = BnValidateSignatureEcdsa(bnR, bnS, E, ecQ, digest);
652             break;
653
654 #if ALG_ECSCHNORR
655         case ALG_ECSCHNORR_VALUE:
656             retVal = BnValidateSignatureEcSchnorr(bnR, bnS,
```

 Copyright © TCG 2006-2020

```
657                                                       signature->signature.any.hashAlg,
658                                                       E, ecQ, digest);
659             break;
660 #endif
661 #if ALG_SM2
662         case ALG_SM2_VALUE:
663             retVal = BnValidateSignatureEcSm2(bnR, bnS, E, ecQ, digest);
664             break;
665 #endif
666         default:
667             FAIL(FATAL_ERROR_INTERNAL);
668     }
669 Exit:
670     CURVE_FREE(E);
671     return retVal;
672 }
```

10.2.12.3.12 CryptEccCommitCompute()

This function performs the point multiply operations required by TPM2_Commit().

If *B* or *M* is provided, they must be on the curve defined by *curveId*. This routine does not check that they are on the curve and results are unpredictable if they are not.

It is a fatal error if *r* is NULL. If *B* is not NULL, then it is a fatal error if *d* is NULL or if *K* and *L* are both NULL. If *M* is not NULL, then it is a fatal error if *E* is NULL.

Error Returns	Meaning
TPM_RC_NO_RESULT	if *K*, *L* or *E* was computed to be the point at infinity
TPM_RC_CANCELED	a cancel indication was asserted during this function

```
673 LIB_EXPORT TPM_RC
674 CryptEccCommitCompute(
675     TPMS_ECC_POINT          *K,             // OUT: [d]B or [r]Q
676     TPMS_ECC_POINT          *L,             // OUT: [r]B
677     TPMS_ECC_POINT          *E,             // OUT: [r]M
678     TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
679     TPMS_ECC_POINT          *M,             // IN: M (optional)
680     TPMS_ECC_POINT          *B,             // IN: B (optional)
681     TPM2B_ECC_PARAMETER     *d,             // IN: d (optional)
682     TPM2B_ECC_PARAMETER     *r              // IN: the computed r value (required)
683     )
684 {
685     CURVE_INITIALIZED(curve, curveId);  // Normally initialize E as the curve, but
686                                         // E means something else in this function
687     ECC_INITIALIZED(bnR, r);
688     TPM_RC                 retVal = TPM_RC_SUCCESS;
689 //
690     // Validate that the required parameters are provided.
691     // Note: E has to be provided if computing E := [r]Q or E := [r]M. Will do
692     // E := [r]Q if both M and B are NULL.
693     pAssert(r != NULL && E != NULL);
694
695     // Initialize the output points in case they are not computed
696     ClearPoint2B(K);
697     ClearPoint2B(L);
698     ClearPoint2B(E);
699
700     // Sizes of the r parameter may not be zero
701     pAssert(r->t.size > 0);
702
703     // If B is provided, compute K=[d]B and L=[r]B
704     if(B != NULL)
```

```
705         {
706             ECC_INITIALIZED(bnD, d);
707             POINT_INITIALIZED(pB, B);
708             POINT(pK);
709             POINT(pL);
710     //
711             pAssert(d != NULL && K != NULL && L != NULL);
712
713             if(!BnIsOnCurve(pB, AccessCurveData(curve)))
714                 ERROR_RETURN(TPM_RC_VALUE);
715              // do the math for K = [d]B
716             if((retVal = BnPointMult(pK, pB, bnD, NULL, NULL, curve)) != TPM_RC_SUCCESS)
717                 goto Exit;
718             // Convert BN K to TPM2B K
719             BnPointTo2B(K, pK, curve);
720             //  compute L= [r]B after checking for cancel
721             if(_plat__IsCanceled())
722                 ERROR_RETURN(TPM_RC_CANCELED);
723             // compute L = [r]B
724             if(!BnIsValidPrivateEcc(bnR, curve))
725                 ERROR_RETURN(TPM_RC_VALUE);
726             if((retVal = BnPointMult(pL, pB, bnR, NULL, NULL, curve)) != TPM_RC_SUCCESS)
727                 goto Exit;
728             // Convert BN L to TPM2B L
729             BnPointTo2B(L, pL, curve);
730         }
731         if((M != NULL) || (B == NULL))
732         {
733             POINT_INITIALIZED(pM, M);
734             POINT(pE);
735     //
736             // Make sure that a place was provided for the result
737             pAssert(E != NULL);
738
739             // if this is the third point multiply, check for cancel first
740             if((B != NULL) && _plat__IsCanceled())
741                 ERROR_RETURN(TPM_RC_CANCELED);
742
743             // If M provided, then pM will not be NULL and will compute E = [r]M.
744             // However, if M was not provided, then pM will be NULL and E = [r]G
745             // will be computed
746             if((retVal = BnPointMult(pE, pM, bnR, NULL, NULL, curve)) != TPM_RC_SUCCESS)
747                 goto Exit;
748             // Convert E to 2B format
749             BnPointTo2B(E, pE, curve);
750         }
751     Exit:
752         CURVE_FREE(curve);
753         return retVal;
754     }
755     #endif  // ALG_ECC
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

10.2.13 CryptHash.c

10.2.13.1 Description

This file contains implementation of cryptographic functions for hashing.

10.2.13.2 Includes, Defines, and Types

```
1   #define _CRYPT_HASH_C_
2   #include "Tpm.h"
3   #include "CryptHash_fp.h"
4   #include "CryptHash.h"
5   #include "OIDs.h"
6   #define HASH_TABLE_SIZE     (HASH_COUNT + 1)
7   #if     ALG_SHA1
8   HASH_DEF_TEMPLATE(SHA1, Sha1);
9   #endif
10  #if     ALG_SHA256
11  HASH_DEF_TEMPLATE(SHA256, Sha256);
12  #endif
13  #if     ALG_SHA384
14  HASH_DEF_TEMPLATE(SHA384, Sha384);
15  #endif
16  #if     ALG_SHA512
17  HASH_DEF_TEMPLATE(SHA512, Sha512);
18  #endif
19  #if ALG_SM3_256
20  HASH_DEF_TEMPLATE(SM3_256, Sm3_256);
21  #endif
22  HASH_DEF NULL_Def = {{0}};
23
24  PHASH_DEF       HashDefArray[] = {
25  #if ALG_SHA1
26      &Sha1_Def,
27  #endif
28  #if ALG_SHA256
29      &Sha256_Def,
30  #endif
31  #if ALG_SHA384
32      &Sha384_Def,
33  #endif
34  #if ALG_SHA512
35      &Sha512_Def,
36  #endif
37  #if ALG_SM3_256
38      &Sm3_256_Def,
39  #endif
40      &NULL_Def
41  };
42
43  //** Obligatory Initialization Functions
44
45  //*** CryptHashInit()
46  // This function is called by _TPM_Init do perform the initialization operations for
47  // the library.
48  BOOL
49  CryptHashInit(
50      void
51      )
52  {
53      LibHashInit();
54      return TRUE;
```

```
55 }
```

10.2.13.2.1 CryptHashStartup()

This function is called by TPM2_Startup(). It checks that the size of the HashDefArray() is consistent with the HASH_COUNT.

```
56 BOOL
57 CryptHashStartup(
58     void
59     )
60 {
61     int         i = sizeof(HashDefArray) / sizeof(PHASH_DEF) - 1;
62     return (i == HASH_COUNT);
63 }
```

10.2.13.3 Hash Information Access Functions

10.2.13.3.1 Introduction

These functions provide access to the hash algorithm description information.

10.2.13.3.2 CryptGetHashDef()

This function accesses the hash descriptor associated with a hash a algorithm. The function returns a pointer to a *null* descriptor if *hashAlg* is TPM_ALG_NULL or not a defined algorithm.

```
64 PHASH_DEF
65 CryptGetHashDef(
66     TPM_ALG_ID      hashAlg
67     )
68 {
69     size_t          i;
70 #define HASHES  (sizeof(HashDefArray) / sizeof(PHASH_DEF))
71     for(i = 0; i < HASHES; i++)
72     {
73         PHASH_DEF p = HashDefArray[i];
74         if(p->hashAlg == hashAlg)
75             return p;
76     }
77     return &NULL_Def;
78 }
```

10.2.13.3.3 CryptHashIsValidAlg()

This function tests to see if an algorithm ID is a valid hash algorithm. If flag is true, then TPM_ALG_NULL is a valid hash.

Return Value	Meaning
TRUE(1)	*hashAlg* is a valid, implemented hash on this TPM
FALSE(0)	*hashAlg* is not valid for this TPM

```
79 BOOL
80 CryptHashIsValidAlg(
81     TPM_ALG_ID      hashAlg,           // IN: the algorithm to check
82     BOOL            flag               // IN: TRUE if TPM_ALG_NULL is to be treated
83                                        //     as a valid hash
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
84      )
85  {
86      if(hashAlg == TPM_ALG_NULL)
87          return flag;
88      return CryptGetHashDef(hashAlg) != &NULL_Def;
89  }
```

10.2.13.3.4 CryptHashGetAlgByIndex()

This function is used to iterate through the hashes. TPM_ALG_NULL is returned for all indexes that are not valid hashes. If the TPM implements 3 hashes, then an *index* value of 0 will return the first implemented hash and an *index* of 2 will return the last. All other index values will return TPM_ALG_NULL.

Return Value	Meaning
TPM_ALG_xxx	a hash algorithm
TPM_ALG_NULL	this can be used as a stop value

```
90  LIB_EXPORT TPM_ALG_ID
91  CryptHashGetAlgByIndex(
92      UINT32          index           // IN: the index
93      )
94  {
95      TPM_ALG_ID      hashAlg;
96      if(index >= HASH_COUNT)
97          hashAlg = TPM_ALG_NULL;
98      else
99          hashAlg = HashDefArray[index]->hashAlg;
100     return hashAlg;
101 }
```

10.2.13.3.5 CryptHashGetDigestSize()

Returns the size of the digest produced by the hash. If *hashAlg* is not a hash algorithm, the TPM will FAIL.

Return Value	Meaning
0	TPM_ALG_NULL
0	the digest size

```
102 LIB_EXPORT UINT16
103 CryptHashGetDigestSize(
104     TPM_ALG_ID      hashAlg         // IN: hash algorithm to look up
105     )
106 {
107     return CryptGetHashDef(hashAlg)->digestSize;
108 }
```

10.2.13.3.6 CryptHashGetBlockSize()

Returns the size of the block used by the hash. If *hashAlg* is not a hash algorithm, the TPM will FAIL.

 Copyright © TCG 2006-2020

Return Value	Meaning
0	TPM_ALG_NULL
0	the digest size

```
109  LIB_EXPORT UINT16
110  CryptHashGetBlockSize(
111      TPM_ALG_ID       hashAlg         // IN: hash algorithm to look up
112      )
113  {
114      return CryptGetHashDef(hashAlg)->blockSize;
115  }
```

10.2.13.3.7 CryptHashGetOid()

This function returns a pointer to DER=encoded OID for a hash algorithm. All OIDs are full OID values including the Tag (0x06) and length byte.

```
116  LIB_EXPORT const BYTE *
117  CryptHashGetOid(
118      TPM_ALG_ID      hashAlg
119  )
120  {
121      return CryptGetHashDef(hashAlg)->OID;
122  }
```

10.2.13.3.8 CryptHashGetContextAlg()

This function returns the hash algorithm associated with a hash context.

```
123  TPM_ALG_ID
124  CryptHashGetContextAlg(
125      PHASH_STATE      state           // IN: the context to check
126      )
127  {
128      return state->hashAlg;
129  }
```

10.2.13.4 State Import and Export

10.2.13.4.1 CryptHashCopyState

This function is used to clone a HASH_STATE.

```
130  LIB_EXPORT void
131  CryptHashCopyState(
132      HASH_STATE          *out,           // OUT: destination of the state
133      const HASH_STATE    *in             // IN: source of the state
134      )
135  {
136      pAssert(out->type == in->type);
137      out->hashAlg = in->hashAlg;
138      out->def = in->def;
139      if(in->hashAlg != TPM_ALG_NULL)
140      {
141          HASH_STATE_COPY(out, in);
142      }
143      if(in->type == HASH_STATE_HMAC)
144      {
```

 Copyright © TCG 2006-2020

```
145             const HMAC_STATE     *hIn = (HMAC_STATE *)in;
146             HMAC_STATE           *hOut = (HMAC_STATE *)out;
147             hOut->hmacKey = hIn->hmacKey;
148         }
149         return;
150     }
```

10.2.13.4.2 CryptHashExportState()

This function is used to export a hash or HMAC hash state. This function would be called when preparing to context save a sequence object.

```
151     void
152     CryptHashExportState(
153         PCHASH_STATE         internalFmt,    // IN: the hash state formatted for use by
154                                              //     library
155         PEXPORT_HASH_STATE   externalFmt     // OUT: the exported hash state
156         )
157     {
158         BYTE                     *outBuf = (BYTE *)externalFmt;
159     //
160         cAssert(sizeof(HASH_STATE) <= sizeof(EXPORT_HASH_STATE));
161         // the following #define is used to move data from an aligned internal data
162         // structure to a byte buffer (external format data.
163     #define CopyToOffset(value)                                                  \
164             memcpy(&outBuf[offsetof(HASH_STATE,value)], &internalFmt->value,        \
165                     sizeof(internalFmt->value))
166         // Copy the hashAlg
167         CopyToOffset(hashAlg);
168         CopyToOffset(type);
169     #ifdef HASH_STATE_SMAC
170         if(internalFmt->type == HASH_STATE_SMAC)
171         {
172             memcpy(outBuf, internalFmt, sizeof(HASH_STATE));
173             return;
174
175         }
176     #endif
177         if(internalFmt->type == HASH_STATE_HMAC)
178         {
179             HMAC_STATE               *from = (HMAC_STATE *)internalFmt;
180             memcpy(&outBuf[offsetof(HMAC_STATE, hmacKey)], &from->hmacKey,
181                    sizeof(from->hmacKey));
182         }
183         if(internalFmt->hashAlg != TPM_ALG_NULL)
184              HASH_STATE_EXPORT(externalFmt, internalFmt);
185     }
```

10.2.13.4.3 CryptHashImportState()

This function is used to import the hash state. This function would be called to import a hash state when the context of a sequence object was being loaded.

```
186     void
187     CryptHashImportState(
188         PHASH_STATE          internalFmt,    // OUT: the hash state formatted for use by
189                                              //     the library
190         PCEXPORT_HASH_STATE  externalFmt     // IN: the exported hash state
191         )
192     {
193         BYTE                     *inBuf = (BYTE *)externalFmt;
194     //
```

```
195 #define CopyFromOffset(value)                                                    \
196     memcpy(&internalFmt->value, &inBuf[offsetof(HASH_STATE,value)],              \
197             sizeof(internalFmt->value))
198
199     // Copy the hashAlg of the byte-aligned input structure to the structure-aligned
200     // internal structure.
201     CopyFromOffset(hashAlg);
202     CopyFromOffset(type);
203     if(internalFmt->hashAlg != TPM_ALG_NULL)
204     {
205 #ifdef HASH_STATE_SMAC
206         if(internalFmt->type == HASH_STATE_SMAC)
207         {
208             memcpy(internalFmt, inBuf, sizeof(HASH_STATE));
209             return;
210         }
211 #endif
212         internalFmt->def = CryptGetHashDef(internalFmt->hashAlg);
213         HASH_STATE_IMPORT(internalFmt, inBuf);
214         if(internalFmt->type == HASH_STATE_HMAC)
215         {
216             HMAC_STATE              *to = (HMAC_STATE *)internalFmt;
217             memcpy(&to->hmacKey, &inBuf[offsetof(HMAC_STATE, hmacKey)],
218                     sizeof(to->hmacKey));
219         }
220     }
221 }
```

10.2.13.5 State Modification Functions

10.2.13.5.1 HashEnd()

Local function to complete a hash that uses the *hashDef* instead of an algorithm ID. This function is used to complete the hash and only return a partial digest. The return value is the size of the data copied.

```
222 static UINT16
223 HashEnd(
224     PHASH_STATE      hashState,     // IN: the hash state
225     UINT32           dOutSize,      // IN: the size of receive buffer
226     PBYTE            dOut           // OUT: the receive buffer
227     )
228 {
229     BYTE                temp[MAX_DIGEST_SIZE];
230     if((hashState->hashAlg == TPM_ALG_NULL)
231        || (hashState->type != HASH_STATE_HASH))
232         dOutSize = 0;
233     if(dOutSize > 0)
234     {
235         hashState->def = CryptGetHashDef(hashState->hashAlg);
236         // Set the final size
237         dOutSize = MIN(dOutSize, hashState->def->digestSize);
238         // Complete into the temp buffer and then copy
239         HASH_END(hashState, temp);
240         // Don't want any other functions calling the HASH_END method
241         // directly.
242 #undef HASH_END
243         memcpy(dOut, &temp, dOutSize);
244     }
245     hashState->type = HASH_STATE_EMPTY;
246     return (UINT16)dOutSize;
247 }
```

10.2.13.5.2 CryptHashStart()

Functions starts a hash stack Start a hash stack and returns the digest size. As a side effect, the value of *stateSize* in *hashState* is updated to indicate the number of bytes of state that were saved. This function calls GetHashServer() and that function will put the TPM into failure mode if the hash algorithm is not supported.

This function does not use the sequence parameter. If it is necessary to import or export context, this will start the sequence in a local state and export the state to the input buffer. Will need to add a flag to the state structure to indicate that it needs to be imported before it can be used. (BLEH).

Return Value	Meaning
0	hash is TPM_ALG_NULL
>0	digest size

```
248 LIB_EXPORT UINT16
249 CryptHashStart(
250     PHASH_STATE      hashState,      // OUT: the running hash state
251     TPM_ALG_ID       hashAlg         // IN: hash algorithm
252     )
253 {
254     UINT16               retVal;
255
256     TEST(hashAlg);
257
258     hashState->hashAlg = hashAlg;
259     if(hashAlg == TPM_ALG_NULL)
260     {
261         retVal = 0;
262     }
263     else
264     {
265         hashState->def = CryptGetHashDef(hashAlg);
266         HASH_START(hashState);
267         retVal = hashState->def->digestSize;
268     }
269 #undef HASH_START
270     hashState->type = HASH_STATE_HASH;
271     return retVal;
272 }
```

10.2.13.5.3 CryptDigestUpdate()

Add data to a hash or HMAC, SMAC stack.

```
273 void
274 CryptDigestUpdate(
275     PHASH_STATE      hashState,      // IN: the hash context information
276     UINT32           dataSize,       // IN: the size of data to be added
277     const BYTE      *data            // IN: data to be hashed
278     )
279 {
280     if(hashState->hashAlg != TPM_ALG_NULL)
281     {
282         if((hashState->type == HASH_STATE_HASH)
283            || (hashState->type == HASH_STATE_HMAC))
284             HASH_DATA(hashState, dataSize, (BYTE *)data);
285 #if SMAC_IMPLEMENTED
286         else if(hashState->type == HASH_STATE_SMAC)
287             (hashState->state.smac.smacMethods.data)(&hashState->state.smac.state,
288                                                      dataSize, data);
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
289 #endif // SMAC_IMPLEMENTED
290         else
291             FAIL(FATAL_ERROR_INTERNAL);
292     }
293     return;
294 }
```

10.2.13.5.4 CryptHashEnd()

Complete a hash or HMAC computation. This function will place the smaller of *digestSize* or the size of the digest in *dOut*. The number of bytes in the placed in the buffer is returned. If there is a failure, the returned value is <= 0.

Return Value	Meaning
0	no data returned
0	the number of bytes in the digest or *dOutSize*, whichever is smaller

```
295 LIB_EXPORT UINT16
296 CryptHashEnd(
297     PHASH_STATE      hashState,     // IN: the state of hash stack
298     UINT32           dOutSize,      // IN: size of digest buffer
299     BYTE            *dOut           // OUT: hash digest
300     )
301 {
302     pAssert(hashState->type == HASH_STATE_HASH);
303     return HashEnd(hashState, dOutSize, dOut);
304 }
```

10.2.13.5.5 CryptHashBlock()

Start a hash, hash a single block, update *digest* and return the size of the results.

The *digestSize* parameter can be smaller than the digest. If so, only the more significant bytes are returned.

Return Value	Meaning
0	number of bytes placed in *dOut*

```
305 LIB_EXPORT UINT16
306 CryptHashBlock(
307     TPM_ALG_ID       hashAlg,       // IN: The hash algorithm
308     UINT32           dataSize,      // IN: size of buffer to hash
309     const BYTE      *data,          // IN: the buffer to hash
310     UINT32           dOutSize,      // IN: size of the digest buffer
311     BYTE            *dOut           // OUT: digest buffer
312     )
313 {
314     HASH_STATE          state;
315     CryptHashStart(&state, hashAlg);
316     CryptDigestUpdate(&state, dataSize, data);
317     return HashEnd(&state, dOutSize, dOut);
318 }
```

10.2.13.5.6 CryptDigestUpdate2B()

This function updates a digest (hash or HMAC) with a TPM2B.

This function can be used for both HMAC and hash functions so the *digestState* is void so that either state type can be passed.

```
319     LIB_EXPORT void
320     CryptDigestUpdate2B(
321         PHASH_STATE      state,          // IN: the digest state
322         const TPM2B     *bIn             // IN: 2B containing the data
323         )
324     {
325         // Only compute the digest if a pointer to the 2B is provided.
326         // In CryptDigestUpdate(), if size is zero or buffer is NULL, then no change
327         // to the digest occurs. This function should not provide a buffer if bIn is
328         // not provided.
329         pAssert(bIn != NULL);
330         CryptDigestUpdate(state, bIn->size, bIn->buffer);
331         return;
332     }
```

10.2.13.5.7 CryptHashEnd2B()

This function is the same as CryptCompleteHash() but the digest is placed in a TPM2B. This is the most common use and this is provided for specification clarity. *digest.size* should be set to indicate the number of bytes to place in the buffer

Return Value	Meaning
>=0	the number of bytes placed in *digest.buffer*

```
333     LIB_EXPORT UINT16
334     CryptHashEnd2B(
335         PHASH_STATE      state,          // IN: the hash state
336         P2B              digest          // IN: the size of the buffer Out: requested
337                                          //     number of bytes
338         )
339     {
340         return CryptHashEnd(state, digest->size, digest->buffer);
341     }
```

10.2.13.5.8 CryptDigestUpdateInt()

This function is used to include an integer value to a hash stack. The function marshals the integer into its canonical form before calling CryptDigestUpdate().

```
342     LIB_EXPORT void
343     CryptDigestUpdateInt(
344         void            *state,          // IN: the state of hash stack
345         UINT32           intSize,        // IN: the size of 'intValue' in bytes
346         UINT64           intValue        // IN: integer value to be hashed
347         )
348     {
349     #if LITTLE_ENDIAN_TPM
350         intValue = REVERSE_ENDIAN_64(intValue);
351     #endif
352         CryptDigestUpdate(state, intSize, &((BYTE *)&intValue)[8 - intSize]);
353     }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.2.13.6 HMAC Functions

10.2.13.6.1 CryptHmacStart()

This function is used to start an HMAC using a temp hash context. The function does the initialization of the hash with the HMAC key XOR *iPad* and updates the HMAC key XOR *oPad*.

The function returns the number of bytes in a digest produced by *hashAlg*.

Return Value	Meaning
0	number of bytes in digest produced by *hashAlg* (may be zero)

```
354  LIB_EXPORT UINT16
355  CryptHmacStart(
356      PHMAC_STATE      state,          // IN/OUT: the state buffer
357      TPM_ALG_ID       hashAlg,        // IN: the algorithm to use
358      UINT16           keySize,        // IN: the size of the HMAC key
359      const BYTE      *key             // IN: the HMAC key
360      )
361  {
362      PHASH_DEF                hashDef;
363      BYTE *                   pb;
364      UINT32                   i;
365  //
366      hashDef = CryptGetHashDef(hashAlg);
367      if(hashDef->digestSize != 0)
368      {
369      // If the HMAC key is larger than the hash block size, it has to be reduced
370      // to fit. The reduction is a digest of the hashKey.
371          if(keySize > hashDef->blockSize)
372          {
373              // if the key is too big, reduce it to a digest of itself
374              state->hmacKey.t.size = CryptHashBlock(hashAlg, keySize, key,
375                                                     hashDef->digestSize,
376                                                     state->hmacKey.t.buffer);
377          }
378          else
379          {
380              memcpy(state->hmacKey.t.buffer, key, keySize);
381              state->hmacKey.t.size = keySize;
382          }
383          // XOR the key with iPad (0x36)
384          pb = state->hmacKey.t.buffer;
385          for(i = state->hmacKey.t.size; i > 0; i--)
386              *pb++ ^= 0x36;
387
388          // if the keySize is smaller than a block, fill the rest with 0x36
389          for(i = hashDef->blockSize - state->hmacKey.t.size; i > 0; i--)
390              *pb++ = 0x36;
391
392          // Increase the oPadSize to a full block
393          state->hmacKey.t.size = hashDef->blockSize;
394
395          // Start a new hash with the HMAC key
396          // This will go in the caller's state structure and may be a sequence or not
397          CryptHashStart((PHASH_STATE)state, hashAlg);
398          CryptDigestUpdate((PHASH_STATE)state, state->hmacKey.t.size,
399                            state->hmacKey.t.buffer);
400          // XOR the key block with 0x5c ^ 0x36
401          for(pb = state->hmacKey.t.buffer, i = hashDef->blockSize; i > 0; i--)
402              *pb++ ^= (0x5c ^ 0x36);
403      }
404      // Set the hash algorithm
```

```
405     state->hashState.hashAlg = hashAlg;
406     // Set the hash state type
407     state->hashState.type = HASH_STATE_HMAC;
408
409     return hashDef->digestSize;
410 }
```

10.2.13.6.2 CryptHmacEnd()

This function is called to complete an HMAC. It will finish the current digest, and start a new digest. It will then add the *oPadKey* and the completed digest and return the results in *dOut*. It will not return more than *dOutSize* bytes.

Return Value	Meaning
0	number of bytes in *dOut* (may be zero)

```
411 LIB_EXPORT UINT16
412 CryptHmacEnd(
413     PHMAC_STATE      state,          // IN: the hash state buffer
414     UINT32           dOutSize,       // IN: size of digest buffer
415     BYTE            *dOut            // OUT: hash digest
416     )
417 {
418     BYTE                  temp[MAX_DIGEST_SIZE];
419     PHASH_STATE           hState = (PHASH_STATE)&state->hashState;
420
421 #if SMAC_IMPLEMENTED
422     if(hState->type == HASH_STATE_SMAC)
423         return (state->hashState.state.smac.smacMethods.end)
424                     (&state->hashState.state.smac.state,
425                      dOutSize,
426                      dOut);
427 #endif
428     pAssert(hState->type == HASH_STATE_HMAC);
429     hState->def = CryptGetHashDef(hState->hashAlg);
430     // Change the state type for completion processing
431     hState->type = HASH_STATE_HASH;
432     if(hState->hashAlg == TPM_ALG_NULL)
433         dOutSize = 0;
434     else
435     {
436
437     // Complete the current hash
438         HashEnd(hState, hState->def->digestSize, temp);
439         // Do another hash starting with the oPad
440         CryptHashStart(hState, hState->hashAlg);
441         CryptDigestUpdate(hState, state->hmacKey.t.size, state->hmacKey.t.buffer);
442         CryptDigestUpdate(hState, hState->def->digestSize, temp);
443     }
444     return HashEnd(hState, dOutSize, dOut);
445 }
```

10.2.13.6.3 CryptHmacStart2B()

This function starts an HMAC and returns the size of the digest that will be produced.

This function is provided to support the most common use of starting an HMAC with a TPM2B key.

The caller must provide a block of memory in which the hash sequence state is kept. The caller should not alter the contents of this buffer until the hash sequence is completed or abandoned.

Return Value	Meaning
0	the digest size of the algorithm
0	the *hashAlg* was TPM_ALG_NULL

```
446  LIB_EXPORT UINT16
447  CryptHmacStart2B(
448      PHMAC_STATE      hmacState,      // OUT: the state of HMAC stack. It will be used
449                                       //     in HMAC update and completion
450      TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
451      P2B              key             // IN: HMAC key
452      )
453  {
454      return CryptHmacStart(hmacState, hashAlg, key->size, key->buffer);
455  }
```

10.2.13.6.4 CryptHmacEnd2B()

This function is the same as CryptHmacEnd() but the HMAC result is returned in a TPM2B which is the most common use.

Return Value	Meaning
>=0	the number of bytes placed in *digest*

```
456  LIB_EXPORT UINT16
457  CryptHmacEnd2B(
458      PHMAC_STATE      hmacState,      // IN: the state of HMAC stack
459      P2B              digest          // OUT: HMAC
460      )
461  {
462      return CryptHmacEnd(hmacState, digest->size, digest->buffer);
463  }
```

10.2.13.7 Mask and Key Generation Functions

10.2.13.7.1 CryptMGF1()

This function performs MGF1 using the selected hash. MGF1 is T(n) = T(n-1) || H(seed || counter). This function returns the length of the mask produced which could be zero if the digest algorithm is not supported

Return Value	Meaning
0	hash algorithm was TPM_ALG_NULL
0	should be the same as *mSize*

```
464  LIB_EXPORT UINT16
465  CryptMGF1(
466      UINT32           mSize,          // IN: length of the mask to be produced
467      BYTE            *mask,           // OUT: buffer to receive the mask
468      TPM_ALG_ID       hashAlg,        // IN: hash to use
469      UINT32           seedSize,       // IN: size of the seed
470      BYTE            *seed            // IN: seed size
471      )
472  {
473      HASH_STATE           hashState;
474      PHASH_DEF            hDef = CryptGetHashDef(hashAlg);
475      UINT32               remaining;
```

```
476     UINT32               counter = 0;
477     BYTE                 swappedCounter[4];
478
479     // If there is no digest to compute return
480     if((hashAlg == TPM_ALG_NULL) || (mSize == 0))
481         return 0;
482
483     for(remaining = mSize; ; remaining -= hDef->digestSize)
484     {
485         // Because the system may be either Endian...
486         UINT32_TO_BYTE_ARRAY(counter, swappedCounter);
487
488         // Start the hash and include the seed and counter
489         CryptHashStart(&hashState, hashAlg);
490         CryptDigestUpdate(&hashState, seedSize, seed);
491         CryptDigestUpdate(&hashState, 4, swappedCounter);
492
493         // Handling the completion depends on how much space remains in the mask
494         // buffer. If it can hold the entire digest, put it there. If not
495         // put the digest in a temp buffer and only copy the amount that
496         // will fit into the mask buffer.
497         HashEnd(&hashState, remaining, mask);
498         if(remaining <= hDef->digestSize)
499             break;
500         mask = &mask[hDef->digestSize];
501         counter++;
502     }
503     return (UINT16)mSize;
504 }
```

10.2.13.7.2 CryptKDFa()

This function performs the key generation according to Part 1 of the TPM specification.

This function returns the number of bytes generated which may be zero.

The *key* and *keyStream* pointers are not allowed to be NULL. The other pointer values may be NULL. The value of *sizeInBits* must be no larger than (2^18)-1 = 256K bits (32385 bytes).

The *once* parameter is set to allow incremental generation of a large value. If this flag is TRUE, *sizeInBits* will be used in the HMAC computation but only one iteration of the KDF is performed. This would be used for XOR obfuscation so that the mask value can be generated in digest-sized chunks rather than having to be generated all at once in an arbitrarily large buffer and then XORed into the result. If *once* is TRUE, then *sizeInBits* must be a multiple of 8.

Any error in the processing of this command is considered fatal.

Return Value	Meaning
0	hash algorithm is not supported or is TPM_ALG_NULL
0	the number of bytes in the *keyStream* buffer

```
505 LIB_EXPORT UINT16
506 CryptKDFa(
507     TPM_ALG_ID       hashAlg,        // IN: hash algorithm used in HMAC
508     const TPM2B     *key,            // IN: HMAC key
509     const TPM2B     *label,          // IN: a label for the KDF
510     const TPM2B     *contextU,       // IN: context U
511     const TPM2B     *contextV,       // IN: context V
512     UINT32           sizeInBits,     // IN: size of generated key in bits
513     BYTE            *keyStream,      // OUT: key buffer
514     UINT32          *counterInOut,   // IN/OUT: caller may provide the iteration
515                                      //     counter for incremental operations to
```

```
516                                     //     avoid large intermediate buffers.
517     UINT16          blocks          // IN: If non-zero, this is the maximum number
518                                     //     of blocks to be returned, regardless
519                                     //     of sizeInBits
520     )
521 {
522     UINT32                   counter = 0;        // counter value
523     INT16                    bytes;              // number of bytes to produce
524     UINT16                   generated;          // number of bytes generated
525     BYTE                    *stream = keyStream;
526     HMAC_STATE               hState;
527     UINT16                   digestSize = CryptHashGetDigestSize(hashAlg);
528
529     pAssert(key != NULL && keyStream != NULL);
530
531     TEST(TPM_ALG_KDF1_SP800_108);
532
533     if(digestSize == 0)
534         return 0;
535
536     if(counterInOut != NULL)
537         counter = *counterInOut;
538
539     // If the size of the request is larger than the numbers will handle,
540     // it is a fatal error.
541     pAssert(((sizeInBits + 7) / 8) <= INT16_MAX);
542
543     // The number of bytes to be generated is the smaller of the sizeInBits bytes or
544     // the number of requested blocks. The number of blocks is the smaller of the
545     // number requested or the number allowed by sizeInBits. A partial block is
546     // a full block.
547     bytes = (blocks > 0) ? blocks * digestSize : (UINT16)BITS_TO_BYTES(sizeInBits);
548     generated = bytes;
549
550     // Generate required bytes
551     for(; bytes > 0; bytes -= digestSize)
552     {
553         counter++;
554         // Start HMAC
555         if(CryptHmacStart(&hState, hashAlg, key->size, key->buffer) == 0)
556             return 0;
557         // Adding counter
558         CryptDigestUpdateInt(&hState.hashState, 4, counter);
559
560         // Adding label
561         if(label != NULL)
562             HASH_DATA(&hState.hashState, label->size, (BYTE *)label->buffer);
563         // Add a null. SP108 is not very clear about when the 0 is needed but to
564         // make this like the previous version that did not add an 0x00 after
565         // a null-terminated string, this version will only add a null byte
566         // if the label parameter did not end in a null byte, or if no label
567         // is present.
568         if((label == NULL)
569            || (label->size == 0)
570            || (label->buffer[label->size - 1] != 0))
571             CryptDigestUpdateInt(&hState.hashState, 1, 0);
572         // Adding contextU
573         if(contextU != NULL)
574             HASH_DATA(&hState.hashState, contextU->size, contextU->buffer);
575         // Adding contextV
576         if(contextV != NULL)
577             HASH_DATA(&hState.hashState, contextV->size, contextV->buffer);
578         // Adding size in bits
579         CryptDigestUpdateInt(&hState.hashState, 4, sizeInBits);
580
581         // Complete and put the data in the buffer
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
582             CryptHmacEnd(&hState, bytes, stream);
583             stream = &stream[digestSize];
584         }
585         // Masking in the KDF is disabled. If the calling function wants something
586         // less than even number of bytes, then the caller should do the masking
587         // because there is no universal way to do it here
588         if(counterInOut != NULL)
589             *counterInOut = counter;
590         return generated;
591     }
```

10.2.13.7.3 CryptKDFe()

This function implements KDFe() as defined in TPM specification part 1.

This function returns the number of bytes generated which may be zero.

The *Z* and *keyStream* pointers are not allowed to be NULL. The other pointer values may be NULL. The value of *sizeInBits* must be no larger than (2^18)-1 = 256K bits (32385 bytes). Any error in the processing of this command is considered fatal.

Return Value	Meaning
0	hash algorithm is not supported or is TPM_ALG_NULL
0	the number of bytes in the *keyStream* buffer

```
592     LIB_EXPORT UINT16
593     CryptKDFe(
594         TPM_ALG_ID       hashAlg,       // IN: hash algorithm used in HMAC
595         TPM2B           *Z,             // IN: Z
596         const TPM2B     *label,         // IN: a label value for the KDF
597         TPM2B           *partyUInfo,    // IN: PartyUInfo
598         TPM2B           *partyVInfo,    // IN: PartyVInfo
599         UINT32           sizeInBits,    // IN: size of generated key in bits
600         BYTE            *keyStream      // OUT: key buffer
601         )
602     {
603         HASH_STATE       hashState;
604         PHASH_DEF        hashDef = CryptGetHashDef(hashAlg);
605
606         UINT32           counter = 0;       // counter value
607         UINT16           hLen;
608         BYTE            *stream = keyStream;
609         INT16            bytes;             // number of bytes to generate
610
611         pAssert(keyStream != NULL && Z != NULL && ((sizeInBits + 7) / 8) < INT16_MAX);
612     //
613         hLen = hashDef->digestSize;
614         bytes = (INT16)((sizeInBits + 7) / 8);
615         if(hashAlg == TPM_ALG_NULL || bytes == 0)
616             return 0;
617
618         // Generate required bytes
619         //The inner loop of that KDF uses:
620         //  Hash[i] := H(counter | Z | OtherInfo) (5)
621         // Where:
622         //  Hash[i]         the hash generated on the i-th iteration of the loop.
623         //  H()             an approved hash function
624         //  counter         a 32-bit counter that is initialized to 1 and incremented
625         //                  on each iteration
626         //  Z               the X coordinate of the product of a public ECC key and a
627         //                  different private ECC key.
628         //  OtherInfo       a collection of qualifying data for the KDF defined below.
629         //  In this specification, OtherInfo will be constructed by:
```

 Copyright © TCG 2006-2020

```
630         //      OtherInfo := Use | PartyUInfo  | PartyVInfo
631         for(; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
632         {
633             if(bytes < hLen)
634                 hLen = bytes;
635             counter++;
636             // Do the hash
637             CryptHashStart(&hashState, hashAlg);
638             // Add counter
639             CryptDigestUpdateInt(&hashState, 4, counter);
640
641             // Add Z
642             if(Z != NULL)
643                 CryptDigestUpdate2B(&hashState, Z);
644             // Add label
645             if(label != NULL)
646                 CryptDigestUpdate2B(&hashState, label);
647             // Add a null. SP108 is not very clear about when the 0 is needed but to
648             // make this like the previous version that did not add an 0x00 after
649             // a null-terminated string, this version will only add a null byte
650             // if the label parameter did not end in a null byte, or if no label
651             // is present.
652             if((label == NULL)
653                || (label->size == 0)
654                || (label->buffer[label->size - 1] != 0))
655                 CryptDigestUpdateInt(&hashState, 1, 0);
656             // Add PartyUInfo
657             if(partyUInfo != NULL)
658                 CryptDigestUpdate2B(&hashState, partyUInfo);
659
660             // Add PartyVInfo
661             if(partyVInfo != NULL)
662                 CryptDigestUpdate2B(&hashState, partyVInfo);
663
664             // Compute Hash. hLen was changed to be the smaller of bytes or hLen
665             // at the start of each iteration.
666             CryptHashEnd(&hashState, hLen, stream);
667         }
668
669         // Mask off bits if the required bits is not a multiple of byte size
670         if((sizeInBits % 8) != 0)
671             keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
672
673         return (UINT16)((sizeInBits + 7) / 8);
674     }
```

10.2.14 CryptPrime.c

10.2.14.1 Introduction

This file contains the code for prime validation.

```
1  #include "Tpm.h"
2  #include "CryptPrime_fp.h"
3  //#define CPRI_PRIME
4  //#include "PrimeTable.h"
5  #include "CryptPrimeSieve_fp.h"
6  extern const uint32_t      s_LastPrimeInTable;
7  extern const uint32_t      s_PrimeTableSize;
8  extern const uint32_t      s_PrimesInTable;
9  extern const unsigned char s_PrimeTable[];
10 extern bigConst            s_CompositeOfSmallPrimes;
11
12 //** Functions
13
14 //*** Root2()
15 // This finds ceil(sqrt(n)) to use as a stopping point for searching the prime
16 // table.
17 static uint32_t
18 Root2(
19     uint32_t             n
20     )
21 {
22     int32_t              last = (int32_t)(n >> 2);
23     int32_t              next = (int32_t)(n >> 1);
24     int32_t              diff;
25     int32_t              stop = 10;
26 //
27     // get a starting point
28     for(; next != 0; last >>= 1, next >>= 2);
29     last++;
30     do
31     {
32         next = (last + (n / last)) >> 1;
33         diff = next - last;
34         last = next;
35         if(stop-- == 0)
36             FAIL(FATAL_ERROR_INTERNAL);
37     } while(diff < -1 || diff > 1);
38     if((n / next) > (unsigned)next)
39         next++;
40     pAssert(next != 0);
41     pAssert(((n / next) <= (unsigned)next) && (n / (next + 1) < (unsigned)next));
42     return next;
43 }
```

10.2.14.1.1 IsPrimeInt()

This will do a test of a word of up to 32-bits in size.

```
44 BOOL
45 IsPrimeInt(
46     uint32_t            n
47     )
48 {
49     uint32_t            i;
50     uint32_t            stop;
51     if(n < 3 || ((n & 1) == 0))
```

```
52              return (n == 2);
53          if(n <= s_LastPrimeInTable)
54          {
55              n >>= 1;
56              return ((s_PrimeTable[n >> 3] >> (n & 7)) & 1);
57          }
58          // Need to search
59          stop = Root2(n) >> 1;
60          // starting at 1 is equivalent to staring at  (1 << 1) + 1 = 3
61          for(i = 1; i < stop; i++)
62          {
63              if((s_PrimeTable[i >> 3] >> (i & 7)) & 1)
64                  // see if this prime evenly divides the number
65                  if((n % ((i << 1) + 1)) == 0)
66                      return FALSE;
67          }
68          return TRUE;
69      }
```

10.2.14.1.2 BnIsProbablyPrime()

This function is used when the key sieve is not implemented. This function Will try to eliminate some of the obvious things before going on to perform MillerRabin() as a final verification of primeness.

```
70      BOOL
71      BnIsProbablyPrime(
72          bigNum          prime,           // IN:
73          RAND_STATE      *rand            // IN: the random state just
74                                           //     in case Miller-Rabin is required
75          )
76      {
77      #if RADIX_BITS > 32
78          if(BnUnsignedCmpWord(prime, UINT32_MAX) <= 0)
79      #else
80          if(BnGetSize(prime) == 1)
81      #endif
82              return IsPrimeInt((uint32_t)prime->d[0]);
83
84          if(BnIsEven(prime))
85              return FALSE;
86          if(BnUnsignedCmpWord(prime, s_LastPrimeInTable) <= 0)
87          {
88              crypt_uword_t       temp = prime->d[0] >> 1;
89              return ((s_PrimeTable[temp >> 3] >> (temp & 7)) & 1);
90          }
91          {
92              BN_VAR(n, LARGEST_NUMBER_BITS);
93              BnGcd(n, prime, s_CompositeOfSmallPrimes);
94              if(!BnEqualWord(n, 1))
95                  return FALSE;
96          }
97          return MillerRabin(prime, rand);
98      }
```

10.2.14.1.3 MillerRabinRounds()

Function returns the number of Miller-Rabin rounds necessary to give an error probability equal to the security strength of the prime. These values are from FIPS 186-3.

```
99      UINT32
100     MillerRabinRounds(
101         UINT32          bits            // IN: Number of bits in the RSA prime
```

 Copyright © TCG 2006-2020

```
102         )
103     {
104         if(bits < 511) return 8;     // don't really expect this
105         if(bits < 1536) return 5;    // for 512 and 1K primes
106         return 4;                    // for 3K public modulus and greater
107     }
```

10.2.14.1.4 MillerRabin()

This function performs a Miller-Rabin test from FIPS 186-3. It does *iterations* trials on the number. In all likelihood, if the number is not prime, the first test fails.

Return Value	Meaning
TRUE(1)	probably prime
FALSE(0)	composite

```
108     BOOL
109     MillerRabin(
110         bigNum          bnW,
111         RAND_STATE     *rand
112         )
113     {
114         BN_MAX(bnWm1);
115         BN_PRIME(bnM);
116         BN_PRIME(bnB);
117         BN_PRIME(bnZ);
118         BOOL            ret = FALSE;    // Assumed composite for easy exit
119         unsigned int    a;
120         unsigned int    j;
121         int             wLen;
122         int             i;
123         int             iterations = MillerRabinRounds(BnSizeInBits(bnW));
124     //
125        INSTRUMENT_INC(MillerRabinTrials[PrimeIndex]);
126
127         pAssert(bnW->size > 1);
128         // Let a be the largest integer such that 2^a divides w1.
129         BnSubWord(bnWm1, bnW, 1);
130         pAssert(bnWm1->size != 0);
131
132         // Since w is odd (w-1) is even so start at bit number 1 rather than 0
133         // Get the number of bits in bnWm1 so that it doesn't have to be recomputed
134         // on each iteration.
135         i = (int)(bnWm1->size * RADIX_BITS);
136       // Now find the largest power of 2 that divides w1
137         for(a = 1;
138          (a < (bnWm1->size * RADIX_BITS)) &&
139              (BnTestBit(bnWm1, a) == 0);
140              a++);
141      // 2. m = (w1) / 2^a
142             BnShiftRight(bnM, bnWm1, a);
143           // 3. wlen = len (w).
144         wLen = BnSizeInBits(bnW);
145       // 4. For i = 1 to iterations do
146         for(i = 0; i < iterations; i++)
147         {
148             // 4.1 Obtain a string b of wlen bits from an RBG.
149             // Ensure that 1 < b < w1.
150             // 4.2 If ((b <= 1) or (b >= w1)), then go to step 4.1.
151             while(BnGetRandomBits(bnB, wLen, rand) && ((BnUnsignedCmpWord(bnB, 1) <= 0)
152                 || (BnUnsignedCmp(bnB, bnWm1) >= 0)));
153             if(g_inFailureMode)
```

```
154                 return FALSE;
155
156             // 4.3 z = b^m mod w.
157             // if ModExp fails, then say this is not
158             // prime and bail out.
159              BnModExp(bnZ, bnB, bnM, bnW);
160
161              // 4.4 If ((z == 1) or (z = w == 1)), then go to step 4.7.
162              if((BnUnsignedCmpWord(bnZ, 1) == 0)
163                 || (BnUnsignedCmp(bnZ, bnWm1) == 0))
164                 goto step4point7;
165              // 4.5 For j = 1 to a  1 do.
166              for(j = 1; j < a; j++)
167              {
168                // 4.5.1 z = z^2 mod w.
169                 BnModMult(bnZ, bnZ, bnZ, bnW);
170                // 4.5.2 If (z = w1), then go to step 4.7.
171                 if(BnUnsignedCmp(bnZ, bnWm1) == 0)
172                     goto step4point7;
173                // 4.5.3 If (z = 1), then go to step 4.6.
174                 if(BnEqualWord(bnZ, 1))
175                     goto step4point6;
176              }
177              // 4.6 Return COMPOSITE.
178      step4point6:
179              INSTRUMENT_INC(failedAtIteration[i]);
180              goto end;
181            // 4.7 Continue. Comment: Increment i for the do-loop in step 4.
182      step4point7:
183              continue;
184          }
185        // 5. Return PROBABLY PRIME
186          ret = TRUE;
187      end:
188          return ret;
189      }
190      #if ALG_RSA
```

10.2.14.1.5 RsaCheckPrime()

This will check to see if a number is prime and appropriate for an RSA prime.

This has different functionality based on whether we are using key sieving or not. If not, the number checked to see if it is divisible by the public exponent, then the number is adjusted either up or down in order to make it a better candidate. It is then checked for being probably prime.

If sieving is used, the number is used to root a sieving process.

```
191      TPM_RC
192      RsaCheckPrime(
193          bigNum           prime,
194          UINT32           exponent,
195          RAND_STATE      *rand
196          )
197      {
198      #if !RSA_KEY_SIEVE
199          TPM_RC          retVal = TPM_RC_SUCCESS;
200          UINT32          modE = BnModWord(prime, exponent);
201
202          NOT_REFERENCED(rand);
203
204          if(modE == 0)
205              // evenly divisible so add two keeping the number odd
206              BnAddWord(prime, prime, 2);
```

```
207     // want 0 != (p - 1) mod e
208     // which is 1 != p mod e
209     else if(modE == 1)
210         // subtract 2 keeping number odd and insuring that
211         // 0 != (p - 1) mod e
212         BnSubWord(prime, prime, 2);
213
214     if(BnIsProbablyPrime(prime, rand) == 0)
215         ERROR_RETURN(g_inFailureMode ? TPM_RC_FAILURE : TPM_RC_VALUE);
216 Exit:
217     return retVal;
218 #else
219     return PrimeSelectWithSieve(prime, exponent, rand);
220 #endif
221 }
```

10.2.14.1.6 AdjustPrimeCandiate()

For this math, we assume that the RSA numbers are fixed-point numbers with the decimal point to the **left** of the most significant bit. This approach helps make it clear what is happening with the MSb of the values. The two RSA primes have to be large enough so that their product will be a number with the necessary number of significant bits. For example, we want to be able to multiply two 1024-bit numbers to produce a number with 2048 significant bits. If we accept any 1024-bit prime that has its MSb set, then it is possible to produce a product that does not have the MSb SET. For example, if we use tiny keys of 16 bits and have two 8-bit *primes* of 0x80, then the public key would be 0x4000 which is only 15-bits. So, what we need to do is made sure that each of the primes is large enough so that the product of the primes is twice as large as each prime. A little arithmetic will show that the only way to do this is to make sure that each of the primes is no less than root(2)/2. That's what this functions does. This function adjusts the candidate prime so that it is odd and >= root(2)/2. This allows the product of these two numbers to be .5, which, in fixed point notation means that the most significant bit is 1. For this routine, the root(2)/2 (0.7071067811865475) approximated with *0xB505* which is, in fixed point, 0.7071075439453125 or an error of 0.000108%. Just setting the upper two bits would give a value > 0.75 which is an error of > 6%. Given the amount of time all the other computations take, reducing the error is not much of a cost, but it isn't totally required either.

This function can be replaced with a function that just sets the two most significant bits of each prime candidate without introducing any computational issues.

```
222 LIB_EXPORT void
223 RsaAdjustPrimeCandidate(
224     bigNum          prime
225     )
226 {
227     UINT32          msw;
228     UINT32          adjusted;
229
230     // If the radix is 32, the compiler should turn this into a simple assignment
231     msw = prime->d[prime->size - 1] >> ((RADIX_BITS == 64) ? 32 : 0);
232     // Multiplying 0xff...f by 0x4AFB gives 0xff..f - 0xB5050...0
233     adjusted = (msw >> 16) * 0x4AFB;
234     adjusted += ((msw & 0xFFFF) * 0x4AFB) >> 16;
235     adjusted += 0xB5050000UL;
236 #if RADIX_BITS == 64
237     // Save the low-order 32 bits
238     prime->d[prime->size - 1] &= 0xFFFFFFFFUL;
239     // replace the upper 32-bits
240     prime->d[prime->size -1] |= ((crypt_uword_t)adjusted << 32);
241 #else
242     prime->d[prime->size - 1] = (crypt_uword_t)adjusted;
243 #endif
244     // make sure the number is odd
245     prime->d[0] |= 1;
```

```
246     }
```

10.2.14.1.7 BnGeneratePrimeForRSA()

Function to generate a prime of the desired size with the proper attributes for an RSA prime.

```
247     TPM_RC
248     BnGeneratePrimeForRSA(
249         bigNum          prime,          // IN/OUT: points to the BN that will get the
250                                         //  random value
251         UINT32          bits,           // IN: number of bits to get
252         UINT32          exponent,       // IN: the exponent
253         RAND_STATE      *rand           // IN: the random state
254         )
255     {
256         BOOL            found = FALSE;
257     //
258         // Make sure that the prime is large enough
259         pAssert(prime->allocated >= BITS_TO_CRYPT_WORDS(bits));
260         // Only try to handle specific sizes of keys in order to save overhead
261         pAssert((bits % 32) == 0);
262
263         prime->size = BITS_TO_CRYPT_WORDS(bits);
264
265         while(!found)
266         {
267     // The change below is to make sure that all keys that are generated from the same
268     // seed value will be the same regardless of the endianess or word size of the CPU.
269     //       DRBG_Generate(rand, (BYTE *)prime->d, (UINT16)BITS_TO_BYTES(bits));// old
270     //       if(g_inFailureMode)                                                // old
271             if(!BnGetRandomBits(prime, bits, rand))                            // new
272                 return TPM_RC_FAILURE;
273             RsaAdjustPrimeCandidate(prime);
274             found = RsaCheckPrime(prime, exponent, rand) == TPM_RC_SUCCESS;
275         }
276         return TPM_RC_SUCCESS;
277     }
278     #endif // ALG_RSA
```

10.2.15 CryptPrimeSieve.c

10.2.15.1 Includes and defines

```
1   #include "Tpm.h"
2   #if RSA_KEY_SIEVE
3   #include "CryptPrimeSieve_fp.h"
```

This determines the number of bits in the largest sieve field.

```
4   #define MAX_FIELD_SIZE  2048
5   extern const uint32_t      s_LastPrimeInTable;
6   extern const uint32_t      s_PrimeTableSize;
7   extern const uint32_t      s_PrimesInTable;
8   extern const unsigned char s_PrimeTable[];
9
10  // This table is set of prime markers. Each entry is the prime value
11  // for the ((n + 1) * 1024) prime. That is, the entry in s_PrimeMarkers[1]
12  // is the value for the 2,048th prime. This is used in the PrimeSieve
13  // to adjust the limit for the prime search. When processing smaller
14  // prime candidates, fewer primes are checked directly before going to
15  // Miller-Rabin. As the prime grows, it is worth spending more time eliminating
16  // primes as, a) the density is lower, and b) the cost of Miller-Rabin is
17  // higher.
18  const uint32_t      s_PrimeMarkersCount = 6;
19  const uint32_t      s_PrimeMarkers[] = {
20      8167, 17881, 28183, 38891, 49871, 60961 };
21  uint32_t   primeLimit;
22
23  //** Functions
24
25  //*** RsaAdjustPrimeLimit()
26  // This used during the sieve process. The iterator for getting the
27  // next prime (RsaNextPrime()) will return primes until it hits the
28  // limit (primeLimit) set up by this function. This causes the sieve
29  // process to stop when an appropriate number of primes have been
30  // sieved.
31  LIB_EXPORT void
32  RsaAdjustPrimeLimit(
33      uint32_t        requestedPrimes
34      )
35  {
36      if(requestedPrimes == 0 || requestedPrimes > s_PrimesInTable)
37          requestedPrimes = s_PrimesInTable;
38      requestedPrimes = (requestedPrimes - 1) / 1024;
39      if(requestedPrimes < s_PrimeMarkersCount)
40          primeLimit = s_PrimeMarkers[requestedPrimes];
41      else
42          primeLimit = s_LastPrimeInTable;
43      primeLimit >>= 1;
44
45  }
```

10.2.15.1.1 RsaNextPrime()

This the iterator used during the sieve process. The input is the last prime returned (or any starting point) and the output is the next higher prime. The function returns 0 when the *primeLimit* is reached.

```
46  LIB_EXPORT uint32_t
47  RsaNextPrime(
48      uint32_t    lastPrime
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
49      )
50  {
51      if(lastPrime == 0)
52          return 0;
53      lastPrime >>= 1;
54      for(lastPrime += 1; lastPrime <= primeLimit; lastPrime++)
55      {
56          if(((s_PrimeTable[lastPrime >> 3] >> (lastPrime & 0x7)) & 1) == 1)
57              return ((lastPrime << 1) + 1);
58      }
59      return 0;
60  }
```

This table contains a previously sieved table. It has the bits for 3, 5, and 7 removed. Because of the factors, it needs to be aligned to 105 and has a repeat of 105.

```
61  const BYTE   seedValues[] = {
62      0x16, 0x29, 0xcb, 0xa4, 0x65, 0xda, 0x30, 0x6c,
63      0x99, 0x96, 0x4c, 0x53, 0xa2, 0x2d, 0x52, 0x96,
64      0x49, 0xcb, 0xb4, 0x61, 0xd8, 0x32, 0x2d, 0x99,
65      0xa6, 0x44, 0x5b, 0xa4, 0x2c, 0x93, 0x96, 0x69,
66      0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
67      0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb,
68      0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x91, 0xb2, 0x4c,
69      0x5a, 0xa6, 0x0d, 0xc3, 0x96, 0x69, 0xc9, 0x34,
70      0x25, 0xda, 0x22, 0x65, 0x99, 0xb4, 0x4c, 0x1b,
71      0x86, 0x2d, 0xd3, 0x92, 0x69, 0x4a, 0xb4, 0x45,
72      0xca, 0x32, 0x69, 0x99, 0x36, 0x0c, 0x5b, 0xa6,
73      0x25, 0xd3, 0x94, 0x68, 0x8b, 0x94, 0x65, 0xd2,
74      0x32, 0x6d, 0x18, 0xb6, 0x4c, 0x4b, 0xa6, 0x29,
75      0xd1};
76
77  #define USE_NIBBLE
78
79  #ifndef USE_NIBBLE
80  static const BYTE bitsInByte[256] = {
81      0x00, 0x01, 0x01, 0x02, 0x01, 0x02, 0x02, 0x03,
82      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
83      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
84      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
85      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
86      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
87      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
88      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
89      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
90      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
91      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
92      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
93      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
94      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
95      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
96      0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
97      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
98      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
99      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
100     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
101     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
102     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
103     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
104     0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
105     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
106     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
107     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
108         0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
109         0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
110         0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
111         0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
112         0x05, 0x06, 0x06, 0x07, 0x06, 0x07, 0x07, 0x08
113     };
114     #define BitsInByte(x)   bitsInByte[(unsigned char)x]
115     #else
116     const BYTE bitsInNibble[16] = {
117         0x00, 0x01, 0x01, 0x02, 0x01, 0x02, 0x02, 0x03,
118         0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04};
119     #define BitsInByte(x)                                           \
120                 (bitsInNibble[(unsigned char)(x) & 0xf]             \
121             +   bitsInNibble[((unsigned char)(x) >> 4) & 0xf])
122     #endif
```

10.2.15.1.2 BitsInArry()

This function counts the number of bits set in an array of bytes.

```
123     static int
124     BitsInArray(
125         const unsigned char     *a,             // IN: A pointer to an array of bytes
126         unsigned int             aSize          // IN: the number of bytes to sum
127         )
128     {
129         int     j = 0;
130         for(; aSize; a++, aSize--)
131             j += BitsInByte(*a);
132         return j;
133     }
```

10.2.15.1.3 FindNthSetBit()

This function finds the nth SET bit in a bit array. The *n* parameter is between 1 and the number of bits in the array (always a multiple of 8). If called when the array does not have n bits set, it will return -1

Return Value	Meaning
<0	no bit is set or no bit with the requested number is set
>=0	the number of the bit in the array that is the nth set

```
134     LIB_EXPORT int
135     FindNthSetBit(
136         const UINT16     aSize,         // IN: the size of the array to check
137         const BYTE      *a,             // IN: the array to check
138         const UINT32     n              // IN, the number of the SET bit
139         )
140     {
141         UINT16       i;
142         int          retValue;
143         UINT32       sum = 0;
144         BYTE         sel;
145
146         //find the bit
147         for(i = 0; (i < (int)aSize) && (sum < n); i++)
148             sum += BitsInByte(a[i]);
149         i--;
150         // The chosen bit is in the byte that was just accessed
151         // Compute the offset to the start of that byte
152         retValue = i * 8 - 1;
```

```
153         sel = a[i];
154         // Subtract the bits in the last byte added.
155         sum -= BitsInByte(sel);
156         // Now process the byte, one bit at a time.
157         for(; (sel != 0) && (sum != n); retValue++, sel = sel >> 1)
158             sum += (sel & 1) != 0;
159         return (sum == n) ? retValue : -1;
160     }
161     typedef struct
162     {
163         UINT16      prime;
164         UINT16      count;
165     } SIEVE_MARKS;
166     const SIEVE_MARKS sieveMarks[5] = {
167         {31, 7}, {73, 5}, {241, 4}, {1621, 3}, {UINT16_MAX, 2}};
168
169     //*** PrimeSieve()
170     // This function does a prime sieve over the input 'field' which has as its
171     // starting address the value in bnN. Since this initializes the Sieve
172     // using a precomputed field with the bits associated with 3, 5 and 7 already
173     // turned off, the value of pnN may need to be adjusted by a few counts to allow
174     // the precomputed field to be used without modification.
175     //
176     // To get better performance, one could address the issue of developing the
177     // composite numbers. When the size of the prime gets large, the time for doing
178     // the divisions goes up, noticeably. It could be better to develop larger composite
179     // numbers even if they need to be bigNum's themselves. The object would be to
180     // reduce the number of times that the large prime is divided into a few large
181     // divides and then use smaller divides to get to the final 16 bit (or smaller)
182     // remainders.
183     LIB_EXPORT UINT32
184     PrimeSieve(
185         bigNum          bnN,        // IN/OUT: number to sieve
186         UINT32          fieldSize, // IN: size of the field area in bytes
187         BYTE           *field       // IN: field
188         )
189     {
190         UINT32          i;
191         UINT32          j;
192         UINT32          fieldBits = fieldSize * 8;
193         UINT32          r;
194         BYTE           *pField;
195         INT32           iter;
196         UINT32          adjust;
197         UINT32          mark = 0;
198         UINT32          count = sieveMarks[0].count;
199         UINT32          stop = sieveMarks[0].prime;
200         UINT32          composite;
201         UINT32          pList[8];
202         UINT32          next;
203
204         pAssert(field != NULL && bnN != NULL);
205
206         // If the remainder is odd, then subtracting the value will give an even number,
207         // but we want an odd number, so subtract the 105+rem. Otherwise, just subtract
208         // the even remainder.
209         adjust = (UINT32)BnModWord(bnN, 105);
210         if(adjust & 1)
211             adjust += 105;
212
213         // Adjust the input number so that it points to the first number in a
214         // aligned field.
215         BnSubWord(bnN, bnN, adjust);
216     //    pAssert(BnModWord(bnN, 105) == 0);
217         pField = field;
218         for(i = fieldSize; i >= sizeof(seedValues);
```

```
219             pField += sizeof(seedValues), i -= sizeof(seedValues))
220         {
221             memcpy(pField, seedValues, sizeof(seedValues));
222         }
223         if(i != 0)
224             memcpy(pField, seedValues, i);
225
226         // Cycle through the primes, clearing bits
227         // Have already done 3, 5, and 7
228         iter = 7;
229
230 #define NEXT_PRIME(iter)     (iter = RsaNextPrime(iter))
231         // Get the next N primes where N is determined by the mark in the sieveMarks
232         while((composite = NEXT_PRIME(iter)) != 0)
233         {
234             next = 0;
235             i = count;
236             pList[i--] = composite;
237             for(; i > 0; i--)
238             {
239                 next = NEXT_PRIME(iter);
240                 pList[i] = next;
241                 if(next != 0)
242                     composite *= next;
243             }
244             // Get the remainder when dividing the base field address
245             // by the composite
246             composite = (UINT32)BnModWord(bnN, composite);
247             // 'composite' is divisible by the composite components. for each of the
248             // composite components, divide 'composite'. That remainder (r) is used to
249             // pick a starting point for clearing the array. The stride is equal to the
250             // composite component. Note, the field only contains odd numbers. If the
251             // field were expanded to contain all numbers, then half of the bits would
252             // have already been cleared. We can save the trouble of clearing them a
253             // second time by having a stride of 2*next. Or we can take all of the even
254             // numbers out of the field and use a stride of 'next'
255             for(i = count; i > 0; i--)
256             {
257                 next = pList[i];
258                 if(next == 0)
259                     goto done;
260                 r = composite % next;
261             // these computations deal with the fact that we have picked a field-sized
262             // range that is aligned to a 105 count boundary. The problem is, this field
263             // only contains odd numbers. If we take our prime guess and walk through all
264             // the numbers using that prime as the 'stride', then every other 'stride' is
265             // going to be an even number. So, we are actually counting by 2 * the stride
266             // We want the count to start on an odd number at the start of our field. That
267             // is, we want to assume that we have counted up to the edge of the field by
268             // the 'stride' and now we are going to start flipping bits in the field as we
269             // continue to count up by 'stride'. If we take the base of our field and
270             // divide by the stride, we find out how much we find out how short the last
271             // count was from reaching the edge of the bit field. Say we get a quotient of
272             // 3 and remainder of 1. This means that after 3 strides, we are 1 short of
273             // the start of the field and the next stride will either land within the
274             // field or step completely over it. The confounding factor is that our field
275             // only contains odd numbers and our stride is actually 2 * stride. If the
276             // quoitent is even, then that means that when we add 2 * stride, we are going
277             // to hit another even number. So, we have to know if we need to back off
278             // by 1 stride before we start couting by 2 * stride.
279             // We can tell from the remainder whether we are on an even or odd
280             // stride when we hit the beginning of the table. If we are on an odd stride
281             // (r & 1), we would start half a stride in (next - r)/2. If we are on an
282             // even stride, we need 0.5 strides (next - r/2) because the table only has
283             // odd numbers. If the remainder happens to be zero, then the start of the
284             // table is on stride so no adjustment is necessary.
```

 Copyright © TCG 2006-2020

```
285             if(r & 1)           j = (next - r) / 2;
286             else if(r == 0)     j = 0;
287             else                 j = next - (r / 2);
288             for(; j < fieldBits; j += next)
289                 ClearBit(j, field, fieldSize);
290         }
291         if(next >= stop)
292         {
293             mark++;
294             count = sieveMarks[mark].count;
295             stop = sieveMarks[mark].prime;
296         }
297     }
298 done:
299     INSTRUMENT_INC(totalFieldsSieved[PrimeIndex]);
300     i = BitsInArray(field, fieldSize);
301     INSTRUMENT_ADD(bitsInFieldAfterSieve[PrimeIndex], i);
302     INSTRUMENT_ADD(emptyFieldsSieved[PrimeIndex], (i == 0));
303     return i;
304 }
305 #ifdef SIEVE_DEBUG
306 static uint32_t fieldSize = 210;
307
308 //***SetFieldSize()
309 // Function to set the field size used for prime generation. Used for tuning.
310 LIB_EXPORT uint32_t
311 SetFieldSize(
312     uint32_t         newFieldSize
313     )
314 {
315     if(newFieldSize == 0 || newFieldSize > MAX_FIELD_SIZE)
316         fieldSize = MAX_FIELD_SIZE;
317     else
318         fieldSize = newFieldSize;
319     return fieldSize;
320 }
321 #endif // SIEVE_DEBUG
```

10.2.15.1.4 PrimeSelectWithSieve()

This function will sieve the field around the input prime candidate. If the sieve field is not empty, one of the one bits in the field is chosen for testing with Miller-Rabin. If the value is prime, *pnP* is updated with this value and the function returns success. If this value is not prime, another pseudo-random candidate is chosen and tested. This process repeats until all values in the field have been checked. If all bits in the field have been checked and none is prime, the function returns FALSE and a new random value needs to be chosen.

Error Returns	Meaning
TPM_RC_FAILURE	TPM in failure mode, probably due to entropy source
TPM_RC_SUCCESS	candidate is probably prime
TPM_RC_NO_RESULT	candidate is not prime and couldn't find and alternative in the field

```
322 LIB_EXPORT TPM_RC
323 PrimeSelectWithSieve(
324     bigNum           candidate,         // IN/OUT: The candidate to filter
325     UINT32           e,                 // IN: the exponent
326     RAND_STATE      *rand               // IN: the random number generator state
327     )
328 {
329     BYTE             field[MAX_FIELD_SIZE];
330     UINT32           first;
```

```
331     UINT32          ones;
332     INT32           chosen;
333     BN_PRIME(test);
334     UINT32          modE;
335 #ifndef SIEVE_DEBUG
336     UINT32          fieldSize = MAX_FIELD_SIZE;
337 #endif
338     UINT32          primeSize;
339 //
340     // Adjust the field size and prime table list to fit the size of the prime
341     // being tested. This is done to try to optimize the trade-off between the
342     // dividing done for sieving and the time for Miller-Rabin. When the size
343     // of the prime is large, the cost of Miller-Rabin is fairly high, as is the
344     // cost of the sieving. However, the time for Miller-Rabin goes up considerably
345     // faster than the cost of dividing by a number of primes.
346     primeSize = BnSizeInBits(candidate);
347
348     if(primeSize <= 512)
349     {
350         RsaAdjustPrimeLimit(1024); // Use just the first 1024 primes
351     }
352     else if(primeSize <= 1024)
353     {
354         RsaAdjustPrimeLimit(4096); // Use just the first 4K primes
355     }
356     else
357     {
358         RsaAdjustPrimeLimit(0);     // Use all available
359     }
360
361     // Save the low-order word to use as a search generator and make sure that
362     // it has some interesting range to it
363     first = (UINT32)(candidate->d[0] | 0x80000000);
364
365     // Sieve the field
366     ones = PrimeSieve(candidate, fieldSize, field);
367     pAssert(ones > 0 && ones < (fieldSize * 8));
368     for(; ones > 0; ones--)
369     {
370         // Decide which bit to look at and find its offset
371         chosen = FindNthSetBit((UINT16)fieldSize, field, ((first % ones) + 1));
372
373         if((chosen < 0) || (chosen >= (INT32)(fieldSize * 8)))
374             FAIL(FATAL_ERROR_INTERNAL);
375
376         // Set this as the trial prime
377         BnAddWord(test, candidate, (crypt_uword_t)(chosen * 2));
378
379         // The exponent might not have been one of the tested primes so
380         // make sure that it isn't divisible and make sure that 0 != (p-1) mod e
381         // Note: This is the same as 1 != p mod e
382         modE = (UINT32)BnModWord(test, e);
383         if((modE != 0) && (modE != 1) && MillerRabin(test, rand))
384         {
385             BnCopy(candidate, test);
386             return TPM_RC_SUCCESS;
387         }
388         // Clear the bit just tested
389         ClearBit(chosen, field, fieldSize);
390     }
391     // Ran out of bits and couldn't find a prime in this field
392     INSTRUMENT_INC(noPrimeFields[PrimeIndex]);
393     return (g_inFailureMode ? TPM_RC_FAILURE : TPM_RC_NO_RESULT);
394 }
395 #if RSA_INSTRUMENT
396 static char             a[256];
```

```
397
398 //*** PrintTuple()
399 char *
400 PrintTuple(
401     UINT32      *i
402     )
403 {
404     sprintf(a, "{%d, %d, %d}", i[0], i[1], i[2]);
405     return a;
406 }
407
408 #define CLEAR_VALUE(x)    memset(x, 0, sizeof(x))
409
410 //*** RsaSimulationEnd()
411 void
412 RsaSimulationEnd(
413     void
414     )
415 {
416     int         i;
417     UINT32      averages[3];
418     UINT32      nonFirst = 0;
419     if((PrimeCounts[0] + PrimeCounts[1] + PrimeCounts[2]) != 0)
420     {
421         printf("Primes generated = %s\n", PrintTuple(PrimeCounts));
422         printf("Fields sieved = %s\n", PrintTuple(totalFieldsSieved));
423         printf("Fields with no primes = %s\n", PrintTuple(noPrimeFields));
424         printf("Primes checked with Miller-Rabin = %s\n",
425                PrintTuple(MillerRabinTrials));
426         for(i = 0; i < 3; i++)
427             averages[i] = (totalFieldsSieved[i]
428                            != 0 ? bitsInFieldAfterSieve[i] / totalFieldsSieved[i]
429                            : 0);
430         printf("Average candidates in field %s\n", PrintTuple(averages));
431         for(i = 1; i < (sizeof(failedAtIteration) / sizeof(failedAtIteration[0]));
432         i++)
433             nonFirst += failedAtIteration[i];
434         printf("Miller-Rabin failures not in first round = %d\n", nonFirst);
435
436     }
437     CLEAR_VALUE(PrimeCounts);
438     CLEAR_VALUE(totalFieldsSieved);
439     CLEAR_VALUE(noPrimeFields);
440     CLEAR_VALUE(MillerRabinTrials);
441     CLEAR_VALUE(bitsInFieldAfterSieve);
442 }
443
444 //*** GetSieveStats()
445 LIB_EXPORT void
446 GetSieveStats(
447     uint32_t        *trials,
448     uint32_t        *emptyFields,
449     uint32_t        *averageBits
450     )
451 {
452     uint32_t        totalBits;
453     uint32_t        fields;
454     *trials = MillerRabinTrials[0] + MillerRabinTrials[1] + MillerRabinTrials[2];
455     *emptyFields = noPrimeFields[0] + noPrimeFields[1] + noPrimeFields[2];
456     fields = totalFieldsSieved[0] + totalFieldsSieved[1]
457         + totalFieldsSieved[2];
458     totalBits = bitsInFieldAfterSieve[0] + bitsInFieldAfterSieve[1]
459         + bitsInFieldAfterSieve[2];
460     if(fields != 0)
461         *averageBits = totalBits / fields;
462     else
```

 Copyright © TCG 2006-2020

```
463             *averageBits = 0;
464         CLEAR_VALUE(PrimeCounts);
465         CLEAR_VALUE(totalFieldsSieved);
466         CLEAR_VALUE(noPrimeFields);
467         CLEAR_VALUE(MillerRabinTrials);
468         CLEAR_VALUE(bitsInFieldAfterSieve);
469
470     }
471     #endif
472
473     #endif // RSA_KEY_SIEVE
474
475     #if !RSA_INSTRUMENT
476
477     //*** RsaSimulationEnd()
478     // Stub for call when not doing instrumentation.
479     void
480     RsaSimulationEnd(
481         void
482         )
483     {
484         return;
485     }
486     #endif
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

10.2.16 CryptRand.c

10.2.16.1 Introduction

This file implements a DRBG with a behavior according to SP800-90A using a block cypher. This is also compliant to ISO/IEC 18031:2011(E) C.3.2.

A state structure is created for use by TPM.lib and functions within the CryptoEngine() my use their own state structures when they need to have deterministic values.

A debug mode is available that allows the random numbers generated for TPM.lib to be repeated during runs of the simulator. The switch for it is in TpmBuildSwitches.h. It is USE_DEBUG_RNG.

This is the implementation layer of CTR DRGB mechanism as defined in SP800-90A and the functions are organized as closely as practical to the organization in SP800-90A. It is intended to be compiled as a separate module that is linked with a secure application so that both reside inside the same boundary [SP 800-90A 8.5]. The secure application in particular manages the accesses protected storage for the state of the DRBG instantiations, and supplies the implementation functions here with a valid pointer to the working state of the given instantiations (as a DRBG_STATE structure).

This DRBG mechanism implementation does not support prediction resistance. Thus *prediction_resistance_flag* is omitted from Instantiate_function(), Reseed_function(), Generate_function() argument lists [SP 800-90A 9.1, 9.2, 9.3], as well as from the working state data structure DRBG_STATE [SP 800-90A 9.1].

This DRBG mechanism implementation always uses the highest security strength of available in the block ciphers. Thus *requested_security_strength* parameter is omitted from Instantiate_function() and Generate_function() argument lists [SP 800-90A 9.1, 9.2, 9.3], as well as from the working state data structure DRBG_STATE [SP 800-90A 9.1].

Internal functions (ones without Crypt prefix) expect validated arguments and therefore use assertions instead of runtime parameter checks and mostly return void instead of a status value.

```
1   #include "Tpm.h"
```

Pull in the test vector definitions and define the space

```
2   #include    "PRNG_TestVectors.h"
3   const BYTE DRBG_NistTestVector_Entropy[] = {DRBG_TEST_INITIATE_ENTROPY};
4   const BYTE DRBG_NistTestVector_GeneratedInterm[] =
5                                  {DRBG_TEST_GENERATED_INTERM};
6
7   const BYTE DRBG_NistTestVector_EntropyReseed[] =
8                                  {DRBG_TEST_RESEED_ENTROPY};
9   const BYTE DRBG_NistTestVector_Generated[] = {DRBG_TEST_GENERATED};
10
11  //** Derivation Functions
12  //*** Description
13  // The functions in this section are used to reduce the personalization input values
14  // to make them usable as input for reseeding and instantiation. The overall
15  // behavior is intended to produce the same results as described in SP800-90A,
16  // section 10.4.2 "Derivation Function Using a Block Cipher Algorithm
17  // (Block_Cipher_df)." The code is broken into several subroutines to deal with the
18  // fact that the data used for personalization may come in several separate blocks
19  // such as a Template hash and a proof value and a primary seed.
20
21  //*** Derivation Function Defines and Structures
22
23  #define     DF_COUNT (DRBG_KEY_SIZE_WORDS / DRBG_IV_SIZE_WORDS + 1)
24  #if DRBG_KEY_SIZE_BITS != 128 && DRBG_KEY_SIZE_BITS != 256
25  #   error "CryptRand.c only written for AES with 128- or 256-bit keys."
26  #endif
```

```
27  typedef struct
28  {
29      DRBG_KEY_SCHEDULE   keySchedule;
30      DRBG_IV             iv[DF_COUNT];
31      DRBG_IV             out1;
32      DRBG_IV             buf;
33      int                 contents;
34  } DF_STATE, *PDF_STATE;
```

10.2.16.1.1 DfCompute()

This function does the incremental update of the derivation function state. It encrypts the *iv* value and XOR's the results into each of the blocks of the output. This is equivalent to processing all of input data for each output block.

```
35  static void
36  DfCompute(
37      PDF_STATE        dfState
38      )
39  {
40      int              i;
41      int              iv;
42      crypt_uword_t   *pIv;
43      crypt_uword_t    temp[DRBG_IV_SIZE_WORDS] = {0};
44  //
45      for(iv = 0; iv < DF_COUNT; iv++)
46      {
47          pIv = (crypt_uword_t *)&dfState->iv[iv].words[0];
48          for(i = 0; i < DRBG_IV_SIZE_WORDS; i++)
49          {
50              temp[i] ^= pIv[i] ^ dfState->buf.words[i];
51          }
52          DRBG_ENCRYPT(&dfState->keySchedule, &temp, pIv);
53      }
54      for(i = 0; i < DRBG_IV_SIZE_WORDS; i++)
55          dfState->buf.words[i] = 0;
56      dfState->contents = 0;
57  }
```

10.2.16.1.2 DfStart()

This initializes the output blocks with an encrypted counter value and initializes the key schedule.

```
58  static void
59  DfStart(
60      PDF_STATE        dfState,
61      uint32_t         inputLength
62      )
63  {
64      BYTE            init[8];
65      int             i;
66      UINT32          drbgSeedSize = sizeof(DRBG_SEED);
67
68      const BYTE dfKey[DRBG_KEY_SIZE_BYTES] = {
69          0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
70          0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f
71      #if DRBG_KEY_SIZE_BYTES > 16
72          ,0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
73          0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f
74      #endif
75      };
76      memset(dfState, 0, sizeof(DF_STATE));
```

 Copyright © TCG 2006-2020

```
77      DRBG_ENCRYPT_SETUP(&dfKey[0], DRBG_KEY_SIZE_BITS, &dfState->keySchedule);
78      // Create the first chaining values
79      for(i = 0; i < DF_COUNT; i++)
80          ((BYTE *)&dfState->iv[i])[3] = (BYTE)i;
81      DfCompute(dfState);
82      // initialize the first 64 bits of the IV in a way that doesn't depend
83      // on the size of the words used.
84      UINT32_TO_BYTE_ARRAY(inputLength, init);
85      UINT32_TO_BYTE_ARRAY(drbgSeedSize, &init[4]);
86      memcpy(&dfState->iv[0], init, 8);
87      dfState->contents = 4;
88  }
```

10.2.16.1.3 DfUpdate()

This updates the state with the input data. A byte at a time is moved into the state buffer until it is full and then that block is encrypted by DfCompute().

```
89  static void
90  DfUpdate(
91      PDF_STATE        dfState,
92      int              size,
93      const BYTE      *data
94      )
95  {
96      while(size > 0)
97      {
98          int         toFill = DRBG_IV_SIZE_BYTES - dfState->contents;
99          if(size < toFill)
100             toFill = size;
101         // Copy as many bytes as there are or until the state buffer is full
102         memcpy(&dfState->buf.bytes[dfState->contents], data, toFill);
103         // Reduce the size left by the amount copied
104         size -= toFill;
105         // Advance the data pointer by the amount copied
106         data += toFill;
107         // increase the buffer contents count by the amount copied
108         dfState->contents += toFill;
109         pAssert(dfState->contents <= DRBG_IV_SIZE_BYTES);
110         // If we have a full buffer, do a computation pass.
111         if(dfState->contents == DRBG_IV_SIZE_BYTES)
112             DfCompute(dfState);
113     }
114 }
```

10.2.16.1.4 DfEnd()

This function is called to get the result of the derivation function computation. If the buffer is not full, it is padded with zeros. The output buffer is structured to be the same as a DRBG_SEED value so that the function can return a pointer to the DRBG_SEED value in the DF_STATE structure.

```
115 static DRBG_SEED *
116 DfEnd(
117     PDF_STATE        dfState
118     )
119 {
120     // Since DfCompute is always called when a buffer is full, there is always
121     // space in the buffer for the terminator
122     dfState->buf.bytes[dfState->contents++] = 0x80;
123     // If the buffer is not full, pad with zeros
124     while(dfState->contents < DRBG_IV_SIZE_BYTES)
125         dfState->buf.bytes[dfState->contents++] = 0;
```

```
126     // Do a final state update
127     DfCompute(dfState);
128     return (DRBG_SEED *)&dfState->iv;
129 }
```

10.2.16.1.5 DfBuffer()

Function to take an input buffer and do the derivation function to produce a DRBG_SEED value that can be used in DRBG_Reseed();

```
130 static DRBG_SEED *
131 DfBuffer(
132     DRBG_SEED       *output,        // OUT: receives the result
133     int              size,          // IN: size of the buffer to add
134     BYTE            *buf            // IN: address of the buffer
135     )
136 {
137     DF_STATE        dfState;
138     if(size == 0 || buf == NULL)
139         return NULL;
140     // Initialize the derivation function
141     DfStart(&dfState, size);
142     DfUpdate(&dfState, size, buf);
143     DfEnd(&dfState);
144     memcpy(output, &dfState.iv[0], sizeof(DRBG_SEED));
145     return output;
146 }
```

10.2.16.1.6 DRBG_GetEntropy()

Even though this implementation never fails, it may get blocked indefinitely long in the call to get entropy from the platform (DRBG_GetEntropy32()). This function is only used during instantiation of the DRBG for manufacturing and on each start-up after an non-orderly shutdown.

Return Value	Meaning
TRUE(1)	requested entropy returned
FALSE(0)	entropy Failure

```
147 BOOL
148 DRBG_GetEntropy(
149     UINT32           requiredEntropy,   // IN: requested number of bytes of full
150                                         //     entropy
151     BYTE            *entropy            // OUT: buffer to return collected entropy
152     )
153 {
154 #if !USE_DEBUG_RNG
155
156     UINT32      obtainedEntropy;
157     INT32       returnedEntropy;
158
159 // If in debug mode, always use the self-test values for initialization
160     if(IsSelfTest())
161     {
162 #endif
163         // If doing simulated DRBG, then check to see if the
164         // entropyFailure condition is being tested
165         if(!IsEntropyBad())
166         {
167             // In self-test, the caller should be asking for exactly the seed
168             // size of entropy.
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
169                 pAssert(requiredEntropy == sizeof(DRBG_NistTestVector_Entropy));
170                 memcpy(entropy, DRBG_NistTestVector_Entropy,
171                        sizeof(DRBG_NistTestVector_Entropy));
172             }
173     #if !USE_DEBUG_RNG
174         }
175         else if(!IsEntropyBad())
176         {
177             // Collect entropy
178             // Note: In debug mode, the only "entropy" value ever returned
179             // is the value of the self-test vector.
180             for(returnedEntropy = 1, obtainedEntropy = 0;
181                 obtainedEntropy < requiredEntropy && !IsEntropyBad();
182                 obtainedEntropy += returnedEntropy)
183             {
184                 returnedEntropy = _plat__GetEntropy(&entropy[obtainedEntropy],
185                                                     requiredEntropy - obtainedEntropy);
186                 if(returnedEntropy <= 0)
187                     SetEntropyBad();
188             }
189         }
190     #endif
191         return !IsEntropyBad();
192     }
```

10.2.16.1.7 IncrementIv()

This function increments the IV value by 1. It is used by EncryptDRBG().

```
193     void
194     IncrementIv(
195         DRBG_IV         *iv
196         )
197     {
198         BYTE      *ivP = ((BYTE *)iv) + DRBG_IV_SIZE_BYTES;
199         while((--ivP >= (BYTE *)iv) && ((*ivP = ((*ivP + 1) & 0xFF)) == 0));
200     }
```

10.2.16.1.8 EncryptDRBG()

This does the encryption operation for the DRBG. It will encrypt the input state counter (IV) using the state key. Into the output buffer for as many times as it takes to generate the required number of bytes.

```
201     static BOOL
202     EncryptDRBG(
203         BYTE                 *dOut,
204         UINT32                dOutBytes,
205         DRBG_KEY_SCHEDULE    *keySchedule,
206         DRBG_IV              *iv,
207         UINT32               *lastValue      // Points to the last output value
208         )
209     {
210     #if FIPS_COMPLIANT
211     // For FIPS compliance, the DRBG has to do a continuous self-test to make sure that
212     // no two consecutive values are the same. This overhead is not incurred if the TPM
213     // is not required to be FIPS compliant
214     //
215         UINT32           temp[DRBG_IV_SIZE_BYTES / sizeof(UINT32)];
216         int              i;
217         BYTE            *p;
218
219         for(; dOutBytes > 0;)
220         {
```

```
221             // Increment the IV before each encryption (this is what makes this
222             // different from normal counter-mode encryption
223             IncrementIv(iv);
224             DRBG_ENCRYPT(keySchedule, iv, temp);
225     // Expect a 16 byte block
226     #if DRBG_IV_SIZE_BITS != 128
227     #error  "Unsuppored IV size in DRBG"
228     #endif
229             if((lastValue[0] == temp[0])
230                 && (lastValue[1] == temp[1])
231                 && (lastValue[2] == temp[2])
232                 && (lastValue[3] == temp[3])
233                 )
234             {
235                 LOG_FAILURE(FATAL_ERROR_ENTROPY);
236                 return FALSE;
237             }
238             lastValue[0] = temp[0];
239             lastValue[1] = temp[1];
240             lastValue[2] = temp[2];
241             lastValue[3] = temp[3];
242             i = MIN(dOutBytes, DRBG_IV_SIZE_BYTES);
243             dOutBytes -= i;
244             for(p = (BYTE *)temp; i > 0; i--)
245                 *dOut++ = *p++;
246         }
247     #else // version without continuous self-test
248         NOT_REFERENCED(lastValue);
249         for(; dOutBytes >= DRBG_IV_SIZE_BYTES;
250         dOut = &dOut[DRBG_IV_SIZE_BYTES], dOutBytes -= DRBG_IV_SIZE_BYTES)
251         {
252             // Increment the IV
253             IncrementIv(iv);
254             DRBG_ENCRYPT(keySchedule, iv, dOut);
255         }
256         // If there is a partial, generate into a block-sized
257         // temp buffer and copy to the output.
258         if(dOutBytes != 0)
259         {
260             BYTE        temp[DRBG_IV_SIZE_BYTES];
261             // Increment the IV
262             IncrementIv(iv);
263             DRBG_ENCRYPT(keySchedule, iv, temp);
264             memcpy(dOut, temp, dOutBytes);
265         }
266     #endif
267         return TRUE;
268     }
```

10.2.16.1.9 DRBG_Update()

This function performs the state update function. According to SP800-90A, a temp value is created by doing CTR mode encryption of *providedData* and replacing the key and IV with these values. The one difference is that, with counter mode, the IV is incremented after each block is encrypted and in this operation, the counter is incremented before each block is encrypted. This function implements an *optimized* version of the algorithm in that it does the update of the *drbgState*->seed in place and then *providedData* is XORed into *drbgState*->seed to complete the encryption of *providedData*. This works because the IV is the last thing that gets encrypted.

```
269     static BOOL
270     DRBG_Update(
271         DRBG_STATE          *drbgState,     // IN:OUT state to update
272         DRBG_KEY_SCHEDULE   *keySchedule,   // IN: the key schedule (optional)
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
273     DRBG_SEED           *providedData   // IN: additional data
274     )
275 {
276     UINT32               i;
277     BYTE                *temp = (BYTE *)&drbgState->seed;
278     DRBG_KEY            *key = pDRBG_KEY(&drbgState->seed);
279     DRBG_IV             *iv = pDRBG_IV(&drbgState->seed);
280     DRBG_KEY_SCHEDULE    localKeySchedule;
281 //
282     pAssert(drbgState->magic == DRBG_MAGIC);
283
284     // If an key schedule was not provided, make one
285     if(keySchedule == NULL)
286     {
287         if(DRBG_ENCRYPT_SETUP((BYTE *)key,
288             DRBG_KEY_SIZE_BITS, &localKeySchedule) != 0)
289         {
290             LOG_FAILURE(FATAL_ERROR_INTERNAL);
291             return FALSE;
292         }
293         keySchedule = &localKeySchedule;
294     }
295     // Encrypt the temp value
296
297     EncryptDRBG(temp, sizeof(DRBG_SEED), keySchedule, iv,
298                 drbgState->lastValue);
299     if(providedData != NULL)
300     {
301         BYTE         *pP = (BYTE *)providedData;
302         for(i = DRBG_SEED_SIZE_BYTES; i != 0; i--)
303             *temp++ ^= *pP++;
304     }
305     // Since temp points to the input key and IV, we are done and
306     // don't need to copy the resulting 'temp' to drbgState->seed
307     return TRUE;
308 }
```

10.2.16.1.10 DRBG_Reseed()

This function is used when reseeding of the DRBG is required. If entropy is provided, it is used in lieu of using hardware entropy.

NOTE: the provided entropy must be the required size.

Return Value	Meaning
TRUE(1)	reseed succeeded
FALSE(0)	reseed failed, probably due to the entropy generation

```
309 BOOL
310 DRBG_Reseed(
311     DRBG_STATE          *drbgState,        // IN: the state to update
312     DRBG_SEED           *providedEntropy,  // IN: entropy
313     DRBG_SEED           *additionalData    // IN:
314     )
315 {
316     DRBG_SEED            seed;
317
318     pAssert((drbgState != NULL) && (drbgState->magic == DRBG_MAGIC));
319
320     if(providedEntropy == NULL)
321     {
322         providedEntropy = &seed;
```

 Copyright © TCG 2006-2020

```
323             if(!DRBG_GetEntropy(sizeof(DRBG_SEED), (BYTE *)providedEntropy))
324                 return FALSE;
325         }
326         if(additionalData != NULL)
327         {
328             unsigned int          i;
329
330             // XOR the provided data into the provided entropy
331             for(i = 0; i < sizeof(DRBG_SEED); i++)
332                 ((BYTE *)providedEntropy)[i] ^= ((BYTE *)additionalData)[i];
333         }
334         DRBG_Update(drbgState, NULL, providedEntropy);
335
336         drbgState->reseedCounter = 1;
337
338         return TRUE;
339     }
```

10.2.16.1.11 DRBG_SelfTest()

This is run when the DRBG is instantiated and at startup

Return Value	Meaning
TRUE(1)	test OK
FALSE(0)	test failed

```
340     BOOL
341     DRBG_SelfTest(
342         void
343         )
344     {
345         BYTE                buf[sizeof(DRBG_NistTestVector_Generated)];
346         DRBG_SEED           seed;
347         UINT32              i;
348         BYTE               *p;
349         DRBG_STATE          testState;
350     //
351         pAssert(!IsSelfTest());
352
353         SetSelfTest();
354         SetDrbgTested();
355         // Do an instantiate
356         if(!DRBG_Instantiate(&testState, 0, NULL))
357             return FALSE;
358     #if DRBG_DEBUG_PRINT
359         dbgDumpMemBlock(pDRBG_KEY(&testState), DRBG_KEY_SIZE_BYTES,
360                         "Key after Instantiate");
361         dbgDumpMemBlock(pDRBG_IV(&testState), DRBG_IV_SIZE_BYTES,
362                         "Value after Instantiate");
363     #endif
364         if(DRBG_Generate((RAND_STATE *)&testState, buf, sizeof(buf)) == 0)
365             return FALSE;
366     #if DRBG_DEBUG_PRINT
367         dbgDumpMemBlock(pDRBG_KEY(&testState.seed), DRBG_KEY_SIZE_BYTES,
368                         "Key after 1st Generate");
369         dbgDumpMemBlock(pDRBG_IV(&testState.seed), DRBG_IV_SIZE_BYTES,
370                         "Value after 1st Generate");
371     #endif
372         if(memcmp(buf, DRBG_NistTestVector_GeneratedInterm, sizeof(buf)) != 0)
373             return FALSE;
374         memcpy(seed.bytes, DRBG_NistTestVector_EntropyReseed, sizeof(seed));
375         DRBG_Reseed(&testState, &seed, NULL);
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
376  #if DRBG_DEBUG_PRINT
377      dbgDumpMemBlock((BYTE *)pDRBG_KEY(&testState.seed), DRBG_KEY_SIZE_BYTES,
378                      "Key after 2nd Generate");
379      dbgDumpMemBlock((BYTE *)pDRBG_IV(&testState.seed), DRBG_IV_SIZE_BYTES,
380                      "Value after 2nd Generate");
381      dbgDumpMemBlock(buf, sizeof(buf), "2nd Generated");
382  #endif
383      if(DRBG_Generate((RAND_STATE *)&testState, buf, sizeof(buf)) == 0)
384          return FALSE;
385      if(memcmp(buf, DRBG_NistTestVector_Generated, sizeof(buf)) != 0)
386          return FALSE;
387      ClearSelfTest();
388
389      DRBG_Uninstantiate(&testState);
390      for(p = (BYTE *)&testState, i = 0; i < sizeof(DRBG_STATE); i++)
391      {
392          if(*p++)
393              return FALSE;
394      }
395      // Simulate hardware failure to make sure that we get an error when
396      // trying to instantiate
397      SetEntropyBad();
398      if(DRBG_Instantiate(&testState, 0, NULL))
399         return FALSE;
400      ClearEntropyBad();
401
402      return TRUE;
403  }
```

10.2.16.2 Public Interface

10.2.16.2.1 Description

The functions in this section are the interface to the RNG. These are the functions that are used by TPM.lib.

10.2.16.2.2 CryptRandomStir()

This function is used to cause a reseed. A DRBG_SEED amount of entropy is collected from the hardware and then additional data is added.

Error Returns	Meaning
TPM_RC_NO_RESULT	failure of the entropy generator

```
404  LIB_EXPORT TPM_RC
405  CryptRandomStir(
406      UINT16           additionalDataSize,
407      BYTE            *additionalData
408      )
409  {
410  #if !USE_DEBUG_RNG
411      DRBG_SEED        tmpBuf;
412      DRBG_SEED        dfResult;
413  //
414      // All reseed with outside data starts with a buffer full of entropy
415      if(!DRBG_GetEntropy(sizeof(tmpBuf), (BYTE *)&tmpBuf))
416          return TPM_RC_NO_RESULT;
417
418      DRBG_Reseed(&drbgDefault, &tmpBuf,
419                  DfBuffer(&dfResult, additionalDataSize, additionalData));
420      drbgDefault.reseedCounter = 1;
```

 Copyright © TCG 2006-2020

```
421
422         return TPM_RC_SUCCESS;
423
424     #else
425         // If doing debug, use the input data as the initial setting for the RNG state
426         // so that the test can be reset at any time.
427         // Note: If this is called with a data size of 0 or less, nothing happens. The
428         // presumption is that, in a debug environment, the caller will have specific
429         // values for initialization, so this check is just a simple way to prevent
430         // inadvertent programming errors from screwing things up. This doesn't use an
431         // pAssert() because the non-debug version of this function will accept these
432         // parameters as meaning that there is no additionalData and only hardware
433         // entropy is used.
434         if((additionalDataSize > 0) && (additionalData != NULL))
435         {
436             memset(drbgDefault.seed.bytes, 0, sizeof(drbgDefault.seed.bytes));
437             memcpy(drbgDefault.seed.bytes, additionalData,
438                    MIN(additionalDataSize, sizeof(drbgDefault.seed.bytes)));
439         }
440         drbgDefault.reseedCounter = 1;
441
442         return TPM_RC_SUCCESS;
443     #endif
444     }
```

10.2.16.2.3 CryptRandomGenerate()

Generate a *randomSize* number or random bytes.

```
445     LIB_EXPORT UINT16
446     CryptRandomGenerate(
447         UINT16           randomSize,
448         BYTE            *buffer
449         )
450     {
451         return DRBG_Generate((RAND_STATE *)&drbgDefault, buffer, randomSize);
452     }
```

10.2.16.2.4 DRBG_InstantiateSeededKdf()

This function is used to instantiate a KDF-based RNG. This is used for derivations. This function always returns TRUE.

```
453     LIB_EXPORT BOOL
454     DRBG_InstantiateSeededKdf(
455         KDF_STATE       *state,          // OUT: buffer to hold the state
456         TPM_ALG_ID       hashAlg,        // IN: hash algorithm
457         TPM_ALG_ID       kdf,            // IN: the KDF to use
458         TPM2B           *seed,           // IN: the seed to use
459         const TPM2B     *label,          // IN: a label for the generation process.
460         TPM2B           *context,        // IN: the context value
461         UINT32           limit           // IN: Maximum number of bits from the KDF
462         )
463     {
464         state->magic = KDF_MAGIC;
465         state->limit = limit;
466         state->seed = seed;
467         state->hash = hashAlg;
468         state->kdf = kdf;
469         state->label = label;
470         state->context = context;
471         state->digestSize = CryptHashGetDigestSize(hashAlg);
472         state->counter = 0;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
473     state->residual.t.size = 0;
474     return TRUE;
475 }
```

10.2.16.2.5 DRBG_AdditionalData()

Function to reseed the DRBG with additional entropy. This is normally called before computing the protection value of a primary key in the Endorsement hierarchy.

```
476 LIB_EXPORT void
477 DRBG_AdditionalData(
478     DRBG_STATE      *drbgState,     // IN:OUT state to update
479     TPM2B           *additionalData // IN: value to incorporate
480     )
481 {
482     DRBG_SEED        dfResult;
483     if(drbgState->magic == DRBG_MAGIC)
484     {
485         DfBuffer(&dfResult, additionalData->size, additionalData->buffer);
486         DRBG_Reseed(drbgState, &dfResult, NULL);
487     }
488 }
```

10.2.16.2.6 DRBG_InstantiateSeeded()

This function is used to instantiate a random number generator from seed values. The nominal use of this generator is to create sequences of pseudo-random numbers from a seed value.

Error Returns	Meaning
TPM_RC_FAILURE	DRBG self-test failure

```
489 LIB_EXPORT TPM_RC
490 DRBG_InstantiateSeeded(
491     DRBG_STATE      *drbgState,     // IN/OUT: buffer to hold the state
492     const TPM2B     *seed,          // IN: the seed to use
493     const TPM2B     *purpose,       // IN: a label for the generation process.
494     const TPM2B     *name,          // IN: name of the object
495     const TPM2B     *additional     // IN: additional data
496     )
497 {
498     DF_STATE         dfState;
499     int              totalInputSize;
500     // DRBG should have been tested, but...
501     if(!IsDrbgTested() && !DRBG_SelfTest())
502     {
503         LOG_FAILURE(FATAL_ERROR_SELF_TEST);
504         return TPM_RC_FAILURE;
505     }
506     // Initialize the DRBG state
507     memset(drbgState, 0, sizeof(DRBG_STATE));
508     drbgState->magic = DRBG_MAGIC;
509
510     // Size all of the values
511     totalInputSize = (seed != NULL) ? seed->size : 0;
512     totalInputSize += (purpose != NULL) ? purpose->size : 0;
513     totalInputSize += (name != NULL) ? name->size : 0;
514     totalInputSize += (additional != NULL) ? additional->size : 0;
515
516     // Initialize the derivation
517     DfStart(&dfState, totalInputSize);
518
```

 Copyright © TCG 2006-2020

```
519     // Run all the input strings through the derivation function
520     if(seed != NULL)
521         DfUpdate(&dfState, seed->size, seed->buffer);
522     if(purpose != NULL)
523         DfUpdate(&dfState, purpose->size, purpose->buffer);
524     if(name != NULL)
525         DfUpdate(&dfState, name->size, name->buffer);
526     if(additional != NULL)
527         DfUpdate(&dfState, additional->size, additional->buffer);
528
529     // Used the derivation function output as the "entropy" input. This is not
530     // how it is described in SP800-90A but this is the equivalent function
531     DRBG_Reseed(((DRBG_STATE *)drbgState), DfEnd(&dfState), NULL);
532
533     return TPM_RC_SUCCESS;
534 }
```

10.2.16.2.7 CryptRandStartup()

This function is called when TPM_Startup is executed. This function always returns TRUE.

```
535 LIB_EXPORT BOOL
536 CryptRandStartup(
537     void
538     )
539 {
540 #if ! _DRBG_STATE_SAVE
541     // If not saved in NV, re-instantiate on each startup
542     DRBG_Instantiate(&drbgDefault, 0, NULL);
543 #else
544     // If the running state is saved in NV, NV has to be loaded before it can
545     // be updated
546     if(go.drbgState.magic == DRBG_MAGIC)
547         DRBG_Reseed(&go.drbgState, NULL, NULL);
548     else
549         DRBG_Instantiate(&go.drbgState, 0, NULL);
550 #endif
551     return TRUE;
552 }
```

10.2.16.2.7.1 CryptRandInit()

This function is called when _TPM_Init() is being processed.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
553 LIB_EXPORT BOOL
554 CryptRandInit(
555     void
556     )
557 {
558 #if !USE_DEBUG_RNG
559     _plat__GetEntropy(NULL, 0);
560 #endif
561     return DRBG_SelfTest();
562 }
```

10.2.16.2.8 DRBG_Generate()

This function generates a random sequence according SP800-90A. If *random* is not NULL, then *randomSize* bytes of random values are generated. If *random* is NULL or *randomSize* is zero, then the function returns zero without generating any bits or updating the reseed counter. This function returns the number of bytes produced which could be less than the number requested if the request is too large ("too large" is implementation dependent.)

```
563 LIB_EXPORT UINT16
564 DRBG_Generate(
565     RAND_STATE      *state,
566     BYTE            *random,        // OUT: buffer to receive the random values
567     UINT16           randomSize     // IN: the number of bytes to generate
568     )
569 {
570     if(state == NULL)
571         state = (RAND_STATE *)&drbgDefault;
572     if(random == NULL)
573         return 0;
574
575     // If the caller used a KDF state, generate a sequence from the KDF not to
576     // exceed the limit.
577     if(state->kdf.magic == KDF_MAGIC)
578     {
579         KDF_STATE       *kdf = (KDF_STATE *)state;
580         UINT32           counter = (UINT32)kdf->counter;
581         INT32            bytesLeft = randomSize;
582 //
583         // If the number of bytes to be returned would put the generator
584         // over the limit, then return 0
585         if((((kdf->counter * kdf->digestSize) + randomSize) * 8) > kdf->limit)
586             return 0;
587         // Process partial and full blocks until all requested bytes provided
588         while(bytesLeft > 0)
589         {
590             // If there is any residual data in the buffer, copy it to the output
591             // buffer
592             if(kdf->residual.t.size > 0)
593             {
594                 INT32      size;
595 //
```

```
596                 // Don't use more of the residual than will fit or more than are
597                 // available
598                 size = MIN(kdf->residual.t.size, bytesLeft);
599
600                 // Copy some or all of the residual to the output. The residual is
601                 // at the end of the buffer. The residual might be a full buffer.
602                 MemoryCopy(random,
603                            &kdf->residual.t.buffer
604                            [kdf->digestSize - kdf->residual.t.size], size);
605
606                 // Advance the buffer pointer
607                 random += size;
608
609                 // Reduce the number of bytes left to get
610                 bytesLeft -= size;
611
612                 // And reduce the residual size appropriately
613                 kdf->residual.t.size -= (UINT16)size;
614             }
615             else
616             {
617                 UINT16           blocks = (UINT16)(bytesLeft / kdf->digestSize);
618 //
619                 // Get the number of required full blocks
620                 if(blocks > 0)
621                 {
622                     UINT16      size = blocks * kdf->digestSize;
623 // Get some number of full blocks and put them in the return buffer
624                     CryptKDFa(kdf->hash, kdf->seed, kdf->label, kdf->context, NULL,
625                               kdf->limit, random, &counter, blocks);
626
627                     // reduce the size remaining to be moved and advance the pointer
628                     bytesLeft -= size;
629                     random += size;
630                 }
631                 else
632                 {
633                     // Fill the residual buffer with a full block and then loop to
634                     // top to get part of it copied to the output.
635                     kdf->residual.t.size = CryptKDFa(kdf->hash, kdf->seed,
636                                                      kdf->label, kdf->context, NULL,
637                                                      kdf->limit,
638                                                      kdf->residual.t.buffer,
639                                                      &counter, 1);
640                 }
641             }
642         }
643         kdf->counter = counter;
644         return randomSize;
645     }
646     else if(state->drbg.magic == DRBG_MAGIC)
647     {
648         DRBG_STATE           *drbgState = (DRBG_STATE *)state;
649         DRBG_KEY_SCHEDULE     keySchedule;
650         DRBG_SEED            *seed = &drbgState->seed;
651
652         if(drbgState->reseedCounter >= CTR_DRBG_MAX_REQUESTS_PER_RESEED)
653         {
654             if(drbgState == &drbgDefault)
655             {
656                 DRBG_Reseed(drbgState, NULL, NULL);
657                 if(IsEntropyBad() && !IsSelfTest())
658                     return 0;
659             }
660             else
661             {
```

```
662                 // If this is a PRNG then the only way to get
663                 // here is if the SW has run away.
664                 LOG_FAILURE(FATAL_ERROR_INTERNAL);
665                 return 0;
666             }
667         }
668         // if the allowed number of bytes in a request is larger than the
669         // less than the number of bytes that can be requested, then check
670 #if UINT16_MAX >=  CTR_DRBG_MAX_BYTES_PER_REQUEST
671         if(randomSize > CTR_DRBG_MAX_BYTES_PER_REQUEST)
672             randomSize = CTR_DRBG_MAX_BYTES_PER_REQUEST;
673 #endif
674         // Create  encryption schedule
675         if(DRBG_ENCRYPT_SETUP((BYTE *)pDRBG_KEY(seed),
676                               DRBG_KEY_SIZE_BITS, &keySchedule) != 0)
677         {
678             LOG_FAILURE(FATAL_ERROR_INTERNAL);
679             return 0;
680         }
681         // Generate the random data
682         EncryptDRBG(random, randomSize, &keySchedule, pDRBG_IV(seed),
683                     drbgState->lastValue);
684         // Do a key update
685         DRBG_Update(drbgState, &keySchedule, NULL);
686
687         // Increment the reseed counter
688         drbgState->reseedCounter += 1;
689     }
690     else
691     {
692         LOG_FAILURE(FATAL_ERROR_INTERNAL);
693         return FALSE;
694     }
695     return randomSize;
696 }
```

10.2.16.2.9 DRBG_Instantiate()

This is CTR_DRBG_Instantiate_algorithm() from [SP 800-90A 10.2.1.3.1]. This is called when a the TPM DRBG is to be instantiated. This is called to instantiate a DRBG used by the TPM for normal operations.

Return Value	Meaning
TRUE(1)	instantiation succeeded
FALSE(0)	instantiation failed

```
697 LIB_EXPORT BOOL
698 DRBG_Instantiate(
699     DRBG_STATE      *drbgState,         // OUT: the instantiated value
700     UINT16           pSize,             // IN: Size of personalization string
701     BYTE            *personalization    // IN: The personalization string
702     )
703 {
704     DRBG_SEED        seed;
705     DRBG_SEED        dfResult;
706 //
707     pAssert((pSize == 0) || (pSize <= sizeof(seed)) || (personalization != NULL));
708     // If the DRBG has not been tested, test when doing an instantiation. Since
709     // Instantiation is called during self test, make sure we don't get stuck in a
710     // loop.
711     if(!IsDrbgTested() && !IsSelfTest() && !DRBG_SelfTest())
712         return FALSE;
713     // If doing a self test, DRBG_GetEntropy will return the NIST
```

```
714         // test vector value.
715         if(!DRBG_GetEntropy(sizeof(seed), (BYTE *)&seed))
716             return FALSE;
717         // set everything to zero
718         memset(drbgState, 0, sizeof(DRBG_STATE));
719         drbgState->magic = DRBG_MAGIC;
720
721         // Steps 1, 2, 3, 6, 7 of SP 800-90A 10.2.1.3.1 are exactly what
722         // reseeding does. So, do a reduction on the personalization value (if any)
723         // and do a reseed.
724         DRBG_Reseed(drbgState, &seed, DfBuffer(&dfResult, pSize, personalization));
725
726         return TRUE;
727     }
```

10.2.16.2.10 DRBG_Uninstantiate()

This is Uninstantiate_function() from [SP 800-90A 9.4].

Error Returns	Meaning
TPM_RC_VALUE	not a valid state

```
728     LIB_EXPORT TPM_RC
729     DRBG_Uninstantiate(
730         DRBG_STATE      *drbgState      // IN/OUT: working state to erase
731         )
732     {
733         if((drbgState == NULL) || (drbgState->magic != DRBG_MAGIC))
734             return TPM_RC_VALUE;
735         memset(drbgState, 0, sizeof(DRBG_STATE));
736         return TPM_RC_SUCCESS;
737     }
```

10.2.17 CryptRsa.c

10.2.17.1 Introduction

This file contains implementation of cryptographic primitives for RSA. Vendors may replace the implementation in this file with their own library functions.

10.2.17.2 Includes

Need this define to get the *private* defines for this function

```
1   #define CRYPT_RSA_C
2   #include "Tpm.h"
3   #if ALG_RSA
```

10.2.17.3 Obligatory Initialization Functions

10.2.17.3.1 CryptRsaInit()

Function called at _TPM_Init().

```
4   BOOL
5   CryptRsaInit(
6       void
7       )
8   {
9       return TRUE;
10  }
```

10.2.17.3.2 CryptRsaStartup()

Function called at TPM2_Startup()

```
11  BOOL
12  CryptRsaStartup(
13      void
14      )
15  {
16      return TRUE;
17  }
```

10.2.17.4 Internal Functions

10.2.17.4.1 RsaInitializeExponent()

This function initializes the bignum data structure that holds the private exponent. This function returns the pointer to the private exponent value so that it can be used in an initializer for a data declaration.

```
18  static privateExponent *
19  RsaInitializeExponent(
20      privateExponent       *Z
21      )
22  {
23      bigNum                *bn = (bigNum *)&Z->P;
24      int                    i;
25  //
```

```
26      for(i = 0; i < 5; i++)
27      {
28          bn[i] = (bigNum)&Z->entries[i];
29          BnInit(bn[i], BYTES_TO_CRYPT_WORDS(sizeof(Z->entries[0].d)));
30      }
31      return Z;
32  }
```

10.2.17.4.2 MakePgreaterThanQ()

This function swaps the pointers for P and Q if Q happens to be larger than Q.

```
33  static void
34  MakePgreaterThanQ(
35      privateExponent          *Z
36  )
37  {
38      if(BnUnsignedCmp(Z->P, Z->Q) < 0)
39      {
40          bigNum          bnT = Z->P;
41          Z->P = Z->Q;
42          Z->Q = bnT;
43      }
44  }
```

10.2.17.4.3 PackExponent()

This function takes the bignum private exponent and converts it into TPM2B form. In this form, the size field contains the overall size of the packed data. The buffer contains 5, equal sized values in P, Q, *dP*, *dQ*, *qInv* order. For example, if a key has a 2Kb public key, then the packed private key will contain 5, 1Kb values. This form makes it relatively easy to load and save the values without changing the normal unmarshaling to do anything more than allow a larger TPM2B for the private key. Also, when exporting the value, all that is needed is to change the size field of the private key in order to save just the P value.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure // The data is too big to fit

```
45  static BOOL
46  PackExponent(
47      TPM2B_PRIVATE_KEY_RSA       *packed,
48      privateExponent             *Z
49  )
50  {
51      int                         i;
52      UINT16                      primeSize = (UINT16)BITS_TO_BYTES(BnMsb(Z->P));
53      UINT16                      pS = primeSize;
54  //
55      pAssert((primeSize * 5) <= sizeof(packed->t.buffer));
56      packed->t.size = (primeSize * 5) + RSA_prime_flag;
57      for(i = 0; i < 5; i++)
58          if(!BnToBytes((bigNum)&Z->entries[i], &packed->t.buffer[primeSize * i], &pS))
59              return FALSE;
60      if(pS != primeSize)
61          return FALSE;
62      return TRUE;
63  }
```

 Copyright © TCG 2006-2020

10.2.17.4.4 UnpackExponent()

This function unpacks the private exponent from its TPM2B form into its bignum form.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	TPM2B is not the correct size

```
64  static BOOL
65  UnpackExponent(
66      TPM2B_PRIVATE_KEY_RSA       *b,
67      privateExponent             *Z
68  )
69  {
70          UINT16                 primeSize = b->t.size & ~RSA_prime_flag;
71          int                    i;
72          bigNum                *bn = &Z->P;
73  //
74      VERIFY(b->t.size & RSA_prime_flag);
75      RsaInitializeExponent(Z);
76      VERIFY((primeSize % 5) == 0);
77      primeSize /= 5;
78      for(i = 0; i < 5; i++)
79          VERIFY(BnFromBytes(bn[i], &b->t.buffer[primeSize * i], primeSize)
80                  != NULL);
81      MakePgreaterThanQ(Z);
82      return TRUE;
83  Error:
84      return FALSE;
85   }
```

10.2.17.4.5 ComputePrivateExponent()

This function computes the private exponent from the primes.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
86  static BOOL
87  ComputePrivateExponent(
88      bigNum                pubExp,         // IN: the public exponent
89      privateExponent      *Z               // IN/OUT: on input, has primes P and Q. On
90                                            //         output, has P, Q, dP, dQ, and pInv
91      )
92  {
93      BOOL                 pOK;
94      BOOL                 qOK;
95      BN_PRIME(pT);
96  //
97      // make p the larger value so that m2 is always less than p
98      MakePgreaterThanQ(Z);
99
100     //dP = (1/e) mod (p-1)
101     pOK = BnSubWord(pT, Z->P, 1);
102     pOK = pOK && BnModInverse(Z->dP, pubExp, pT);
103     //dQ = (1/e) mod (q-1)
104     qOK = BnSubWord(pT, Z->Q, 1);
105     qOK = qOK && BnModInverse(Z->dQ, pubExp, pT);
106     // qInv = (1/q) mod p
```

```
107         if(pOK && qOK)
108             pOK = qOK = BnModInverse(Z->qInv, Z->Q, Z->P);
109         if(!pOK)
110             BnSetWord(Z->P, 0);
111         if(!qOK)
112             BnSetWord(Z->Q, 0);
113         return pOK && qOK;
114     }
```

10.2.17.4.6 RsaPrivateKeyOp()

This function is called to do the exponentiation with the private key. Compile options allow use of the simple (but slow) private exponent, or the more complex but faster CRT method.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
115     static BOOL
116     RsaPrivateKeyOp(
117         bigNum               inOut, // IN/OUT: number to be exponentiated
118         privateExponent     *Z
119         )
120     {
121         BN_RSA(M1);
122         BN_RSA(M2);
123         BN_RSA(M);
124         BN_RSA(H);
125     //
126         MakePgreaterThanQ(Z);
127         // m1 = cdP mod p
128         VERIFY(BnModExp(M1, inOut, Z->dP, Z->P));
129         // m2 = cdQ mod q
130         VERIFY(BnModExp(M2, inOut, Z->dQ, Z->Q));
131         // h = qInv * (m1 - m2) mod p = qInv * (m1 + P - m2) mod P because Q < P
132         // so m2 < P
133         VERIFY(BnSub(H, Z->P, M2));
134         VERIFY(BnAdd(H, H, M1));
135         VERIFY(BnModMult(H, H, Z->qInv, Z->P));
136         // m = m2 + h * q
137         VERIFY(BnMult(M, H, Z->Q));
138         VERIFY(BnAdd(inOut, M2, M));
139         return TRUE;
140     Error:
141         return FALSE;
142     }
```

10.2.17.4.7 RSAEP()

This function performs the RSAEP operation defined in PKCS#1v2.1. It is an exponentiation of a value (*m*) with the public exponent (*e*), modulo the public (*n*).

Error Returns	Meaning
TPM_RC_VALUE	number to exponentiate is larger than the modulus

```
143     static TPM_RC
144     RSAEP(
145         TPM2B       *dInOut,        // IN: size of the encrypted block and the size of
146                                     //     the encrypted value. It must be the size of
```

```
147                                     //      the modulus.
148                                     // OUT: the encrypted data. Will receive the
149                                     //      decrypted value
150         OBJECT      *key            // IN: the key to use
151         )
152     {
153         TPM2B_TYPE(4BYTES, 4);
154         TPM2B_4BYTES        e2B;
155         UINT32              e = key->publicArea.parameters.rsaDetail.exponent;
156     //
157         if(e == 0)
158             e = RSA_DEFAULT_PUBLIC_EXPONENT;
159         UINT32_TO_BYTE_ARRAY(e, e2B.t.buffer);
160         e2B.t.size = 4;
161         return ModExpB(dInOut->size, dInOut->buffer, dInOut->size, dInOut->buffer,
162                        e2B.t.size, e2B.t.buffer, key->publicArea.unique.rsa.t.size,
163                        key->publicArea.unique.rsa.t.buffer);
164     }
```

10.2.17.4.8 RSADP()

This function performs the RSADP operation defined in PKCS#1v2.1. It is an exponentiation of a value (*c*) with the private exponent (*d*), modulo the public modulus (*n*). The decryption is in place.

This function also checks the size of the private key. If the size indicates that only a prime value is present, the key is converted to being a private exponent.

Error Returns	Meaning
TPM_RC_SIZE	the value to decrypt is larger than the modulus

```
165     static TPM_RC
166     RSADP(
167         TPM2B           *inOut,         // IN/OUT: the value to encrypt
168         OBJECT          *key            // IN: the key
169         )
170     {
171         BN_RSA_INITIALIZED(bnM, inOut);
172         NEW_PRIVATE_EXPONENT(Z);
173         if(UnsignedCompareB(inOut->size, inOut->buffer,
174                             key->publicArea.unique.rsa.t.size,
175                             key->publicArea.unique.rsa.t.buffer) >= 0)
176             return TPM_RC_SIZE;
177         // private key operation requires that private exponent be loaded
178         // During self-test, this might not be the case so load it up if it hasn't
179         // already done
180         // been done
181         if((key->sensitive.sensitive.rsa.t.size & RSA_prime_flag) == 0)
182         {
183             if(CryptRsaLoadPrivateExponent(&key->publicArea, &key->sensitive)
184                != TPM_RC_SUCCESS)
185                 return TPM_RC_BINDING;
186         }
187         VERIFY(UnpackExponent(&key->sensitive.sensitive.rsa, Z));
188         VERIFY(RsaPrivateKeyOp(bnM, Z));
189         VERIFY(BnTo2B(bnM, inOut, inOut->size));
190         return TPM_RC_SUCCESS;
191     Error:
192         return TPM_RC_FAILURE;
193     }
```

10.2.17.4.9 OaepEncode()

This function performs OAEP padding. The size of the buffer to receive the OAEP padded data must equal the size of the modulus

Error Returns	Meaning
TPM_RC_VALUE	*hashAlg* is not valid or message size is too large

```
194  static TPM_RC
195  OaepEncode(
196      TPM2B        *padded,         // OUT: the pad data
197      TPM_ALG_ID    hashAlg,        // IN: algorithm to use for padding
198      const TPM2B *label,           // IN: null-terminated string (may be NULL)
199      TPM2B        *message,        // IN: the message being padded
200      RAND_STATE   *rand            // IN: the random number generator to use
201      )
202  {
203      INT32         padLen;
204      INT32         dbSize;
205      INT32         i;
206      BYTE          mySeed[MAX_DIGEST_SIZE];
207      BYTE         *seed = mySeed;
208      UINT16        hLen = CryptHashGetDigestSize(hashAlg);
209      BYTE          mask[MAX_RSA_KEY_BYTES];
210      BYTE         *pp;
211      BYTE         *pm;
212      TPM_RC        retVal = TPM_RC_SUCCESS;
213
214      pAssert(padded != NULL && message != NULL);
215
216      // A value of zero is not allowed because the KDF can't produce a result
217      // if the digest size is zero.
218      if(hLen == 0)
219          return TPM_RC_VALUE;
220
221      // Basic size checks
222      //  make sure digest isn't too big for key size
223      if(padded->size < (2 * hLen) + 2)
224          ERROR_RETURN(TPM_RC_HASH);
225
226      // and that message will fit messageSize <= k - 2hLen - 2
227      if(message->size > (padded->size - (2 * hLen) - 2))
228          ERROR_RETURN(TPM_RC_VALUE);
229
230      // Hash L even if it is null
231      // Offset into padded leaving room for masked seed and byte of zero
232      pp = &padded->buffer[hLen + 1];
233      if(CryptHashBlock(hashAlg, label->size, (BYTE *)label->buffer,
234                        hLen, pp) != hLen)
235          ERROR_RETURN(TPM_RC_FAILURE);
236
237      // concatenate PS of k  mLen  2hLen  2
238      padLen = padded->size - message->size - (2 * hLen) - 2;
239      MemorySet(&pp[hLen], 0, padLen);
240      pp[hLen + padLen] = 0x01;
241      padLen += 1;
242      memcpy(&pp[hLen + padLen], message->buffer, message->size);
243
244      // The total size of db = hLen + pad + mSize;
245      dbSize = hLen + padLen + message->size;
246
247      // If testing, then use the provided seed. Otherwise, use values
248      // from the RNG
249      CryptRandomGenerate(hLen, mySeed);
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
250         DRBG_Generate(rand, mySeed, (UINT16)hLen);
251         if(g_inFailureMode)
252             ERROR_RETURN(TPM_RC_FAILURE);
253         // mask = MGF1 (seed, nSize  hLen  1)
254         CryptMGF1(dbSize, mask, hashAlg, hLen, seed);
255
256         // Create the masked db
257         pm = mask;
258         for(i = dbSize; i > 0; i--)
259             *pp++ ^= *pm++;
260         pp = &padded->buffer[hLen + 1];
261
262         // Run the masked data through MGF1
263         if(CryptMGF1(hLen, &padded->buffer[1], hashAlg, dbSize, pp) != (unsigned)hLen)
264             ERROR_RETURN(TPM_RC_VALUE);
265     // Now XOR the seed to create masked seed
266         pp = &padded->buffer[1];
267         pm = seed;
268         for(i = hLen; i > 0; i--)
269             *pp++ ^= *pm++;
270         // Set the first byte to zero
271         padded->buffer[0] = 0x00;
272     Exit:
273         return retVal;
274     }
```

10.2.17.4.10 OaepDecode()

This function performs OAEP padding checking. The size of the buffer to receive the recovered data. If the padding is not valid, the *dSize* size is set to zero and the function returns TPM_RC_VALUE.

The *dSize* parameter is used as an input to indicate the size available in the buffer. If insufficient space is available, the size is not changed and the return code is TPM_RC_VALUE.

Error Returns	Meaning
TPM_RC_VALUE	the value to decode was larger than the modulus, or the padding is wrong or the buffer to receive the results is too small

```
275     static TPM_RC
276     OaepDecode(
277         TPM2B           *dataOut,        // OUT: the recovered data
278         TPM_ALG_ID       hashAlg,        // IN: algorithm to use for padding
279         const TPM2B     *label,          // IN: null-terminated string (may be NULL)
280         TPM2B           *padded          // IN: the padded data
281         )
282     {
283         UINT32        i;
284         BYTE          seedMask[MAX_DIGEST_SIZE];
285         UINT32        hLen = CryptHashGetDigestSize(hashAlg);
286
287         BYTE          mask[MAX_RSA_KEY_BYTES];
288         BYTE         *pp;
289         BYTE         *pm;
290         TPM_RC        retVal = TPM_RC_SUCCESS;
291
292         // Strange size (anything smaller can't be an OAEP padded block)
293         // Also check for no leading 0
294         if((padded->size < (unsigned)((2 * hLen) + 2)) || (padded->buffer[0] != 0))
295             ERROR_RETURN(TPM_RC_VALUE);
296     // Use the hash size to determine what to put through MGF1 in order
297     // to recover the seedMask
298         CryptMGF1(hLen, seedMask, hashAlg, padded->size - hLen - 1,
299                   &padded->buffer[hLen + 1]);
```

```
300
301         // Recover the seed into seedMask
302         pAssert(hLen <= sizeof(seedMask));
303         pp = &padded->buffer[1];
304         pm = seedMask;
305         for(i = hLen; i > 0; i--)
306             *pm++ ^= *pp++;
307
308         // Use the seed to generate the data mask
309         CryptMGF1(padded->size - hLen - 1, mask, hashAlg, hLen, seedMask);
310
311         // Use the mask generated from seed to recover the padded data
312         pp = &padded->buffer[hLen + 1];
313         pm = mask;
314         for(i = (padded->size - hLen - 1); i > 0; i--)
315             *pm++ ^= *pp++;
316
317         // Make sure that the recovered data has the hash of the label
318         // Put trial value in the seed mask
319         if((CryptHashBlock(hashAlg, label->size, (BYTE *)label->buffer,
320                            hLen, seedMask)) != hLen)
321             FAIL(FATAL_ERROR_INTERNAL);
322         if(memcmp(seedMask, mask, hLen) != 0)
323             ERROR_RETURN(TPM_RC_VALUE);
324
325         // find the start of the data
326         pm = &mask[hLen];
327         for(i = (UINT32)padded->size - (2 * hLen) - 1; i > 0; i--)
328         {
329             if(*pm++ != 0)
330                 break;
331         }
332         // If we ran out of data or didn't end with 0x01, then return an error
333         if(i == 0 || pm[-1] != 0x01)
334             ERROR_RETURN(TPM_RC_VALUE);
335
336         // pm should be pointing at the first part of the data
337         // and i is one greater than the number of bytes to move
338         i--;
339         if(i > dataOut->size)
340             // Special exit to preserve the size of the output buffer
341             return TPM_RC_VALUE;
342         memcpy(dataOut->buffer, pm, i);
343         dataOut->size = (UINT16)i;
344     Exit:
345         if(retVal != TPM_RC_SUCCESS)
346             dataOut->size = 0;
347         return retVal;
348     }
```

10.2.17.4.11 PKCS1v1_5Encode()

This function performs the encoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in PKCS#1V2.1

Error Returns	Meaning
TPM_RC_VALUE	message size is too large

```
349     static TPM_RC
350     RSAES_PKCS1v1_5Encode(
351         TPM2B         *padded,          // OUT: the pad data
352         TPM2B         *message,         // IN: the message being padded
353         RAND_STATE    *rand
354         )
```


 Copyright © TCG 2006-2020

```
355 {
356     UINT32      ps = padded->size - message->size - 3;
357 //
358     if(message->size > padded->size - 11)
359         return TPM_RC_VALUE;
360     // move the message to the end of the buffer
361     memcpy(&padded->buffer[padded->size - message->size], message->buffer,
362            message->size);
363     // Set the first byte to 0x00 and the second to 0x02
364     padded->buffer[0] = 0;
365     padded->buffer[1] = 2;
366
367     // Fill with random bytes
368     DRBG_Generate(rand, &padded->buffer[2], (UINT16)ps);
369     if(g_inFailureMode)
370         return TPM_RC_FAILURE;
371
372     // Set the delimiter for the random field to 0
373     padded->buffer[2 + ps] = 0;
374
375     // Now, the only messy part. Make sure that all the 'ps' bytes are non-zero
376     // In this implementation, use the value of the current index
377     for(ps++; ps > 1; ps--)
378     {
379         if(padded->buffer[ps] == 0)
380             padded->buffer[ps] = 0x55;  // In the < 0.5% of the cases that the
381                                         // random value is 0, just pick a value to
382                                         // put into the spot.
383     }
384     return TPM_RC_SUCCESS;
385 }
```

10.2.17.4.12 RSAES_Decode()

This function performs the decoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in PKCS#1V2.1

Error Returns	Meaning
TPM_RC_FAIL	decoding error or results would no fit into provided buffer

```
386 static TPM_RC
387 RSAES_Decode(
388     TPM2B       *message,       // OUT: the recovered message
389     TPM2B       *coded          // IN: the encoded message
390     )
391 {
392     BOOL        fail = FALSE;
393     UINT16      pSize;
394
395     fail = (coded->size < 11);
396     fail = (coded->buffer[0] != 0x00) | fail;
397     fail = (coded->buffer[1] != 0x02) | fail;
398     for(pSize = 2; pSize < coded->size; pSize++)
399     {
400         if(coded->buffer[pSize] == 0)
401             break;
402     }
403     pSize++;
404
405     // Make sure that pSize has not gone over the end and that there are at least 8
406     // bytes of pad data.
407     fail = (pSize > coded->size) | fail;
408     fail = ((pSize - 2) < 8) | fail;
409     if((message->size < (UINT16)(coded->size - pSize)) || fail)
```

 Copyright © TCG 2006-2020

```
410             return TPM_RC_VALUE;
411         message->size = coded->size - pSize;
412         memcpy(message->buffer, &coded->buffer[pSize], coded->size - pSize);
413         return TPM_RC_SUCCESS;
414     }
```

10.2.17.4.13 CryptRsaPssSaltSize()

This function computes the salt size used in PSS. It is broken out so that the X509 code can get the same value that is used by the encoding function in this module.

```
415     INT16
416     CryptRsaPssSaltSize(
417         INT16              hashSize,
418         INT16               outSize
419     )
420     {
421         INT16               saltSize;
422     //
423         // (Mask Length) = (outSize - hashSize - 1);
424         // Max saltSize is (Mask Length) - 1
425         saltSize = (outSize - hashSize - 1) - 1;
426         // Use the maximum salt size allowed by FIPS 186-4
427         if(saltSize > hashSize)
428             saltSize = hashSize;
429         else if(saltSize < 0)
430             saltSize = 0;
431         return saltSize;
432     }
```

10.2.17.4.14 PssEncode()

This function creates an encoded block of data that is the size of modulus. The function uses the maximum salt size that will fit in the encoded block.

Returns TPM_RC_SUCCESS or goes into failure mode.

```
433     static TPM_RC
434     PssEncode(
435         TPM2B           *out,       // OUT: the encoded buffer
436         TPM_ALG_ID       hashAlg,   // IN: hash algorithm for the encoding
437         TPM2B           *digest,    // IN: the digest
438         RAND_STATE      *rand       // IN: random number source
439         )
440     {
441         UINT32                hLen = CryptHashGetDigestSize(hashAlg);
442         BYTE                  salt[MAX_RSA_KEY_BYTES - 1];
443         UINT16                saltSize;
444         BYTE                 *ps = salt;
445         BYTE                 *pOut;
446         UINT16                mLen;
447         HASH_STATE            hashState;
448
449         // These are fatal errors indicating bad TPM firmware
450         pAssert(out != NULL && hLen > 0 && digest != NULL);
451
452         // Get the size of the mask
453         mLen = (UINT16)(out->size - hLen - 1);
454
455         // Set the salt size
456         saltSize = CryptRsaPssSaltSize((INT16)hLen, (INT16)out->size);
457
458     //using eOut for scratch space
```

 Copyright © TCG 2006-2020

```
459     // Set the first 8 bytes to zero
460     pOut = out->buffer;
461     memset(pOut, 0, 8);
462
463     // Get set the salt
464     DRBG_Generate(rand, salt, saltSize);
465     if(g_inFailureMode)
466         return TPM_RC_FAILURE;
467
468     // Create the hash of the pad || input hash || salt
469     CryptHashStart(&hashState, hashAlg);
470     CryptDigestUpdate(&hashState, 8, pOut);
471     CryptDigestUpdate2B(&hashState, digest);
472     CryptDigestUpdate(&hashState, saltSize, salt);
473     CryptHashEnd(&hashState, hLen, &pOut[out->size - hLen - 1]);
474
475     // Create a mask
476     if(CryptMGF1(mLen, pOut, hashAlg, hLen, &pOut[mLen]) != mLen)
477         FAIL(FATAL_ERROR_INTERNAL);
478
479     // Since this implementation uses key sizes that are all even multiples of
480     // 8, just need to make sure that the most significant bit is CLEAR
481     *pOut &= 0x7f;
482
483     // Before we mess up the pOut value, set the last byte to 0xbc
484     pOut[out->size - 1] = 0xbc;
485
486     // XOR a byte of 0x01 at the position just before where the salt will be XOR'ed
487     pOut = &pOut[mLen - saltSize - 1];
488     *pOut++ ^= 0x01;
489
490     // XOR the salt data into the buffer
491     for(; saltSize > 0; saltSize--)
492         *pOut++ ^= *ps++;
493
494     // and we are done
495     return TPM_RC_SUCCESS;
496 }
```

10.2.17.4.15 PssDecode()

This function checks that the PSS encoded block was built from the provided digest. If the check is successful, TPM_RC_SUCCESS is returned. Any other value indicates an error.

This implementation of PSS decoding is intended for the reference TPM implementation and is not at all generalized. It is used to check signatures over hashes and assumptions are made about the sizes of values. Those assumptions are enforce by this implementation. This implementation does allow for a variable size salt value to have been used by the creator of the signature.

Error Returns	Meaning
TPM_RC_SCHEME	*hashAlg* is not a supported hash algorithm
TPM_RC_VALUE	decode operation failed

```
497 static TPM_RC
498 PssDecode(
499     TPM_ALG_ID   hashAlg,        // IN: hash algorithm to use for the encoding
500     TPM2B       *dIn,            // In: the digest to compare
501     TPM2B       *eIn             // IN: the encoded data
502     )
503 {
504     UINT32           hLen = CryptHashGetDigestSize(hashAlg);
505     BYTE             mask[MAX_RSA_KEY_BYTES];
```

```
506         BYTE             *pm = mask;
507         BYTE             *pe;
508         BYTE              pad[8] = {0};
509         UINT32            i;
510         UINT32            mLen;
511         BYTE              fail;
512         TPM_RC            retVal = TPM_RC_SUCCESS;
513         HASH_STATE        hashState;
514
515         // These errors are indicative of failures due to programmer error
516         pAssert(dIn != NULL && eIn != NULL);
517         pe = eIn->buffer;
518
519         // check the hash scheme
520         if(hLen == 0)
521             ERROR_RETURN(TPM_RC_SCHEME);
522
523         // most significant bit must be zero
524         fail = pe[0] & 0x80;
525
526         // last byte must be 0xbc
527         fail |= pe[eIn->size - 1] ^ 0xbc;
528
529         // Use the hLen bytes at the end of the buffer to generate a mask
530         // Doesn't start at the end which is a flag byte
531         mLen = eIn->size - hLen - 1;
532         CryptMGF1(mLen, mask, hashAlg, hLen, &pe[mLen]);
533
534         // Clear the MSO of the mask to make it consistent with the encoding.
535         mask[0] &= 0x7F;
536
537         pAssert(mLen <= sizeof(mask));
538         // XOR the data into the mask to recover the salt. This sequence
539         // advances eIn so that it will end up pointing to the seed data
540         // which is the hash of the signature data
541         for(i = mLen; i > 0; i--)
542             *pm++ ^= *pe++;
543
544         // Find the first byte of 0x01 after a string of all 0x00
545         for(pm = mask, i = mLen; i > 0; i--)
546         {
547             if(*pm == 0x01)
548                 break;
549             else
550                 fail |= *pm++;
551         }
552         // i should not be zero
553         fail |= (i == 0);
554
555         // if we have failed, will continue using the entire mask as the salt value so
556         // that the timing attacks will not disclose anything (I don't think that this
557         // is a problem for TPM applications but, usually, we don't fail so this
558         // doesn't cost anything).
559         if(fail)
560         {
561             i = mLen;
562             pm = mask;
563         }
564         else
565         {
566             pm++;
567             i--;
568         }
569         // i contains the salt size and pm points to the salt. Going to use the input
570         // hash and the seed to recreate the hash in the lower portion of eIn.
571         CryptHashStart(&hashState, hashAlg);
```

```
572
573         // add the pad of 8 zeros
574         CryptDigestUpdate(&hashState, 8, pad);
575
576         // add the provided digest value
577         CryptDigestUpdate(&hashState, dIn->size, dIn->buffer);
578
579         // and the salt
580         CryptDigestUpdate(&hashState, i, pm);
581
582         // get the result
583         fail |= (CryptHashEnd(&hashState, hLen, mask) != hLen);
584
585         // Compare all bytes
586         for(pm = mask; hLen > 0; hLen--)
587             // don't use fail = because that could skip the increment and compare
588             // operations after the first failure and that gives away timing
589             // information.
590             fail |= *pm++ ^ *pe++;
591
592         retVal = (fail != 0) ? TPM_RC_VALUE : TPM_RC_SUCCESS;
593     Exit:
594             return retVal;
595     }
```

10.2.17.4.16 MakeDerTag()

Construct the DER value that is used in RSASSA

Return Value	Meaning
0	size of value
0	no hash exists

```
596     INT16
597     MakeDerTag(
598         TPM_ALG_ID    hashAlg,
599         INT16         sizeOfBuffer,
600         BYTE         *buffer
601     )
602     {
603     //    0x30, 0x31,       // SEQUENCE (2 elements) 1st
604     //        0x30, 0x0D,    // SEQUENCE (2 elements)
605     //            0x06, 0x09,    // HASH OID
606     //                0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01,
607     //             0x05, 0x00,   // NULL
608     //        0x04, 0x20   //  OCTET STRING
609         HASH_DEF    *info = CryptGetHashDef(hashAlg);
610         INT16       oidSize;
611         // If no OID, can't do encode
612         VERIFY(info != NULL);
613         oidSize = 2 + (info->OID)[1];
614         // make sure this fits in the buffer
615         VERIFY(sizeOfBuffer >= (oidSize + 8));
616         *buffer++ = 0x30;  // 1st SEQUENCE
617         // Size of the 1st SEQUENCE is 6 bytes + size of the hash OID + size of the
618         // digest size
619         *buffer++ = (BYTE)(6 + oidSize + info->digestSize);   //
620         *buffer++ = 0x30; // 2nd SEQUENCE
621         // size is 4 bytes of overhead plus the side of the OID
622         *buffer++ = (BYTE)(2 + oidSize);
623         MemoryCopy(buffer, info->OID, oidSize);
624         buffer += oidSize;
```

```
625         *buffer++ = 0x05;    // Add a NULL
626         *buffer++ = 0x00;
627
628         *buffer++ = 0x04;
629         *buffer++ = (BYTE)(info->digestSize);
630         return oidSize + 8;
631     Error:
632         return 0;
633
634     }
```

10.2.17.4.17 RSASSA_Encode()

Encode a message using PKCS1v1.5 method.

Error Returns	Meaning
TPM_RC_SCHEME	*hashAlg* is not a supported hash algorithm
TPM_RC_SIZE	*eOutSize* is not large enough
TPM_RC_VALUE	*hInSize* does not match the digest size of *hashAlg*

```
635     static TPM_RC
636     RSASSA_Encode(
637         TPM2B               *pOut,      // IN:OUT on in, the size of the public key
638                                         //         on out, the encoded area
639         TPM_ALG_ID           hashAlg,   // IN: hash algorithm for PKCS1v1_5
640         TPM2B               *hIn        // IN: digest value to encode
641         )
642     {
643         BYTE              DER[20];
644         BYTE             *der = DER;
645         INT32             derSize = MakeDerTag(hashAlg, sizeof(DER), DER);
646         BYTE             *eOut;
647         INT32             fillSize;
648         TPM_RC            retVal = TPM_RC_SUCCESS;
649
650         // Can't use this scheme if the algorithm doesn't have a DER string defined.
651         if(derSize == 0)
652             ERROR_RETURN(TPM_RC_SCHEME);
653
654         // If the digest size of 'hashAl' doesn't match the input digest size, then
655         // the DER will misidentify the digest so return an error
656         if(CryptHashGetDigestSize(hashAlg) != hIn->size)
657             ERROR_RETURN(TPM_RC_VALUE);
658         fillSize = pOut->size - derSize - hIn->size - 3;
659         eOut = pOut->buffer;
660
661         // Make sure that this combination will fit in the provided space
662         if(fillSize < 8)
663             ERROR_RETURN(TPM_RC_SIZE);
664
665         // Start filling
666         *eOut++ = 0; // initial byte of zero
667         *eOut++ = 1; // byte of 0x01
668         for(; fillSize > 0; fillSize--)
669             *eOut++ = 0xff; // bunch of 0xff
670         *eOut++ = 0; // another 0
671         for(; derSize > 0; derSize--)
672             *eOut++ = *der++;    // copy the DER
673         der = hIn->buffer;
674         for(fillSize = hIn->size; fillSize > 0; fillSize--)
675             *eOut++ = *der++;    // copy the hash
676     Exit:
```

```
677     return retVal;
678 }
```

10.2.17.4.18 RSASSA_Decode()

This function performs the RSASSA decoding of a signature.

Error Returns	Meaning
TPM_RC_VALUE	decode unsuccessful
TPM_RC_SCHEME	*hashAlg* is not supported

```
679 static TPM_RC
680 RSASSA_Decode(
681     TPM_ALG_ID       hashAlg,        // IN: hash algorithm to use for the encoding
682     TPM2B           *hIn,            // In: the digest to compare
683     TPM2B           *eIn             // IN: the encoded data
684     )
685 {
686     BYTE             fail;
687     BYTE             DER[20];
688     BYTE            *der = DER;
689     INT32            derSize = MakeDerTag(hashAlg, sizeof(DER), DER);
690     BYTE            *pe;
691     INT32            hashSize = CryptHashGetDigestSize(hashAlg);
692     INT32            fillSize;
693     TPM_RC           retVal;
694     BYTE            *digest;
695     UINT16           digestSize;
696
697     pAssert(hIn != NULL && eIn != NULL);
698     pe = eIn->buffer;
699
700     // Can't use this scheme if the algorithm doesn't have a DER string
701     // defined or if the provided hash isn't the right size
702     if(derSize == 0 || (unsigned)hashSize != hIn->size)
703         ERROR_RETURN(TPM_RC_SCHEME);
704
705     // Make sure that this combination will fit in the provided space
706     // Since no data movement takes place, can just walk though this
707     // and accept nearly random values. This can only be called from
708     // CryptValidateSignature() so eInSize is known to be in range.
709     fillSize = eIn->size - derSize - hashSize - 3;
710
711     // Start checking (fail will become non-zero if any of the bytes do not have
712     // the expected value.
713     fail = *pe++;                   // initial byte of zero
714     fail |= *pe++ ^ 1;              // byte of 0x01
715     for(; fillSize > 0; fillSize--)
716         fail |= *pe++ ^ 0xff;       // bunch of 0xff
717     fail |= *pe++;                  // another 0
718     for(; derSize > 0; derSize--)
719         fail |= *pe++ ^ *der++;    // match the DER
720     digestSize = hIn->size;
721     digest = hIn->buffer;
722     for(; digestSize > 0; digestSize--)
723         fail |= *pe++ ^ *digest++; // match the hash
724     retVal = (fail != 0) ? TPM_RC_VALUE : TPM_RC_SUCCESS;
725 Exit:
726     return retVal;
727 }
```

 Copyright © TCG 2006-2020

10.2.17.5 Externally Accessible Functions

10.2.17.5.1 CryptRsaSelectScheme()

This function is used by TPM2_RSA_Decrypt() and TPM2_RSA_Encrypt(). It sets up the rules to select a scheme between input and object default. This function assume the RSA object is loaded. If a default scheme is defined in object, the default scheme should be chosen, otherwise, the input scheme should be chosen. In the case that both the object and *scheme* are not TPM_ALG_NULL, then if the schemes are the same, the input scheme will be chosen. if the scheme are not compatible, a NULL pointer will be returned.

The return pointer may point to a TPM_ALG_NULL scheme.

```
728  TPMT_RSA_DECRYPT*
729  CryptRsaSelectScheme(
730      TPMI_DH_OBJECT       rsaHandle,     // IN: handle of an RSA key
731      TPMT_RSA_DECRYPT    *scheme         // IN: a sign or decrypt scheme
732      )
733  {
734      OBJECT              *rsaObject;
735      TPMT_ASYM_SCHEME    *keyScheme;
736      TPMT_RSA_DECRYPT    *retVal = NULL;
737
738      // Get sign object pointer
739      rsaObject = HandleToObject(rsaHandle);
740      keyScheme = &rsaObject->publicArea.parameters.asymDetail.scheme;
741
742      // if the default scheme of the object is TPM_ALG_NULL, then select the
743      // input scheme
744      if(keyScheme->scheme == TPM_ALG_NULL)
745      {
746          retVal = scheme;
747      }
748      // if the object scheme is not TPM_ALG_NULL and the input scheme is
749      // TPM_ALG_NULL, then select the default scheme of the object.
750      else if(scheme->scheme == TPM_ALG_NULL)
751      {
752          // if input scheme is NULL
753          retVal = (TPMT_RSA_DECRYPT *)keyScheme;
754      }
755      // get here if both the object scheme and the input scheme are
756      // not TPM_ALG_NULL. Need to insure that they are the same.
757      // IMPLEMENTATION NOTE: This could cause problems if future versions have
758      // schemes that have more values than just a hash algorithm. A new function
759      // (IsSchemeSame()) might be needed then.
760      else if(keyScheme->scheme == scheme->scheme
761              && keyScheme->details.anySig.hashAlg == scheme->details.anySig.hashAlg)
762      {
763          retVal = scheme;
764      }
765      // two different, incompatible schemes specified will return NULL
766      return retVal;
767  }
```

10.2.17.5.2 CryptRsaLoadPrivateExponent()

This function is called to generate the private exponent of an RSA key.

 Copyright © TCG 2006-2020

Error Returns	Meaning
TPM_RC_BINDING	public and private parts of *rsaKey* are not matched

```
768  TPM_RC
769  CryptRsaLoadPrivateExponent(
770      TPMT_PUBLIC            *publicArea,
771      TPMT_SENSITIVE         *sensitive
772      )
773  {
774  //
775      if((sensitive->sensitive.rsa.t.size & RSA_prime_flag) == 0)
776      {
777          if((sensitive->sensitive.rsa.t.size * 2) == publicArea->unique.rsa.t.size)
778          {
779              NEW_PRIVATE_EXPONENT(Z);
780              BN_RSA_INITIALIZED(bnN, &publicArea->unique.rsa);
781              BN_RSA(bnQr);
782              BN_VAR(bnE, RADIX_BITS);
783
784              TEST(ALG_NULL_VALUE);
785
786              VERIFY((sensitive->sensitive.rsa.t.size * 2)
787                     == publicArea->unique.rsa.t.size);
788              // Initialize the exponent
789              BnSetWord(bnE, publicArea->parameters.rsaDetail.exponent);
790              if(BnEqualZero(bnE))
791                  BnSetWord(bnE, RSA_DEFAULT_PUBLIC_EXPONENT);
792              // Convert first prime to 2B
793              VERIFY(BnFrom2B(Z->P, &sensitive->sensitive.rsa.b) != NULL);
794
795              // Find the second prime by division. This uses 'bQ' rather than Z->Q
796              // because the division could make the quotient larger than a prime during
797              // some intermediate step.
798              VERIFY(BnDiv(Z->Q, bnQr, bnN, Z->P));
799              VERIFY(BnEqualZero(bnQr));
800              // Compute the private exponent and return it if found
801              VERIFY(ComputePrivateExponent(bnE, Z));
802              VERIFY(PackExponent(&sensitive->sensitive.rsa, Z));
803          }
804          else
805              VERIFY(((sensitive->sensitive.rsa.t.size / 5) * 2)
806                     == publicArea->unique.rsa.t.size);
807          sensitive->sensitive.rsa.t.size |= RSA_prime_flag;
808      }
809      return TPM_RC_SUCCESS;
810  Error:
811      return TPM_RC_BINDING;
812  }
```

10.2.17.5.3 CryptRsaEncrypt()

This is the entry point for encryption using RSA. Encryption is use of the public exponent. The padding parameter determines what padding will be used.

The *cOutSize* parameter must be at least as large as the size of the key.

If the padding is RSA_PAD_NONE, *dIn* is treated as a number. It must be lower in value than the key modulus.

 Copyright © TCG 2006-2020

NOTE: If *dIn* has fewer bytes than *cOut*, then we don't add low-order zeros to *dIn* to make it the size of the RSA key for the call to RSAEP. This is because the high order bytes of *dIn* might have a numeric value that is greater than the value of the key modulus. If this had low-order zeros added, it would have a numeric value larger than the modulus even though it started out with a lower numeric value.

Error Returns	Meaning
TPM_RC_VALUE	*cOutSize* is too small (must be the size of the modulus)
TPM_RC_SCHEME	*padType* is not a supported scheme

```
813  LIB_EXPORT TPM_RC
814  CryptRsaEncrypt(
815      TPM2B_PUBLIC_KEY_RSA        *cOut,          // OUT: the encrypted data
816      TPM2B                       *dIn,           // IN: the data to encrypt
817      OBJECT                      *key,           // IN: the key used for encryption
818      TPMT_RSA_DECRYPT            *scheme,        // IN: the type of padding and hash
819                                                  //     if needed
820      const TPM2B                 *label,         // IN: in case it is needed
821      RAND_STATE                  *rand           // IN: random number generator
822                                                  //     state (mostly for testing)
823      )
824  {
825      TPM_RC                       retVal = TPM_RC_SUCCESS;
826      TPM2B_PUBLIC_KEY_RSA         dataIn;
827  //
828      // if the input and output buffers are the same, copy the input to a scratch
829      // buffer so that things don't get messed up.
830      if(dIn == &cOut->b)
831      {
832          MemoryCopy2B(&dataIn.b, dIn, sizeof(dataIn.t.buffer));
833          dIn = &dataIn.b;
834      }
835      // All encryption schemes return the same size of data
836      cOut->t.size = key->publicArea.unique.rsa.t.size;
837      TEST(scheme->scheme);
838
839      switch(scheme->scheme)
840      {
841          case ALG_NULL_VALUE:  // 'raw' encryption
842          {
843              INT32            i;
844              INT32            dSize = dIn->size;
845              // dIn can have more bytes than cOut as long as the extra bytes
846              // are zero. Note: the more significant bytes of a number in a byte
847              // buffer are the bytes at the start of the array.
848              for(i = 0; (i < dSize) && (dIn->buffer[i] == 0); i++);
849              dSize -= i;
850              if(dSize > cOut->t.size)
851                  ERROR_RETURN(TPM_RC_VALUE);
852              // Pad cOut with zeros if dIn is smaller
853              memset(cOut->t.buffer, 0, cOut->t.size - dSize);
854              // And copy the rest of the value
855              memcpy(&cOut->t.buffer[cOut->t.size - dSize], &dIn->buffer[i], dSize);
856
857              // If the size of dIn is the same as cOut dIn could be larger than
858              // the modulus. If it is, then RSAEP() will catch it.
859          }
860          break;
861          case ALG_RSAES_VALUE:
862              retVal = RSAES_PKCS1v1_5Encode(&cOut->b, dIn, rand);
863              break;
864          case ALG_OAEP_VALUE:
865              retVal = OaepEncode(&cOut->b, scheme->details.oaep.hashAlg, label, dIn,
866                                  rand);
```

 Copyright © TCG 2006-2020

```
867             break;
868         default:
869             ERROR_RETURN(TPM_RC_SCHEME);
870             break;
871     }
872     // All the schemes that do padding will come here for the encryption step
873     // Check that the Encoding worked
874     if(retVal == TPM_RC_SUCCESS)
875         // Padding OK so do the encryption
876         retVal = RSAEP(&cOut->b, key);
877 Exit:
878     return retVal;
879 }
```

10.2.17.5.4 CryptRsaDecrypt()

This is the entry point for decryption using RSA. Decryption is use of the private exponent. The *padType* parameter determines what padding was used.

Error Returns	Meaning
TPM_RC_SIZE	*cInSize* is not the same as the size of the public modulus of *key*; or numeric value of the encrypted data is greater than the modulus
TPM_RC_VALUE	*dOutSize* is not large enough for the result
TPM_RC_SCHEME	*padType* is not supported

```
880 LIB_EXPORT TPM_RC
881 CryptRsaDecrypt(
882     TPM2B               *dOut,          // OUT: the decrypted data
883     TPM2B               *cIn,           // IN: the data to decrypt
884     OBJECT              *key,           // IN: the key to use for decryption
885     TPMT_RSA_DECRYPT    *scheme,        // IN: the padding scheme
886     const TPM2B         *label          // IN: in case it is needed for the scheme
887     )
888 {
889     TPM_RC                  retVal;
890
891     // Make sure that the necessary parameters are provided
892     pAssert(cIn != NULL && dOut != NULL && key != NULL);
893
894     // Size is checked to make sure that the encrypted value is the right size
895     if(cIn->size != key->publicArea.unique.rsa.t.size)
896         ERROR_RETURN(TPM_RC_SIZE);
897
898     TEST(scheme->scheme);
899
900     // For others that do padding, do the decryption in place and then
901     // go handle the decoding.
902     retVal = RSADP(cIn, key);
903     if(retVal == TPM_RC_SUCCESS)
904     {
905         // Remove padding
906         switch(scheme->scheme)
907         {
908             case ALG_NULL_VALUE:
909                 if(dOut->size < cIn->size)
910                     return TPM_RC_VALUE;
911                 MemoryCopy2B(dOut, cIn, dOut->size);
912                 break;
913             case ALG_RSAES_VALUE:
914                 retVal = RSAES_Decode(dOut, cIn);
915                 break;
916             case ALG_OAEP_VALUE:
```

```
917                 retVal = OaepDecode(dOut, scheme->details.oaep.hashAlg, label, cIn);
918                 break;
919             default:
920                 retVal = TPM_RC_SCHEME;
921                 break;
922         }
923     }
924 Exit:
925     return retVal;
926 }
```

10.2.17.5.5 CryptRsaSign()

This function is used to generate an RSA signature of the type indicated in *scheme*.

Error Returns	Meaning
TPM_RC_SCHEME	*scheme* or *hashAlg* are not supported
TPM_RC_VALUE	*hInSize* does not match *hashAlg* (for RSASSA)

```
927 LIB_EXPORT TPM_RC
928 CryptRsaSign(
929     TPMT_SIGNATURE      *sigOut,
930     OBJECT              *key,           // IN: key to use
931     TPM2B_DIGEST        *hIn,           // IN: the digest to sign
932     RAND_STATE          *rand           // IN: the random number generator
933                                         //      to use (mostly for testing)
934     )
935 {
936     TPM_RC                retVal = TPM_RC_SUCCESS;
937     UINT16                modSize;
938
939     // parameter checks
940     pAssert(sigOut != NULL && key != NULL && hIn != NULL);
941
942     modSize = key->publicArea.unique.rsa.t.size;
943
944     // for all non-null signatures, the size is the size of the key modulus
945     sigOut->signature.rsapss.sig.t.size = modSize;
946
947     TEST(sigOut->sigAlg);
948
949     switch(sigOut->sigAlg)
950     {
951         case ALG_NULL_VALUE:
952             sigOut->signature.rsapss.sig.t.size = 0;
953             return TPM_RC_SUCCESS;
954         case ALG_RSAPSS_VALUE:
955             retVal = PssEncode(&sigOut->signature.rsapss.sig.b,
956                                sigOut->signature.rsapss.hash, &hIn->b, rand);
957             break;
958         case ALG_RSASSA_VALUE:
959             retVal = RSASSA_Encode(&sigOut->signature.rsassa.sig.b,
960                                    sigOut->signature.rsassa.hash, &hIn->b);
961             break;
962         default:
963             retVal = TPM_RC_SCHEME;
964     }
965     if(retVal == TPM_RC_SUCCESS)
966     {
967         // Do the encryption using the private key
968         retVal = RSADP(&sigOut->signature.rsapss.sig.b, key);
969     }
```

```
970         return retVal;
971     }
```

10.2.17.5.6 CryptRsaValidateSignature()

This function is used to validate an RSA signature. If the signature is valid TPM_RC_SUCCESS is returned. If the signature is not valid, TPM_RC_SIGNATURE is returned. Other return codes indicate either parameter problems or fatal errors.

Error Returns	Meaning
TPM_RC_SIGNATURE	the signature does not check
TPM_RC_SCHEME	unsupported scheme or hash algorithm

```
972     LIB_EXPORT TPM_RC
973     CryptRsaValidateSignature(
974         TPMT_SIGNATURE   *sig,            // IN: signature
975         OBJECT           *key,            // IN: public modulus
976         TPM2B_DIGEST     *digest          // IN: The digest being validated
977         )
978     {
979         TPM_RC           retVal;
980     //
981         // Fatal programming errors
982         pAssert(key != NULL && sig != NULL && digest != NULL);
983         switch(sig->sigAlg)
984         {
985             case ALG_RSAPSS_VALUE:
986             case ALG_RSASSA_VALUE:
987                 break;
988             default:
989                 return TPM_RC_SCHEME;
990         }
991
992         // Errors that might be caused by calling parameters
993         if(sig->signature.rsassa.sig.t.size != key->publicArea.unique.rsa.t.size)
994             ERROR_RETURN(TPM_RC_SIGNATURE);
995
996         TEST(sig->sigAlg);
997
998         // Decrypt the block
999         retVal = RSAEP(&sig->signature.rsassa.sig.b, key);
1000        if(retVal == TPM_RC_SUCCESS)
1001        {
1002            switch(sig->sigAlg)
1003            {
1004                case ALG_RSAPSS_VALUE:
1005                    retVal = PssDecode(sig->signature.any.hashAlg, &digest->b,
1006                                       &sig->signature.rsassa.sig.b);
1007                    break;
1008                case ALG_RSASSA_VALUE:
1009                    retVal = RSASSA_Decode(sig->signature.any.hashAlg, &digest->b,
1010                                           &sig->signature.rsassa.sig.b);
1011                    break;
1012                default:
1013                    return TPM_RC_SCHEME;
1014            }
1015        }
1016    Exit:
1017        return (retVal != TPM_RC_SUCCESS) ? TPM_RC_SIGNATURE : TPM_RC_SUCCESS;
1018    }
1019    #if SIMULATION && USE_RSA_KEY_CACHE
1020    extern int s_rsaKeyCacheEnabled;
```

```
1021 int GetCachedRsaKey(TPMT_PUBLIC *publicArea, TPMT_SENSITIVE *sensitive,
1022                     RAND_STATE *rand);
1023 #define GET_CACHED_KEY(publicArea, sensitive, rand)                      \
1024             (s_rsaKeyCacheEnabled && GetCachedRsaKey(publicArea, sensitive, rand))
1025 #else
1026 #define GET_CACHED_KEY(key, rand)
1027 #endif
```

10.2.17.5.7 CryptRsaGenerateKey()

Generate an RSA key from a provided seed

Error Returns	Meaning
TPM_RC_CANCELED	operation was canceled
TPM_RC_RANGE	public exponent is not supported
TPM_RC_VALUE	could not find a prime using the provided parameters

```
1028 LIB_EXPORT TPM_RC
1029 CryptRsaGenerateKey(
1030     TPMT_PUBLIC          *publicArea,
1031     TPMT_SENSITIVE       *sensitive,
1032     RAND_STATE           *rand               // IN: if not NULL, the deterministic
1033                                              //     RNG state
1034     )
1035 {
1036     UINT32                i;
1037     BN_RSA(bnD);
1038     BN_RSA(bnN);
1039     BN_WORD(bnPubExp);
1040     UINT32                e = publicArea->parameters.rsaDetail.exponent;
1041     int                   keySizeInBits;
1042     TPM_RC                retVal = TPM_RC_NO_RESULT;
1043     NEW_PRIVATE_EXPONENT(Z);
1044 //
1045
1046 // Need to make sure that the caller did not specify an exponent that is
1047 // not supported
1048     e = publicArea->parameters.rsaDetail.exponent;
1049     if(e == 0)
1050         e = RSA_DEFAULT_PUBLIC_EXPONENT;
1051     else
1052     {
1053         if(e < 65537)
1054             ERROR_RETURN(TPM_RC_RANGE);
1055         // Check that e is prime
1056         if(!IsPrimeInt(e))
1057             ERROR_RETURN(TPM_RC_RANGE);
1058     }
1059     BnSetWord(bnPubExp, e);
1060
1061     // check for supported key size.
1062     keySizeInBits = publicArea->parameters.rsaDetail.keyBits;
1063     if(((keySizeInBits % 1024) != 0)
1064        || (keySizeInBits > MAX_RSA_KEY_BITS)  // this might be redundant, but...
1065        || (keySizeInBits == 0))
1066         ERROR_RETURN(TPM_RC_VALUE);
1067
1068     // Set the prime size for instrumentation purposes
1069     INSTRUMENT_SET(PrimeIndex, PRIME_INDEX(keySizeInBits / 2));
1070
1071 #if SIMULATION && USE_RSA_KEY_CACHE
1072     if(GET_CACHED_KEY(publicArea, sensitive, rand))
```

```
1073            return TPM_RC_SUCCESS;
1074    #endif
1075
1076        // Make sure that key generation has been tested
1077        TEST(ALG_NULL_VALUE);
1078
1079        // The prime is computed in P. When a new prime is found, Q is checked to
1080        // see if it is zero.  If so, P is copied to Q and a new P is found.
1081        // When both P and Q are non-zero, the modulus and
1082        // private exponent are computed and a trial encryption/decryption is
1083        // performed.  If the encrypt/decrypt fails, assume that at least one of the
1084        // primes is composite. Since we don't know which one, set Q to zero and start
1085        // over and find a new pair of primes.
1086
1087        for(i = 1; (retVal == TPM_RC_NO_RESULT) && (i != 100); i++)
1088        {
1089            if(_plat__IsCanceled())
1090                ERROR_RETURN(TPM_RC_CANCELED);
1091
1092            if(BnGeneratePrimeForRSA(Z->P, keySizeInBits / 2, e, rand) == TPM_RC_FAILURE)
1093            {
1094                retVal = TPM_RC_FAILURE;
1095                goto Exit;
1096            }
1097
1098            INSTRUMENT_INC(PrimeCounts[PrimeIndex]);
1099
1100            // If this is the second prime, make sure that it differs from the
1101            // first prime by at least 2^100
1102            if(BnEqualZero(Z->Q))
1103            {
1104                // copy p to q and compute another prime in p
1105                BnCopy(Z->Q, Z->P);
1106                continue;
1107            }
1108            // Make sure that the difference is at least 100 bits. Need to do it this
1109            // way because the big numbers are only positive values
1110            if(BnUnsignedCmp(Z->P, Z->Q) < 0)
1111                BnSub(bnD, Z->Q, Z->P);
1112            else
1113                BnSub(bnD, Z->P, Z->Q);
1114            if(BnMsb(bnD) < 100)
1115                continue;
1116
1117            //Form the public modulus and set the unique value
1118            BnMult(bnN, Z->P, Z->Q);
1119            BnTo2B(bnN, &publicArea->unique.rsa.b,
1120                   (NUMBYTES)BITS_TO_BYTES(keySizeInBits));
1121            // Make sure everything came out right. The MSb of the values must be one
1122            if(((publicArea->unique.rsa.t.buffer[0] & 0x80) == 0)
1123                || (publicArea->unique.rsa.t.size
1124                    != (NUMBYTES)BITS_TO_BYTES(keySizeInBits)))
1125                FAIL(FATAL_ERROR_INTERNAL);
1126
1127            // Make sure that we can form the private exponent values
1128            if(ComputePrivateExponent(bnPubExp, Z) != TRUE)
1129            {
1130                // If ComputePrivateExponent could not find an inverse for
1131                // Q, then copy P and recompute P. This might
1132                // cause both to be recomputed if P is also zero
1133                if(BnEqualZero(Z->Q))
1134                    BnCopy(Z->Q, Z->P);
1135                continue;
1136            }
1137
1138            // Pack the private exponent into the sensitive area
```

```
1139            PackExponent(&sensitive->sensitive.rsa, Z);
1140            // Make sure everything came out right. The MSb of the values must be one
1141            if(((publicArea->unique.rsa.t.buffer[0] & 0x80) == 0)
1142               || ((sensitive->sensitive.rsa.t.buffer[0] & 0x80) == 0))
1143                FAIL(FATAL_ERROR_INTERNAL);
1144
1145            retVal = TPM_RC_SUCCESS;
1146            // Do a trial encryption decryption if this is a signing key
1147            if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
1148            {
1149                BN_RSA(temp1);
1150                BN_RSA(temp2);
1151                BnGenerateRandomInRange(temp1, bnN, rand);
1152
1153                // Encrypt with public exponent...
1154                BnModExp(temp2, temp1, bnPubExp, bnN);
1155                // ...  then decrypt with private exponent
1156                RsaPrivateKeyOp(temp2, Z);
1157
1158                // If the starting and ending values are not the same,
1159                // start over )-;
1160                if(BnUnsignedCmp(temp2, temp1) != 0)
1161                {
1162                    BnSetWord(Z->Q, 0);
1163                    retVal = TPM_RC_NO_RESULT;
1164                }
1165            }
1166        }
1167    Exit:
1168        return retVal;
1169    }
1170    #endif // ALG_RSA
```

10.2.18 CryptSmac.c

10.2.18.1 Introduction

This file contains the implementation of the message authentication codes based on a symmetric block cipher. These functions only use the single block encryption functions of the selected symmetric cryptographic library.

10.2.18.2 Includes, Defines, and Typedefs

```
1   #define _CRYPT_HASH_C_
2   #include "Tpm.h"
3   #if SMAC_IMPLEMENTED
```

10.2.18.2.1 CryptSmacStart()

Function to start an SMAC.

```
4   UINT16
5   CryptSmacStart(
6       HASH_STATE              *state,
7       TPMU_PUBLIC_PARMS       *keyParameters,
8       TPM_ALG_ID               macAlg,          // IN: the type of MAC
9       TPM2B                   *key
10  )
11  {
12      UINT16                  retVal = 0;
13  //
14      // Make sure that the key size is correct. This should have been checked
15      // at key load, but...
16      if(BITS_TO_BYTES(keyParameters->symDetail.sym.keyBits.sym) == key->size)
17      {
18          switch(macAlg)
19          {
20  #if ALG_CMAC
21              case ALG_CMAC_VALUE:
22                  retVal = CryptCmacStart(&state->state.smac, keyParameters,
23                                          macAlg, key);
24                  break;
25  #endif
26              default:
27                  break;
28          }
29      }
30      state->type = (retVal != 0) ? HASH_STATE_SMAC : HASH_STATE_EMPTY;
31      return retVal;
32  }
```

10.2.18.2.2 CryptMacStart()

Function to start either an HMAC or an SMAC. Cannot reuse the CryptHmacStart() function because of the difference in number of parameters.

```
33  UINT16
34  CryptMacStart(
35      HMAC_STATE              *state,
36      TPMU_PUBLIC_PARMS       *keyParameters,
37      TPM_ALG_ID               macAlg,          // IN: the type of MAC
38      TPM2B                   *key
```

```
39  )
40  {
41      MemorySet(state, 0, sizeof(HMAC_STATE));
42      if(CryptHashIsValidAlg(macAlg, FALSE))
43      {
44          return CryptHmacStart(state, macAlg, key->size, key->buffer);
45      }
46      else if(CryptSmacIsValidAlg(macAlg, FALSE))
47      {
48          return CryptSmacStart(&state->hashState, keyParameters, macAlg, key);
49      }
50      else
51          return 0;
52  }
```

10.2.18.2.3 CryptMacEnd()

Dispatch to the MAC end function using a size and buffer pointer.

```
53  UINT16
54  CryptMacEnd(
55      HMAC_STATE          *state,
56      UINT32               size,
57      BYTE                *buffer
58  )
59  {
60      UINT16              retVal = 0;
61      if(state->hashState.type == HASH_STATE_SMAC)
62          retVal = (state->hashState.state.smac.smacMethods.end)(
63                      &state->hashState.state.smac.state, size, buffer);
64      else if(state->hashState.type == HASH_STATE_HMAC)
65          retVal = CryptHmacEnd(state, size, buffer);
66      state->hashState.type = HASH_STATE_EMPTY;
67      return retVal;
68  }
```

10.2.18.2.4 CryptMacEnd2B()

Dispatch to the MAC end function using a 2B.

```
69  UINT16
70  CryptMacEnd2B (
71      HMAC_STATE          *state,
72      TPM2B               *data
73  )
74  {
75      return CryptMacEnd(state, data->size, data->buffer);
76  }
77  #endif // SMAC_IMPLEMENTED
```

 Copyright © TCG 2006-2020

10.2.19 CryptSym.c

10.2.19.1 Introduction

This file contains the implementation of the symmetric block cipher modes allowed for a TPM. These functions only use the single block encryption functions of the selected symmetric crypto library.

10.2.19.2 Includes, Defines, and Typedefs

```
1   #include "Tpm.h"
2   #include "CryptSym.h"
3   #define     KEY_BLOCK_SIZES(ALG, alg)                                              \
4   static const INT16       alg##KeyBlockSizes[] = {                               \
5                                  ALG##_KEY_SIZES_BITS, -1, ALG##_BLOCK_SIZES };
6   #if ALG_AES
7       KEY_BLOCK_SIZES(AES, aes);
8   #endif // ALG_AES
9   #if ALG_SM4
10      KEY_BLOCK_SIZES(SM4, sm4);
11  #endif
12  #if ALG_CAMELLIA
13      KEY_BLOCK_SIZES(CAMELLIA, camellia);
14  #endif
15  #if ALG_TDES
16      KEY_BLOCK_SIZES(TDES, tdes);
17  #endif
```

10.2.19.3 Initialization and Data Access Functions

10.2.19.3.1 CryptSymInit()

This function is called to do _TPM_Init() processing

```
18  BOOL
19  CryptSymInit(
20      void
21      )
22  {
23      return TRUE;
24  }
```

10.2.19.3.2 CryptSymStartup()

This function is called to do TPM2_Startup() processing

```
25  BOOL
26  CryptSymStartup(
27      void
28      )
29  {
30      return TRUE;
31  }
```

10.2.19.3.3 CryptGetSymmetricBlockSize()

This function returns the block size of the algorithm. The table of bit sizes has an entry for each allowed key size. The entry for a key size is 0 if the TPM does not implement that key size. The key size table is

delimited with a negative number (-1). After the delimiter is a list of block sizes with each entry corresponding to the key bit size. For most symmetric algorithms, the block size is the same regardless of the key size but this arrangement allows them to be different.

Return Value	Meaning
0	cipher not supported
0	the cipher block size in bytes

```
32  LIB_EXPORT INT16
33  CryptGetSymmetricBlockSize(
34      TPM_ALG_ID      symmetricAlg,   // IN: the symmetric algorithm
35      UINT16          keySizeInBits   // IN: the key size
36      )
37  {
38      const INT16    *sizes;
39      INT16           i;
40  #define ALG_CASE(SYM, sym)  case ALG_##SYM##_VALUE: sizes = sym##KeyBlockSizes; break
41      switch(symmetricAlg)
42      {
43  #if ALG_AES
44          ALG_CASE(AES, aes);
45  #endif
46  #if ALG_SM4
47          ALG_CASE(SM4, sm4);
48  #endif
49  #if ALG_CAMELLIA
50          ALG_CASE(CAMELLIA, camellia);
51  #endif
52  #if ALG_TDES
53          ALG_CASE(TDES, tdes);
54  #endif
55          default:
56              return 0;
57      }
58      // Find the index of the indicated keySizeInBits
59      for(i = 0; *sizes >= 0; i++, sizes++)
60      {
61          if(*sizes == keySizeInBits)
62              break;
63      }
64      // If sizes is pointing at the end of the list of key sizes, then the desired
65      // key size was not found so set the block size to zero.
66      if(*sizes++ < 0)
67          return 0;
68      // Advance until the end of the list is found
69      while(*sizes++ >= 0);
70      // sizes is pointing to the first entry in the list of block sizes. Use the
71      // ith index to find the block size for the corresponding key size.
72      return sizes[i];
73  }
```

10.2.19.4 Symmetric Encryption

This function performs symmetric encryption based on the mode.

 Copyright © TCG 2006-2020

Error Returns	Meaning
TPM_RC_SIZE	*dSize* is not a multiple of the block size for an algorithm that requires it
TPM_RC_FAILURE	Fatal error

```
74   LIB_EXPORT TPM_RC
75   CryptSymmetricEncrypt(
76       BYTE                 *dOut,          // OUT:
77       TPM_ALG_ID            algorithm,     // IN: the symmetric algorithm
78       UINT16                keySizeInBits, // IN: key size in bits
79       const BYTE           *key,           // IN: key buffer. The size of this buffer
80                                            //     in bytes is (keySizeInBits + 7) / 8
81       TPM2B_IV             *ivInOut,       // IN/OUT: IV for decryption.
82       TPM_ALG_ID            mode,          // IN: Mode to use
83       INT32                 dSize,         // IN: data size (may need to be a
84                                            //     multiple of the blockSize)
85       const BYTE           *dIn            // IN: data buffer
86       )
87   {
88       BYTE                 *pIv;
89       int                   i;
90       BYTE                  tmp[MAX_SYM_BLOCK_SIZE];
91       BYTE                 *pT;
92       tpmCryptKeySchedule_t        keySchedule;
93       INT16                 blockSize;
94       TpmCryptSetSymKeyCall_t        encrypt;
95       BYTE                 *iv;
96       BYTE                  defaultIv[MAX_SYM_BLOCK_SIZE] = {0};
97   //
98       pAssert(dOut != NULL && key != NULL && dIn != NULL);
99       if(dSize == 0)
100          return TPM_RC_SUCCESS;
101
102      TEST(algorithm);
103      blockSize = CryptGetSymmetricBlockSize(algorithm, keySizeInBits);
104      if(blockSize == 0)
105          return TPM_RC_FAILURE;
106      // If the iv is provided, then it is expected to be block sized. In some cases,
107      // the caller is providing an array of 0's that is equal to [MAX_SYM_BLOCK_SIZE]
108      // with no knowledge of the actual block size. This function will set it.
109      if((ivInOut != NULL) && (mode != ALG_ECB_VALUE))
110      {
111          ivInOut->t.size = blockSize;
112          iv = ivInOut->t.buffer;
113      }
114      else
115          iv = defaultIv;
116      pIv = iv;
117
118      // Create encrypt key schedule and set the encryption function pointer.
119
120      SELECT(ENCRYPT);
121
122      switch(mode)
123      {
124  #if ALG_CTR
125          case ALG_CTR_VALUE:
126              for(; dSize > 0; dSize -= blockSize)
127              {
128                  // Encrypt the current value of the IV(counter)
129                  ENCRYPT(&keySchedule, iv, tmp);
130
131                  //increment the counter (counter is big-endian so start at end)
```

```
132                 for(i = blockSize - 1; i >= 0; i--)
133                     if((iv[i] += 1) != 0)
134                         break;
135                 // XOR the encrypted counter value with input and put into output
136                 pT = tmp;
137                 for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
138                     *dOut++ = *dIn++ ^ *pT++;
139             }
140             break;
141 #endif
142 #if ALG_OFB
143         case ALG_OFB_VALUE:
144             // This is written so that dIn and dOut may be the same
145             for(; dSize > 0; dSize -= blockSize)
146             {
147                 // Encrypt the current value of the "IV"
148                 ENCRYPT(&keySchedule, iv, iv);
149
150                 // XOR the encrypted IV into dIn to create the cipher text (dOut)
151                 pIv = iv;
152                 for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
153                     *dOut++ = (*pIv++ ^ *dIn++);
154             }
155             break;
156 #endif
157 #if ALG_CBC
158         case ALG_CBC_VALUE:
159             // For CBC the data size must be an even multiple of the
160             // cipher block size
161             if((dSize % blockSize) != 0)
162                 return TPM_RC_SIZE;
163             // XOR the data block into the IV, encrypt the IV into the IV
164             // and then copy the IV to the output
165             for(; dSize > 0; dSize -= blockSize)
166             {
167                 pIv = iv;
168                 for(i = blockSize; i > 0; i--)
169                     *pIv++ ^= *dIn++;
170                 ENCRYPT(&keySchedule, iv, iv);
171                 pIv = iv;
172                 for(i = blockSize; i > 0; i--)
173                     *dOut++ = *pIv++;
174             }
175             break;
176 #endif
177         // CFB is not optional
178         case ALG_CFB_VALUE:
179             // Encrypt the IV into the IV, XOR in the data, and copy to output
180             for(; dSize > 0; dSize -= blockSize)
181             {
182                 // Encrypt the current value of the IV
183                 ENCRYPT(&keySchedule, iv, iv);
184                 pIv = iv;
185                 for(i = (int)(dSize < blockSize) ? dSize : blockSize; i > 0; i--)
186                     // XOR the data into the IV to create the cipher text
187                     // and put into the output
188                     *dOut++ = *pIv++ ^= *dIn++;
189             }
190             // If the inner loop (i loop) was smaller than blockSize, then dSize
191             // would have been smaller than blockSize and it is now negative. If
192             // it is negative, then it indicates how many bytes are needed to pad
193             // out the IV for the next round.
194             for(; dSize < 0; dSize++)
195                 *pIv++ = 0;
196             break;
197 #if ALG_ECB
```

 Copyright © TCG 2006-2020

```
198         case ALG_ECB_VALUE:
199             // For ECB the data size must be an even multiple of the
200             // cipher block size
201             if((dSize % blockSize) != 0)
202                 return TPM_RC_SIZE;
203             // Encrypt the input block to the output block
204             for(; dSize > 0; dSize -= blockSize)
205             {
206                 ENCRYPT(&keySchedule, dIn, dOut);
207                 dIn = &dIn[blockSize];
208                 dOut = &dOut[blockSize];
209             }
210             break;
211 #endif
212         default:
213             return TPM_RC_FAILURE;
214     }
215     return TPM_RC_SUCCESS;
216 }
```

10.2.19.4.1 CryptSymmetricDecrypt()

This function performs symmetric decryption based on the mode.

Error Returns	Meaning
TPM_RC_FAILURE	A fatal error
TPM_RCS_SIZE	*dSize* is not a multiple of the block size for an algorithm that requires it

```
217 LIB_EXPORT TPM_RC
218 CryptSymmetricDecrypt(
219     BYTE                *dOut,           // OUT: decrypted data
220     TPM_ALG_ID           algorithm,      // IN: the symmetric algorithm
221     UINT16               keySizeInBits, // IN: key size in bits
222     const BYTE          *key,            // IN: key buffer. The size of this buffer
223                                          //     in bytes is (keySizeInBits + 7) / 8
224     TPM2B_IV            *ivInOut,        // IN/OUT: IV for decryption.
225     TPM_ALG_ID           mode,           // IN: Mode to use
226     INT32                dSize,          // IN: data size (may need to be a
227                                          //     multiple of the blockSize)
228     const BYTE          *dIn             // IN: data buffer
229     )
230 {
231     BYTE                *pIv;
232     int                  i;
233     BYTE                 tmp[MAX_SYM_BLOCK_SIZE];
234     BYTE                *pT;
235     tpmCryptKeySchedule_t        keySchedule;
236     INT16                blockSize;
237     BYTE                *iv;
238     TpmCryptSetSymKeyCall_t        encrypt;
239     TpmCryptSetSymKeyCall_t        decrypt;
240     BYTE                 defaultIv[MAX_SYM_BLOCK_SIZE] = {0};
241
242     // These are used but the compiler can't tell because they are initialized
243     // in case statements and it can't tell if they are always initialized
244     // when needed, so... Comment these out if the compiler can tell or doesn't
245     // care that these are initialized before use.
246     encrypt = NULL;
247     decrypt = NULL;
248
249     pAssert(dOut != NULL && key != NULL && dIn != NULL);
```

```
250         if(dSize == 0)
251             return TPM_RC_SUCCESS;
252
253         TEST(algorithm);
254         blockSize = CryptGetSymmetricBlockSize(algorithm, keySizeInBits);
255         if(blockSize == 0)
256             return TPM_RC_FAILURE;
257         // If the iv is provided, then it is expected to be block sized. In some cases,
258         // the caller is providing an array of 0's that is equal to [MAX_SYM_BLOCK_SIZE]
259         // with no knowledge of the actual block size. This function will set it.
260         if((ivInOut != NULL) && (mode != ALG_ECB_VALUE))
261         {
262             ivInOut->t.size = blockSize;
263             iv = ivInOut->t.buffer;
264         }
265         else
266             iv = defaultIv;
267
268         pIv = iv;
269         // Use the mode to select the key schedule to create. Encrypt always uses the
270         // encryption schedule. Depending on the mode, decryption might use either
271         // the decryption or encryption schedule.
272         switch(mode)
273         {
274     #if ALG_CBC || ALG_ECB
275             case ALG_CBC_VALUE: // decrypt = decrypt
276             case ALG_ECB_VALUE:
277                 // For ECB and CBC, the data size must be an even multiple of the
278                 // cipher block size
279                 if((dSize % blockSize) != 0)
280                     return TPM_RC_SIZE;
281                 SELECT(DECRYPT);
282                 break;
283     #endif
284             default:
285                 // For the remaining stream ciphers, use encryption to decrypt
286                 SELECT(ENCRYPT);
287                 break;
288         }
289         // Now do the mode-dependent decryption
290         switch(mode)
291         {
292     #if ALG_CBC
293             case ALG_CBC_VALUE:
294                 // Copy the input data to a temp buffer, decrypt the buffer into the
295                 // output, XOR in the IV, and copy the temp buffer to the IV and repeat.
296                 for(; dSize > 0; dSize -= blockSize)
297                 {
298                     pT = tmp;
299                     for(i = blockSize; i > 0; i--)
300                         *pT++ = *dIn++;
301                     DECRYPT(&keySchedule, tmp, dOut);
302                     pIv = iv;
303                     pT = tmp;
304                     for(i = blockSize; i > 0; i--)
305                     {
306                         *dOut++ ^= *pIv;
307                         *pIv++ = *pT++;
308                     }
309                 }
310                 break;
311     #endif
312             case ALG_CFB_VALUE:
313                 for(; dSize > 0; dSize -= blockSize)
314                 {
315                     // Encrypt the IV into the temp buffer
```

```
316                     ENCRYPT(&keySchedule, iv, tmp);
317                     pT = tmp;
318                     pIv = iv;
319                     for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
320                         // Copy the current cipher text to IV, XOR
321                         // with the temp buffer and put into the output
322                         *dOut++ = *pT++ ^ (*pIv++ = *dIn++);
323                 }
324                 // If the inner loop (i loop) was smaller than blockSize, then dSize
325                 // would have been smaller than blockSize and it is now negative
326                 // If it is negative, then it indicates how may fill bytes
327                 // are needed to pad out the IV for the next round.
328                 for(; dSize < 0; dSize++)
329                     *pIv++ = 0;
330
331                 break;
332     #if ALG_CTR
333             case ALG_CTR_VALUE:
334                 for(; dSize > 0; dSize -= blockSize)
335                 {
336                     // Encrypt the current value of the IV(counter)
337                     ENCRYPT(&keySchedule, iv, tmp);
338
339                     //increment the counter (counter is big-endian so start at end)
340                     for(i = blockSize - 1; i >= 0; i--)
341                         if((iv[i] += 1) != 0)
342                             break;
343                     // XOR the encrypted counter value with input and put into output
344                     pT = tmp;
345                     for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
346                         *dOut++ = *dIn++ ^ *pT++;
347                 }
348                 break;
349     #endif
350     #if ALG_ECB
351             case ALG_ECB_VALUE:
352                 for(; dSize > 0; dSize -= blockSize)
353                 {
354                     DECRYPT(&keySchedule, dIn, dOut);
355                     dIn = &dIn[blockSize];
356                     dOut = &dOut[blockSize];
357                 }
358                 break;
359     #endif
360     #if ALG_OFB
361             case ALG_OFB_VALUE:
362                 // This is written so that dIn and dOut may be the same
363                 for(; dSize > 0; dSize -= blockSize)
364                 {
365                     // Encrypt the current value of the "IV"
366                     ENCRYPT(&keySchedule, iv, iv);
367
368                     // XOR the encrypted IV into dIn to create the cipher text (dOut)
369                     pIv = iv;
370                     for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
371                         *dOut++ = (*pIv++ ^ *dIn++);
372                 }
373                 break;
374     #endif
375             default:
376                 return TPM_RC_FAILURE;
377         }
378         return TPM_RC_SUCCESS;
379     }
```

10.2.19.4.2 CryptSymKeyValidate()

Validate that a provided symmetric key meets the requirements of the TPM

Error Returns	Meaning
TPM_RC_KEY_SIZE	Key size specifiers do not match
TPM_RC_KEY	Key is not allowed

```
380 TPM_RC
381 CryptSymKeyValidate(
382     TPMT_SYM_DEF_OBJECT *symDef,
383     TPM2B_SYM_KEY       *key
384     )
385 {
386     if(key->t.size != BITS_TO_BYTES(symDef->keyBits.sym))
387         return TPM_RCS_KEY_SIZE;
388 #if ALG_TDES
389     if(symDef->algorithm == TPM_ALG_TDES && !CryptDesValidateKey(key))
390         return TPM_RCS_KEY;
391 #endif // ALG_TDES
392     return TPM_RC_SUCCESS;
393 }
```

10.2.20 PrimeData.c

```
1    #include "Tpm.h"
```

This table is the product of all of the primes up to 1000. Checking to see if there is a GCD between a prime candidate and this number will eliminate many prime candidates from consideration before running Miller-Rabin on the result.

```
2    const BN_STRUCT(43 * RADIX_BITS) s_CompositeOfSmallPrimes_ =
3    {44, 44,
4    { 0x2ED42696, 0x2BBFA177, 0x4820594F, 0xF73F4841,
5    0xBFAC313A, 0xCAC3EB81, 0xF6F26BF8, 0x7FAB5061,
6    0x59746FB7, 0xF71377F6, 0x3B19855B, 0xCBD03132,
7    0xBB92EF1B, 0x3AC3152C, 0xE87C8273, 0xC0AE0E69,
8    0x74A9E295, 0x448CCE86, 0x63CA1907, 0x8A0BF944,
9    0xF8CC3BE0, 0xC26F0AF5, 0xC501C02F, 0x6579441A,
10   0xD1099CDA, 0x6BC76A00, 0xC81A3228, 0xBFB1AB25,
11   0x70FA3841, 0x51B3D076, 0xCC2359ED, 0xD9EE0769,
12   0x75E47AF0, 0xD45FF31E, 0x52CCE4F6, 0x04DBC891,
13   0x96658ED2, 0x1753EFE5, 0x3AE4A5A6, 0x8FD4A97F,
14   0x8B15E7EB, 0x0243C3E1, 0xE0F0C31D, 0x0000000B }
15   };
16   bigConst        s_CompositeOfSmallPrimes = (const bigNum)&s_CompositeOfSmallPrimes_;
```

This table contains a bit for each of the odd values between 1 and 2^16 + 1. This table allows fast checking of the primes in that range. Don't change the size of this table unless you are prepared to do redo IsPrimeInt().

```
17   const uint32_t      s_LastPrimeInTable = 65537;
18   const uint32_t      s_PrimeTableSize = 4097;
19   const uint32_t      s_PrimesInTable = 6542;
20   const unsigned char s_PrimeTable[] = {
21       0x6e, 0xcb, 0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x81, 0x32, 0x4c, 0x4a, 0x86,
22       0x0d, 0x82, 0x96, 0x21, 0xc9, 0x34, 0x04, 0x5a, 0x20, 0x61, 0x89, 0xa4,
23       0x44, 0x11, 0x86, 0x29, 0xd1, 0x82, 0x28, 0x4a, 0x30, 0x40, 0x42, 0x32,
24       0x21, 0x99, 0x34, 0x08, 0x4b, 0x06, 0x25, 0x42, 0x84, 0x48, 0x8a, 0x14,
25       0x05, 0x42, 0x30, 0x6c, 0x08, 0xb4, 0x40, 0x0b, 0xa0, 0x08, 0x51, 0x12,
26       0x28, 0x89, 0x04, 0x65, 0x98, 0x30, 0x4c, 0x80, 0x96, 0x44, 0x12, 0x80,
27       0x21, 0x42, 0x12, 0x41, 0xc9, 0x04, 0x21, 0xc0, 0x32, 0x2d, 0x98, 0x00,
28       0x00, 0x49, 0x04, 0x08, 0x81, 0x96, 0x68, 0x82, 0xb0, 0x25, 0x08, 0x22,
29       0x48, 0x89, 0xa2, 0x40, 0x59, 0x26, 0x04, 0x90, 0x06, 0x40, 0x43, 0x30,
30       0x44, 0x92, 0x00, 0x69, 0x10, 0x82, 0x08, 0x08, 0xa4, 0x0d, 0x41, 0x12,
31       0x60, 0xc0, 0x00, 0x24, 0xd2, 0x22, 0x61, 0x08, 0x84, 0x04, 0x1b, 0x82,
32       0x01, 0xd3, 0x10, 0x01, 0x02, 0xa0, 0x44, 0xc0, 0x22, 0x60, 0x91, 0x14,
33       0x0c, 0x40, 0xa6, 0x04, 0xd2, 0x94, 0x20, 0x09, 0x94, 0x20, 0x52, 0x00,
34       0x08, 0x10, 0xa2, 0x4c, 0x00, 0x82, 0x01, 0x51, 0x10, 0x08, 0x8b, 0xa4,
35       0x25, 0x9a, 0x30, 0x44, 0x81, 0x10, 0x4c, 0x03, 0x02, 0x25, 0x52, 0x80,
36       0x08, 0x49, 0x84, 0x20, 0x50, 0x32, 0x00, 0x18, 0xa2, 0x40, 0x11, 0x24,
37       0x28, 0x01, 0x84, 0x01, 0x01, 0xa0, 0x41, 0x0a, 0x12, 0x45, 0x00, 0x36,
38       0x08, 0x00, 0x26, 0x29, 0x83, 0x82, 0x61, 0xc0, 0x80, 0x04, 0x10, 0x10,
39       0x6d, 0x00, 0x22, 0x48, 0x58, 0x26, 0x0c, 0xc2, 0x10, 0x48, 0x89, 0x24,
40       0x20, 0x58, 0x20, 0x45, 0x88, 0x24, 0x00, 0x19, 0x02, 0x25, 0xc0, 0x10,
41       0x68, 0x08, 0x14, 0x01, 0xca, 0x32, 0x28, 0x80, 0x00, 0x04, 0x4b, 0x26,
42       0x00, 0x13, 0x90, 0x60, 0x82, 0x80, 0x25, 0xd0, 0x00, 0x01, 0x10, 0x32,
43       0x0c, 0x43, 0x86, 0x21, 0x11, 0x00, 0x08, 0x43, 0x24, 0x04, 0x48, 0x10,
44       0x0c, 0x90, 0x92, 0x00, 0x43, 0x20, 0x2d, 0x00, 0x06, 0x09, 0x88, 0x24,
45       0x40, 0xc0, 0x32, 0x09, 0x09, 0x82, 0x00, 0x53, 0x80, 0x08, 0x80, 0x96,
46       0x41, 0x81, 0x00, 0x40, 0x48, 0x10, 0x48, 0x08, 0x96, 0x48, 0x58, 0x20,
47       0x29, 0xc3, 0x80, 0x20, 0x02, 0x94, 0x60, 0x92, 0x00, 0x20, 0x81, 0x22,
48       0x44, 0x10, 0xa0, 0x05, 0x40, 0x90, 0x01, 0x49, 0x20, 0x04, 0x0a, 0x00,
49       0x24, 0x89, 0x34, 0x48, 0x13, 0x80, 0x2c, 0xc0, 0x82, 0x29, 0x00, 0x24,
```

 Copyright © TCG 2006-2020

```
50        0x45, 0x08, 0x00, 0x08, 0x98, 0x36, 0x04, 0x52, 0x84, 0x04, 0xd0, 0x04,
51        0x00, 0x8a, 0x90, 0x44, 0x82, 0x32, 0x65, 0x18, 0x90, 0x00, 0x0a, 0x02,
52        0x01, 0x40, 0x02, 0x28, 0x40, 0xa4, 0x04, 0x92, 0x30, 0x04, 0x11, 0x86,
53        0x08, 0x42, 0x00, 0x2c, 0x52, 0x04, 0x08, 0xc9, 0x84, 0x60, 0x48, 0x12,
54        0x09, 0x99, 0x24, 0x44, 0x00, 0x24, 0x00, 0x03, 0x14, 0x21, 0x00, 0x10,
55        0x01, 0x1a, 0x32, 0x05, 0x88, 0x20, 0x40, 0x40, 0x06, 0x09, 0xc3, 0x84,
56        0x40, 0x01, 0x30, 0x60, 0x18, 0x02, 0x68, 0x11, 0x90, 0x0c, 0x02, 0xa2,
57        0x04, 0x00, 0x86, 0x29, 0x89, 0x14, 0x24, 0x82, 0x02, 0x41, 0x08, 0x80,
58        0x04, 0x19, 0x80, 0x08, 0x10, 0x12, 0x68, 0x42, 0xa4, 0x04, 0x00, 0x02,
59        0x61, 0x10, 0x06, 0x0c, 0x10, 0x00, 0x01, 0x12, 0x10, 0x20, 0x03, 0x94,
60        0x21, 0x42, 0x12, 0x65, 0x18, 0x94, 0x0c, 0x0a, 0x04, 0x28, 0x01, 0x14,
61        0x29, 0x0a, 0xa4, 0x40, 0xd0, 0x00, 0x40, 0x01, 0x90, 0x04, 0x41, 0x20,
62        0x2d, 0x40, 0x82, 0x48, 0xc1, 0x20, 0x00, 0x10, 0x30, 0x01, 0x08, 0x24,
63        0x04, 0x59, 0x84, 0x24, 0x00, 0x02, 0x29, 0x82, 0x00, 0x61, 0x58, 0x02,
64        0x48, 0x81, 0x16, 0x48, 0x10, 0x00, 0x21, 0x11, 0x06, 0x00, 0xca, 0xa0,
65        0x40, 0x02, 0x00, 0x04, 0x91, 0xb0, 0x00, 0x42, 0x04, 0x0c, 0x81, 0x06,
66        0x09, 0x48, 0x14, 0x25, 0x92, 0x20, 0x25, 0x11, 0xa0, 0x00, 0x0a, 0x86,
67        0x0c, 0xc1, 0x02, 0x48, 0x00, 0x20, 0x45, 0x08, 0x32, 0x00, 0x98, 0x06,
68        0x04, 0x13, 0x22, 0x00, 0x82, 0x04, 0x48, 0x81, 0x14, 0x44, 0x82, 0x12,
69        0x24, 0x18, 0x10, 0x40, 0x43, 0x80, 0x28, 0xd0, 0x04, 0x20, 0x81, 0x24,
70        0x64, 0xd8, 0x00, 0x2c, 0x09, 0x12, 0x08, 0x41, 0xa2, 0x00, 0x00, 0x02,
71        0x41, 0xca, 0x20, 0x41, 0xc0, 0x10, 0x01, 0x18, 0xa4, 0x04, 0x18, 0xa4,
72        0x20, 0x12, 0x94, 0x20, 0x83, 0xa0, 0x40, 0x02, 0x32, 0x44, 0x80, 0x04,
73        0x00, 0x18, 0x00, 0x0c, 0x40, 0x86, 0x60, 0x8a, 0x00, 0x64, 0x88, 0x12,
74        0x05, 0x01, 0x82, 0x00, 0x4a, 0xa2, 0x01, 0xc1, 0x10, 0x61, 0x09, 0x04,
75        0x01, 0x88, 0x00, 0x60, 0x01, 0xb4, 0x40, 0x08, 0x06, 0x01, 0x03, 0x80,
76        0x08, 0x40, 0x94, 0x04, 0x8a, 0x20, 0x29, 0x80, 0x02, 0x0c, 0x52, 0x02,
77        0x01, 0x42, 0x84, 0x00, 0x80, 0x84, 0x64, 0x02, 0x32, 0x48, 0x00, 0x30,
78        0x44, 0x40, 0x22, 0x21, 0x00, 0x02, 0x08, 0xc3, 0xa0, 0x04, 0xd0, 0x20,
79        0x40, 0x18, 0x16, 0x40, 0x40, 0x00, 0x28, 0x52, 0x90, 0x08, 0x82, 0x14,
80        0x01, 0x18, 0x10, 0x08, 0x09, 0x82, 0x40, 0x0a, 0xa0, 0x20, 0x93, 0x80,
81        0x08, 0xc0, 0x00, 0x20, 0x52, 0x00, 0x05, 0x01, 0x10, 0x40, 0x11, 0x06,
82        0x0c, 0x82, 0x00, 0x00, 0x4b, 0x90, 0x44, 0x9a, 0x00, 0x28, 0x80, 0x90,
83        0x04, 0x4a, 0x06, 0x09, 0x43, 0x02, 0x28, 0x00, 0x34, 0x01, 0x18, 0x00,
84        0x65, 0x09, 0x80, 0x44, 0x03, 0x00, 0x24, 0x02, 0x82, 0x61, 0x48, 0x14,
85        0x41, 0x00, 0x12, 0x28, 0x00, 0x34, 0x08, 0x51, 0x04, 0x05, 0x12, 0x90,
86        0x28, 0x89, 0x84, 0x60, 0x12, 0x10, 0x49, 0x10, 0x26, 0x40, 0x49, 0x82,
87        0x00, 0x91, 0x10, 0x01, 0x0a, 0x24, 0x40, 0x88, 0x10, 0x4c, 0x10, 0x04,
88        0x00, 0x50, 0xa2, 0x2c, 0x40, 0x90, 0x48, 0x0a, 0xb0, 0x01, 0x50, 0x12,
89        0x08, 0x00, 0xa4, 0x04, 0x09, 0xa0, 0x28, 0x92, 0x02, 0x00, 0x43, 0x10,
90        0x21, 0x02, 0x20, 0x41, 0x81, 0x32, 0x00, 0x08, 0x04, 0x0c, 0x52, 0x00,
91        0x21, 0x49, 0x84, 0x20, 0x10, 0x02, 0x01, 0x81, 0x10, 0x48, 0x40, 0x22,
92        0x01, 0x01, 0x84, 0x69, 0xc1, 0x30, 0x01, 0xc8, 0x02, 0x44, 0x88, 0x00,
93        0x0c, 0x01, 0x02, 0x2d, 0xc0, 0x12, 0x61, 0x00, 0xa0, 0x00, 0xc0, 0x30,
94        0x40, 0x01, 0x12, 0x08, 0x0b, 0x20, 0x00, 0x80, 0x94, 0x40, 0x01, 0x84,
95        0x40, 0x00, 0x32, 0x00, 0x10, 0x84, 0x00, 0x0b, 0x24, 0x00, 0x01, 0x06,
96        0x29, 0x8a, 0x84, 0x41, 0x80, 0x10, 0x08, 0x08, 0x94, 0x4c, 0x03, 0x80,
97        0x01, 0x40, 0x96, 0x40, 0x41, 0x20, 0x20, 0x50, 0x22, 0x25, 0x89, 0xa2,
98        0x40, 0x40, 0xa4, 0x20, 0x02, 0x86, 0x28, 0x01, 0x20, 0x21, 0x4a, 0x10,
99        0x08, 0x00, 0x14, 0x08, 0x40, 0x04, 0x25, 0x42, 0x02, 0x21, 0x43, 0x10,
100       0x04, 0x92, 0x00, 0x21, 0x11, 0xa0, 0x4c, 0x18, 0x22, 0x09, 0x03, 0x84,
101       0x41, 0x89, 0x10, 0x04, 0x82, 0x22, 0x24, 0x01, 0x14, 0x08, 0x08, 0x84,
102       0x08, 0xc1, 0x00, 0x09, 0x42, 0xb0, 0x41, 0x8a, 0x02, 0x00, 0x80, 0x36,
103       0x04, 0x49, 0xa0, 0x24, 0x91, 0x00, 0x00, 0x02, 0x94, 0x41, 0x92, 0x02,
104       0x01, 0x08, 0x06, 0x08, 0x09, 0x00, 0x01, 0xd0, 0x16, 0x28, 0x89, 0x80,
105       0x60, 0x00, 0x00, 0x68, 0x01, 0x90, 0x0c, 0x50, 0x20, 0x01, 0x40, 0x80,
106       0x40, 0x42, 0x30, 0x41, 0x00, 0x20, 0x25, 0x81, 0x06, 0x40, 0x49, 0x00,
107       0x08, 0x01, 0x12, 0x49, 0x00, 0xa0, 0x20, 0x18, 0x30, 0x05, 0x01, 0xa6,
108       0x00, 0x10, 0x24, 0x28, 0x00, 0x02, 0x20, 0xc8, 0x20, 0x00, 0x88, 0x12,
109       0x0c, 0x90, 0x92, 0x00, 0x02, 0x26, 0x01, 0x42, 0x16, 0x49, 0x00, 0x04,
110       0x24, 0x42, 0x02, 0x01, 0x88, 0x80, 0x0c, 0x1a, 0x80, 0x08, 0x10, 0x00,
111       0x60, 0x02, 0x94, 0x44, 0x88, 0x00, 0x69, 0x11, 0x30, 0x08, 0x12, 0xa0,
```

```
112        0x24, 0x13, 0x84, 0x00, 0x82, 0x00, 0x65, 0xc0, 0x10, 0x28, 0x00, 0x30,
113        0x04, 0x03, 0x20, 0x01, 0x11, 0x06, 0x01, 0xc8, 0x80, 0x00, 0xc2, 0x20,
114        0x08, 0x10, 0x82, 0x0c, 0x13, 0x02, 0x0c, 0x52, 0x06, 0x40, 0x00, 0xb0,
115        0x61, 0x40, 0x10, 0x01, 0x98, 0x86, 0x04, 0x10, 0x84, 0x08, 0x92, 0x14,
116        0x60, 0x41, 0x80, 0x41, 0x1a, 0x10, 0x04, 0x81, 0x22, 0x40, 0x41, 0x20,
117        0x29, 0x52, 0x00, 0x41, 0x08, 0x34, 0x60, 0x10, 0x00, 0x28, 0x01, 0x10,
118        0x40, 0x00, 0x84, 0x08, 0x42, 0x90, 0x20, 0x48, 0x04, 0x04, 0x52, 0x02,
119        0x00, 0x08, 0x20, 0x04, 0x00, 0x82, 0x0d, 0x00, 0x82, 0x40, 0x02, 0x10,
120        0x05, 0x48, 0x20, 0x40, 0x99, 0x00, 0x00, 0x01, 0x06, 0x24, 0xc0, 0x00,
121        0x68, 0x82, 0x04, 0x21, 0x12, 0x10, 0x44, 0x08, 0x04, 0x00, 0x40, 0xa6,
122        0x20, 0xd0, 0x16, 0x09, 0xc9, 0x24, 0x41, 0x02, 0x20, 0x0c, 0x09, 0x92,
123        0x40, 0x12, 0x00, 0x00, 0x40, 0x00, 0x09, 0x43, 0x84, 0x20, 0x98, 0x02,
124        0x01, 0x11, 0x24, 0x00, 0x43, 0x24, 0x00, 0x03, 0x90, 0x08, 0x41, 0x30,
125        0x24, 0x58, 0x20, 0x4c, 0x80, 0x82, 0x08, 0x10, 0x24, 0x25, 0x81, 0x06,
126        0x41, 0x09, 0x10, 0x20, 0x18, 0x10, 0x44, 0x80, 0x10, 0x00, 0x4a, 0x24,
127        0x0d, 0x01, 0x94, 0x28, 0x80, 0x30, 0x00, 0xc0, 0x02, 0x60, 0x10, 0x84,
128        0x0c, 0x02, 0x00, 0x09, 0x02, 0x82, 0x01, 0x08, 0x10, 0x04, 0xc2, 0x20,
129        0x68, 0x09, 0x06, 0x04, 0x18, 0x00, 0x00, 0x11, 0x90, 0x08, 0x0b, 0x10,
130        0x21, 0x82, 0x02, 0x0c, 0x10, 0xb6, 0x08, 0x00, 0x26, 0x00, 0x41, 0x02,
131        0x01, 0x4a, 0x24, 0x21, 0x1a, 0x20, 0x24, 0x80, 0x00, 0x44, 0x02, 0x00,
132        0x2d, 0x40, 0x02, 0x00, 0x8b, 0x94, 0x20, 0x10, 0x00, 0x20, 0x90, 0xa6,
133        0x40, 0x13, 0x00, 0x2c, 0x11, 0x86, 0x61, 0x01, 0x80, 0x41, 0x10, 0x02,
134        0x04, 0x81, 0x30, 0x48, 0x48, 0x20, 0x28, 0x50, 0x80, 0x21, 0x8a, 0x10,
135        0x04, 0x08, 0x10, 0x09, 0x10, 0x10, 0x48, 0x42, 0xa0, 0x0c, 0x82, 0x92,
136        0x60, 0xc0, 0x20, 0x05, 0xd2, 0x20, 0x40, 0x01, 0x00, 0x04, 0x08, 0x82,
137        0x2d, 0x82, 0x02, 0x00, 0x48, 0x80, 0x41, 0x48, 0x10, 0x00, 0x91, 0x04,
138        0x04, 0x03, 0x84, 0x00, 0xc2, 0x04, 0x68, 0x00, 0x00, 0x64, 0xc0, 0x22,
139        0x40, 0x08, 0x32, 0x44, 0x09, 0x86, 0x00, 0x91, 0x02, 0x28, 0x01, 0x00,
140        0x64, 0x48, 0x00, 0x24, 0x10, 0x90, 0x00, 0x43, 0x00, 0x21, 0x52, 0x86,
141        0x41, 0x8b, 0x90, 0x20, 0x40, 0x20, 0x08, 0x88, 0x04, 0x44, 0x13, 0x20,
142        0x00, 0x02, 0x84, 0x60, 0x81, 0x90, 0x24, 0x40, 0x30, 0x00, 0x08, 0x10,
143        0x08, 0x08, 0x02, 0x01, 0x10, 0x04, 0x20, 0x43, 0xb4, 0x40, 0x90, 0x12,
144        0x68, 0x01, 0x80, 0x4c, 0x18, 0x00, 0x08, 0xc0, 0x12, 0x49, 0x40, 0x10,
145        0x24, 0x1a, 0x00, 0x41, 0x89, 0x24, 0x4c, 0x10, 0x00, 0x04, 0x52, 0x10,
146        0x09, 0x4a, 0x20, 0x41, 0x48, 0x22, 0x69, 0x11, 0x14, 0x08, 0x10, 0x06,
147        0x24, 0x80, 0x84, 0x28, 0x00, 0x10, 0x00, 0x40, 0x10, 0x01, 0x08, 0x26,
148        0x08, 0x48, 0x06, 0x28, 0x00, 0x14, 0x01, 0x42, 0x84, 0x04, 0x0a, 0x20,
149        0x00, 0x01, 0x82, 0x08, 0x00, 0x82, 0x24, 0x12, 0x04, 0x40, 0x40, 0xa0,
150        0x40, 0x90, 0x10, 0x04, 0x90, 0x22, 0x40, 0x10, 0x20, 0x2c, 0x80, 0x10,
151        0x28, 0x43, 0x00, 0x04, 0x58, 0x00, 0x01, 0x81, 0x10, 0x48, 0x09, 0x20,
152        0x21, 0x83, 0x04, 0x00, 0x42, 0xa4, 0x44, 0x00, 0x00, 0x6c, 0x10, 0xa0,
153        0x44, 0x48, 0x80, 0x00, 0x83, 0x80, 0x48, 0xc9, 0x00, 0x00, 0x00, 0x02,
154        0x05, 0x10, 0xb0, 0x04, 0x13, 0x04, 0x29, 0x10, 0x92, 0x40, 0x08, 0x04,
155        0x44, 0x82, 0x22, 0x00, 0x19, 0x20, 0x00, 0x19, 0x20, 0x01, 0x81, 0x90,
156        0x60, 0x8a, 0x00, 0x41, 0xc0, 0x02, 0x45, 0x10, 0x04, 0x00, 0x02, 0xa2,
157        0x09, 0x40, 0x10, 0x21, 0x49, 0x20, 0x01, 0x42, 0x30, 0x2c, 0x00, 0x14,
158        0x44, 0x01, 0x22, 0x04, 0x02, 0x92, 0x08, 0x89, 0x04, 0x21, 0x80, 0x10,
159        0x05, 0x01, 0x20, 0x40, 0x41, 0x80, 0x04, 0x00, 0x12, 0x09, 0x40, 0xb0,
160        0x64, 0x58, 0x32, 0x01, 0x08, 0x90, 0x00, 0x41, 0x04, 0x09, 0xc1, 0x80,
161        0x61, 0x08, 0x90, 0x00, 0x9a, 0x00, 0x24, 0x01, 0x12, 0x08, 0x02, 0x26,
162        0x05, 0x82, 0x06, 0x08, 0x08, 0x00, 0x20, 0x48, 0x20, 0x00, 0x18, 0x24,
163        0x48, 0x03, 0x02, 0x00, 0x11, 0x00, 0x09, 0x00, 0x84, 0x01, 0x4a, 0x10,
164        0x01, 0x98, 0x00, 0x04, 0x18, 0x86, 0x00, 0xc0, 0x00, 0x20, 0x81, 0x80,
165        0x04, 0x10, 0x30, 0x05, 0x00, 0xb4, 0x0c, 0x4a, 0x82, 0x29, 0x91, 0x02,
166        0x28, 0x00, 0x20, 0x44, 0xc0, 0x00, 0x2c, 0x91, 0x80, 0x40, 0x01, 0xa2,
167        0x00, 0x12, 0x04, 0x09, 0xc3, 0x20, 0x00, 0x08, 0x02, 0x0c, 0x10, 0x22,
168        0x04, 0x00, 0x00, 0x2c, 0x11, 0x86, 0x00, 0xc0, 0x00, 0x00, 0x12, 0x32,
169        0x40, 0x89, 0x80, 0x40, 0x40, 0x02, 0x05, 0x50, 0x86, 0x60, 0x82, 0xa4,
170        0x60, 0x0a, 0x12, 0x4d, 0x80, 0x90, 0x08, 0x12, 0x80, 0x09, 0x02, 0x14,
171        0x48, 0x01, 0x24, 0x20, 0x8a, 0x00, 0x44, 0x90, 0x04, 0x04, 0x01, 0x02,
172        0x00, 0xd1, 0x12, 0x00, 0x0a, 0x04, 0x40, 0x00, 0x32, 0x21, 0x81, 0x24,
173        0x08, 0x19, 0x84, 0x20, 0x02, 0x04, 0x08, 0x89, 0x80, 0x24, 0x02, 0x02,
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
174        0x68, 0x18, 0x82, 0x44, 0x42, 0x00, 0x21, 0x40, 0x00, 0x28, 0x01, 0x80,
175        0x45, 0x82, 0x20, 0x40, 0x11, 0x80, 0x0c, 0x02, 0x00, 0x24, 0x40, 0x90,
176        0x01, 0x40, 0x20, 0x20, 0x50, 0x20, 0x28, 0x19, 0x00, 0x40, 0x09, 0x20,
177        0x08, 0x80, 0x04, 0x60, 0x40, 0x80, 0x20, 0x08, 0x30, 0x49, 0x09, 0x34,
178        0x00, 0x11, 0x24, 0x24, 0x82, 0x00, 0x41, 0xc2, 0x00, 0x04, 0x92, 0x02,
179        0x24, 0x80, 0x00, 0x0c, 0x02, 0xa0, 0x00, 0x01, 0x06, 0x60, 0x41, 0x04,
180        0x21, 0xd0, 0x00, 0x01, 0x01, 0x00, 0x48, 0x12, 0x84, 0x04, 0x91, 0x12,
181        0x08, 0x00, 0x24, 0x44, 0x00, 0x12, 0x41, 0x18, 0x26, 0x0c, 0x41, 0x80,
182        0x00, 0x52, 0x04, 0x20, 0x09, 0x00, 0x24, 0x90, 0x20, 0x48, 0x18, 0x02,
183        0x00, 0x03, 0xa2, 0x09, 0xd0, 0x14, 0x00, 0x8a, 0x84, 0x25, 0x4a, 0x00,
184        0x20, 0x98, 0x14, 0x40, 0x00, 0xa2, 0x05, 0x00, 0x00, 0x00, 0x40, 0x14,
185        0x01, 0x58, 0x20, 0x2c, 0x80, 0x84, 0x00, 0x09, 0x20, 0x20, 0x91, 0x02,
186        0x08, 0x02, 0xb0, 0x41, 0x08, 0x30, 0x00, 0x09, 0x10, 0x00, 0x18, 0x02,
187        0x21, 0x02, 0x02, 0x00, 0x00, 0x24, 0x44, 0x08, 0x12, 0x60, 0x00, 0xb2,
188        0x44, 0x12, 0x02, 0x0c, 0xc0, 0x80, 0x40, 0xc8, 0x20, 0x04, 0x50, 0x20,
189        0x05, 0x00, 0xb0, 0x04, 0x0b, 0x04, 0x29, 0x53, 0x00, 0x61, 0x48, 0x30,
190        0x00, 0x82, 0x20, 0x29, 0x00, 0x16, 0x00, 0x53, 0x22, 0x20, 0x43, 0x10,
191        0x48, 0x00, 0x80, 0x04, 0xd2, 0x00, 0x40, 0x00, 0xa2, 0x44, 0x03, 0x80,
192        0x29, 0x00, 0x04, 0x08, 0xc0, 0x04, 0x64, 0x40, 0x30, 0x28, 0x09, 0x84,
193        0x44, 0x50, 0x80, 0x21, 0x02, 0x92, 0x00, 0xc0, 0x10, 0x60, 0x88, 0x22,
194        0x08, 0x80, 0x00, 0x00, 0x18, 0x84, 0x04, 0x83, 0x96, 0x00, 0x81, 0x20,
195        0x05, 0x02, 0x00, 0x45, 0x88, 0x84, 0x00, 0x51, 0x20, 0x20, 0x51, 0x86,
196        0x41, 0x4b, 0x94, 0x00, 0x80, 0x00, 0x08, 0x11, 0x20, 0x4c, 0x58, 0x80,
197        0x04, 0x03, 0x06, 0x20, 0x89, 0x00, 0x05, 0x08, 0x22, 0x05, 0x90, 0x00,
198        0x40, 0x00, 0x82, 0x09, 0x50, 0x00, 0x00, 0x00, 0xa0, 0x41, 0xc2, 0x20,
199        0x08, 0x00, 0x16, 0x08, 0x40, 0x26, 0x21, 0xd0, 0x90, 0x08, 0x81, 0x90,
200        0x41, 0x00, 0x02, 0x44, 0x08, 0x10, 0x0c, 0x0a, 0x86, 0x09, 0x90, 0x04,
201        0x00, 0xc8, 0xa0, 0x04, 0x08, 0x30, 0x20, 0x89, 0x84, 0x00, 0x11, 0x22,
202        0x2c, 0x40, 0x00, 0x08, 0x02, 0xb0, 0x01, 0x48, 0x02, 0x01, 0x09, 0x20,
203        0x04, 0x03, 0x04, 0x00, 0x80, 0x02, 0x60, 0x42, 0x30, 0x21, 0x4a, 0x10,
204        0x44, 0x09, 0x02, 0x00, 0x01, 0x24, 0x00, 0x12, 0x82, 0x21, 0x80, 0xa4,
205        0x20, 0x10, 0x02, 0x04, 0x91, 0xa0, 0x40, 0x18, 0x04, 0x00, 0x02, 0x06,
206        0x69, 0x09, 0x00, 0x05, 0x58, 0x02, 0x01, 0x00, 0x00, 0x48, 0x00, 0x00,
207        0x00, 0x03, 0x92, 0x20, 0x00, 0x34, 0x01, 0xc8, 0x20, 0x48, 0x08, 0x30,
208        0x08, 0x42, 0x80, 0x20, 0x91, 0x90, 0x68, 0x01, 0x04, 0x40, 0x12, 0x02,
209        0x61, 0x00, 0x12, 0x08, 0x01, 0xa0, 0x00, 0x11, 0x04, 0x21, 0x48, 0x04,
210        0x24, 0x92, 0x00, 0x0c, 0x01, 0x84, 0x04, 0x00, 0x00, 0x01, 0x12, 0x96,
211        0x40, 0x01, 0xa0, 0x41, 0x88, 0x22, 0x28, 0x88, 0x00, 0x44, 0x42, 0x80,
212        0x24, 0x12, 0x14, 0x01, 0x42, 0x90, 0x60, 0x1a, 0x10, 0x04, 0x81, 0x10,
213        0x48, 0x08, 0x06, 0x29, 0x83, 0x02, 0x40, 0x02, 0x24, 0x64, 0x80, 0x10,
214        0x05, 0x80, 0x10, 0x40, 0x02, 0x02, 0x08, 0x42, 0x84, 0x01, 0x09, 0x20,
215        0x04, 0x50, 0x00, 0x60, 0x11, 0x30, 0x40, 0x13, 0x02, 0x04, 0x81, 0x00,
216        0x09, 0x08, 0x20, 0x45, 0x4a, 0x10, 0x61, 0x90, 0x26, 0x0c, 0x08, 0x02,
217        0x21, 0x91, 0x00, 0x60, 0x02, 0x04, 0x00, 0x02, 0x00, 0x0c, 0x08, 0x06,
218        0x08, 0x48, 0x84, 0x08, 0x11, 0x02, 0x00, 0x80, 0xa4, 0x00, 0x5a, 0x20,
219        0x00, 0x88, 0x04, 0x04, 0x02, 0x00, 0x09, 0x00, 0x14, 0x08, 0x49, 0x14,
220        0x20, 0xc8, 0x00, 0x04, 0x91, 0xa0, 0x40, 0x59, 0x80, 0x00, 0x12, 0x10,
221        0x00, 0x80, 0x80, 0x65, 0x00, 0x00, 0x04, 0x00, 0x80, 0x40, 0x19, 0x00,
222        0x21, 0x03, 0x84, 0x60, 0xc0, 0x04, 0x24, 0x1a, 0x12, 0x61, 0x80, 0x80,
223        0x08, 0x02, 0x04, 0x09, 0x42, 0x12, 0x20, 0x08, 0x34, 0x04, 0x90, 0x20,
224        0x01, 0x01, 0xa0, 0x00, 0x0b, 0x00, 0x08, 0x91, 0x92, 0x40, 0x02, 0x34,
225        0x40, 0x88, 0x10, 0x61, 0x19, 0x02, 0x00, 0x40, 0x04, 0x25, 0xc0, 0x80,
226        0x68, 0x08, 0x04, 0x21, 0x80, 0x22, 0x04, 0x00, 0xa0, 0x0c, 0x01, 0x84,
227        0x20, 0x41, 0x00, 0x08, 0x8a, 0x00, 0x20, 0x8a, 0x00, 0x48, 0x88, 0x04,
228        0x04, 0x11, 0x82, 0x08, 0x40, 0x86, 0x09, 0x49, 0xa4, 0x40, 0x00, 0x10,
229        0x01, 0x01, 0xa2, 0x04, 0x50, 0x80, 0x0c, 0x80, 0x00, 0x48, 0x82, 0xa0,
230        0x01, 0x18, 0x12, 0x41, 0x01, 0x04, 0x48, 0x41, 0x00, 0x24, 0x01, 0x00,
231        0x00, 0x88, 0x14, 0x00, 0x02, 0x00, 0x68, 0x01, 0x20, 0x08, 0x4a, 0x22,
232        0x08, 0x83, 0x80, 0x00, 0x89, 0x04, 0x01, 0xc2, 0x00, 0x00, 0x00, 0x34,
233        0x04, 0x00, 0x82, 0x28, 0x02, 0x02, 0x41, 0x4a, 0x90, 0x05, 0x82, 0x02,
234        0x09, 0x80, 0x24, 0x04, 0x41, 0x00, 0x01, 0x92, 0x80, 0x28, 0x01, 0x14,
235        0x00, 0x50, 0x20, 0x4c, 0x10, 0xb0, 0x04, 0x43, 0xa4, 0x21, 0x90, 0x04,
```

```
236       0x01, 0x02, 0x00, 0x44, 0x48, 0x00, 0x64, 0x08, 0x06, 0x00, 0x42, 0x20,
237       0x08, 0x02, 0x92, 0x01, 0x4a, 0x00, 0x20, 0x50, 0x32, 0x25, 0x90, 0x22,
238       0x04, 0x09, 0x00, 0x08, 0x11, 0x80, 0x21, 0x01, 0x10, 0x05, 0x00, 0x32,
239       0x08, 0x88, 0x94, 0x08, 0x08, 0x24, 0x0d, 0xc1, 0x80, 0x40, 0x0b, 0x20,
240       0x40, 0x18, 0x12, 0x04, 0x00, 0x22, 0x40, 0x10, 0x26, 0x05, 0xc1, 0x82,
241       0x00, 0x01, 0x30, 0x24, 0x02, 0x22, 0x41, 0x08, 0x24, 0x48, 0x1a, 0x00,
242       0x25, 0xd2, 0x12, 0x28, 0x42, 0x00, 0x04, 0x40, 0x30, 0x41, 0x00, 0x02,
243       0x00, 0x13, 0x20, 0x24, 0xd1, 0x84, 0x08, 0x89, 0x80, 0x04, 0x52, 0x00,
244       0x44, 0x18, 0xa4, 0x00, 0x00, 0x06, 0x20, 0x91, 0x10, 0x09, 0x42, 0x20,
245       0x24, 0x40, 0x30, 0x28, 0x00, 0x84, 0x40, 0x40, 0x80, 0x08, 0x10, 0x04,
246       0x09, 0x08, 0x04, 0x40, 0x08, 0x22, 0x00, 0x19, 0x02, 0x00, 0x00, 0x80,
247       0x2c, 0x02, 0x02, 0x21, 0x01, 0x90, 0x20, 0x40, 0x00, 0x0c, 0x00, 0x34,
248       0x48, 0x58, 0x20, 0x01, 0x43, 0x04, 0x20, 0x80, 0x14, 0x00, 0x90, 0x00,
249       0x6d, 0x11, 0x00, 0x00, 0x40, 0x20, 0x00, 0x03, 0x10, 0x40, 0x88, 0x30,
250       0x05, 0x4a, 0x00, 0x65, 0x10, 0x24, 0x08, 0x18, 0x84, 0x28, 0x03, 0x80,
251       0x20, 0x42, 0xb0, 0x40, 0x00, 0x10, 0x69, 0x19, 0x04, 0x00, 0x00, 0x80,
252       0x04, 0xc2, 0x04, 0x00, 0x01, 0x00, 0x05, 0x00, 0x22, 0x25, 0x08, 0x96,
253       0x04, 0x02, 0x22, 0x00, 0xd0, 0x10, 0x29, 0x01, 0xa0, 0x60, 0x08, 0x10,
254       0x04, 0x01, 0x16, 0x44, 0x10, 0x02, 0x28, 0x02, 0x82, 0x48, 0x40, 0x84,
255       0x20, 0x90, 0x22, 0x28, 0x80, 0x04, 0x00, 0x40, 0x04, 0x24, 0x00, 0x80,
256       0x29, 0x03, 0x10, 0x60, 0x48, 0x00, 0x00, 0x81, 0xa0, 0x00, 0x51, 0x20,
257       0x0c, 0xd1, 0x00, 0x01, 0x41, 0x20, 0x04, 0x92, 0x00, 0x00, 0x10, 0x92,
258       0x00, 0x42, 0x04, 0x05, 0x01, 0x86, 0x40, 0x80, 0x10, 0x20, 0x52, 0x20,
259       0x21, 0x00, 0x10, 0x48, 0x0a, 0x02, 0x00, 0xd0, 0x12, 0x41, 0x48, 0x80,
260       0x04, 0x00, 0x00, 0x48, 0x09, 0x22, 0x04, 0x00, 0x24, 0x00, 0x43, 0x10,
261       0x60, 0x0a, 0x00, 0x44, 0x12, 0x20, 0x2c, 0x08, 0x20, 0x44, 0x00, 0x84,
262       0x09, 0x40, 0x06, 0x08, 0xc1, 0x00, 0x40, 0x80, 0x20, 0x00, 0x98, 0x12,
263       0x48, 0x10, 0xa2, 0x20, 0x00, 0x84, 0x48, 0xc0, 0x10, 0x20, 0x90, 0x12,
264       0x08, 0x98, 0x82, 0x00, 0x0a, 0xa0, 0x04, 0x03, 0x00, 0x28, 0xc3, 0x00,
265       0x44, 0x42, 0x10, 0x04, 0x08, 0x04, 0x40, 0x00, 0x00, 0x05, 0x10, 0x00,
266       0x21, 0x03, 0x80, 0x04, 0x88, 0x12, 0x69, 0x10, 0x00, 0x04, 0x08, 0x04,
267       0x04, 0x02, 0x84, 0x48, 0x49, 0x04, 0x20, 0x18, 0x02, 0x64, 0x80, 0x30,
268       0x08, 0x01, 0x02, 0x00, 0x52, 0x12, 0x49, 0x08, 0x20, 0x41, 0x88, 0x10,
269       0x48, 0x08, 0x34, 0x00, 0x01, 0x86, 0x05, 0xd0, 0x00, 0x00, 0x83, 0x84,
270       0x21, 0x40, 0x02, 0x41, 0x10, 0x80, 0x48, 0x40, 0xa2, 0x20, 0x51, 0x00,
271       0x00, 0x49, 0x00, 0x01, 0x90, 0x20, 0x40, 0x18, 0x02, 0x40, 0x02, 0x22,
272       0x05, 0x40, 0x80, 0x08, 0x82, 0x10, 0x20, 0x18, 0x00, 0x05, 0x01, 0x82,
273       0x40, 0x58, 0x00, 0x04, 0x81, 0x90, 0x29, 0x01, 0xa0, 0x64, 0x00, 0x22,
274       0x40, 0x01, 0xa2, 0x00, 0x18, 0x04, 0x0d, 0x00, 0x00, 0x60, 0x80, 0x94,
275       0x60, 0x82, 0x10, 0x0d, 0x80, 0x30, 0x0c, 0x12, 0x20, 0x00, 0x00, 0x12,
276       0x40, 0xc0, 0x20, 0x21, 0x58, 0x02, 0x41, 0x10, 0x80, 0x44, 0x03, 0x02,
277       0x04, 0x13, 0x90, 0x29, 0x08, 0x00, 0x44, 0xc0, 0x00, 0x21, 0x00, 0x26,
278       0x00, 0x1a, 0x80, 0x01, 0x13, 0x14, 0x20, 0x0a, 0x14, 0x20, 0x00, 0x32,
279       0x61, 0x08, 0x00, 0x40, 0x42, 0x20, 0x09, 0x80, 0x06, 0x01, 0x81, 0x80,
280       0x60, 0x42, 0x00, 0x68, 0x90, 0x82, 0x08, 0x42, 0x80, 0x04, 0x02, 0x80,
281       0x09, 0x0b, 0x04, 0x00, 0x98, 0x00, 0x0c, 0x81, 0x06, 0x44, 0x48, 0x84,
282       0x28, 0x03, 0x92, 0x00, 0x01, 0x80, 0x40, 0x0a, 0x00, 0x0c, 0x81, 0x02,
283       0x08, 0x51, 0x04, 0x28, 0x90, 0x02, 0x20, 0x09, 0x10, 0x60, 0x00, 0x00,
284       0x09, 0x81, 0xa0, 0x0c, 0x00, 0xa4, 0x09, 0x00, 0x02, 0x28, 0x80, 0x20,
285       0x00, 0x02, 0x02, 0x04, 0x81, 0x14, 0x04, 0x00, 0x04, 0x09, 0x11, 0x12,
286       0x60, 0x40, 0x20, 0x01, 0x48, 0x30, 0x40, 0x11, 0x00, 0x08, 0x0a, 0x86,
287       0x00, 0x00, 0x04, 0x60, 0x81, 0x04, 0x01, 0xd0, 0x02, 0x41, 0x18, 0x90,
288       0x00, 0x0a, 0x20, 0x00, 0xc1, 0x06, 0x01, 0x08, 0x80, 0x64, 0xca, 0x10,
289       0x04, 0x99, 0x80, 0x48, 0x01, 0x82, 0x20, 0x50, 0x90, 0x48, 0x80, 0x84,
290       0x20, 0x90, 0x22, 0x00, 0x19, 0x00, 0x04, 0x18, 0x20, 0x24, 0x10, 0x86,
291       0x40, 0xc2, 0x00, 0x24, 0x12, 0x10, 0x44, 0x00, 0x16, 0x08, 0x10, 0x24,
292       0x00, 0x12, 0x06, 0x01, 0x08, 0x90, 0x00, 0x12, 0x02, 0x4d, 0x10, 0x80,
293       0x40, 0x50, 0x22, 0x00, 0x43, 0x10, 0x01, 0x00, 0x30, 0x21, 0x0a, 0x00,
294       0x00, 0x01, 0x14, 0x00, 0x10, 0x84, 0x04, 0xc1, 0x10, 0x29, 0x0a, 0x00,
295       0x01, 0x8a, 0x00, 0x20, 0x01, 0x12, 0x0c, 0x49, 0x20, 0x04, 0x81, 0x00,
296       0x48, 0x01, 0x04, 0x60, 0x80, 0x12, 0x0c, 0x08, 0x10, 0x48, 0x4a, 0x04,
297       0x28, 0x10, 0x00, 0x28, 0x40, 0x84, 0x45, 0x50, 0x10, 0x60, 0x10, 0x06,
```

```
298         0x44, 0x01, 0x80, 0x09, 0x00, 0x86, 0x01, 0x42, 0xa0, 0x00, 0x90, 0x00,
299         0x05, 0x90, 0x22, 0x40, 0x41, 0x00, 0x08, 0x80, 0x02, 0x08, 0xc0, 0x00,
300         0x01, 0x58, 0x30, 0x49, 0x09, 0x14, 0x00, 0x41, 0x02, 0x0c, 0x02, 0x80,
301         0x40, 0x89, 0x00, 0x24, 0x08, 0x10, 0x05, 0x90, 0x32, 0x40, 0x0a, 0x82,
302         0x08, 0x00, 0x12, 0x61, 0x00, 0x04, 0x21, 0x00, 0x22, 0x04, 0x10, 0x24,
303         0x08, 0x0a, 0x04, 0x01, 0x10, 0x00, 0x20, 0x40, 0x84, 0x04, 0x88, 0x22,
304         0x20, 0x90, 0x12, 0x00, 0x53, 0x06, 0x24, 0x01, 0x04, 0x40, 0x0b, 0x14,
305         0x60, 0x82, 0x02, 0x0d, 0x10, 0x90, 0x0c, 0x08, 0x20, 0x09, 0x00, 0x14,
306         0x09, 0x80, 0x80, 0x24, 0x82, 0x00, 0x40, 0x01, 0x02, 0x44, 0x01, 0x20,
307         0x0c, 0x40, 0x84, 0x40, 0x0a, 0x10, 0x41, 0x00, 0x30, 0x05, 0x09, 0x80,
308         0x44, 0x08, 0x20, 0x20, 0x02, 0x00, 0x49, 0x43, 0x20, 0x21, 0x00, 0x20,
309         0x00, 0x01, 0xb6, 0x08, 0x40, 0x04, 0x08, 0x02, 0x80, 0x01, 0x41, 0x80,
310         0x40, 0x08, 0x10, 0x24, 0x00, 0x20, 0x04, 0x12, 0x86, 0x09, 0xc0, 0x12,
311         0x21, 0x81, 0x14, 0x04, 0x00, 0x02, 0x20, 0x89, 0xb4, 0x44, 0x12, 0x80,
312         0x00, 0xd1, 0x00, 0x69, 0x40, 0x80, 0x00, 0x42, 0x12, 0x00, 0x18, 0x04,
313         0x00, 0x49, 0x06, 0x21, 0x02, 0x04, 0x28, 0x02, 0x84, 0x01, 0xc0, 0x10,
314         0x68, 0x00, 0x20, 0x08, 0x40, 0x00, 0x08, 0x91, 0x10, 0x01, 0x81, 0x24,
315         0x04, 0xd2, 0x10, 0x4c, 0x88, 0x86, 0x00, 0x10, 0x80, 0x0c, 0x02, 0x14,
316         0x00, 0x8a, 0x90, 0x40, 0x18, 0x20, 0x21, 0x80, 0xa4, 0x00, 0x58, 0x24,
317         0x20, 0x10, 0x10, 0x60, 0xc1, 0x30, 0x41, 0x48, 0x02, 0x48, 0x09, 0x00,
318         0x40, 0x09, 0x02, 0x05, 0x11, 0x82, 0x20, 0x4a, 0x20, 0x24, 0x18, 0x02,
319         0x0c, 0x10, 0x22, 0x0c, 0x0a, 0x04, 0x00, 0x03, 0x06, 0x48, 0x48, 0x04,
320         0x04, 0x02, 0x00, 0x21, 0x80, 0x84, 0x00, 0x18, 0x00, 0x0c, 0x02, 0x12,
321         0x01, 0x00, 0x14, 0x05, 0x82, 0x10, 0x41, 0x89, 0x12, 0x08, 0x40, 0xa4,
322         0x21, 0x01, 0x84, 0x48, 0x02, 0x10, 0x60, 0x40, 0x02, 0x28, 0x00, 0x14,
323         0x08, 0x40, 0xa0, 0x20, 0x51, 0x12, 0x00, 0xc2, 0x00, 0x01, 0x1a, 0x30,
324         0x40, 0x89, 0x12, 0x4c, 0x02, 0x80, 0x00, 0x00, 0x14, 0x01, 0x01, 0xa0,
325         0x21, 0x18, 0x22, 0x21, 0x18, 0x06, 0x40, 0x01, 0x80, 0x00, 0x90, 0x04,
326         0x48, 0x02, 0x30, 0x04, 0x08, 0x00, 0x05, 0x88, 0x24, 0x08, 0x48, 0x04,
327         0x24, 0x02, 0x06, 0x00, 0x80, 0x00, 0x00, 0x00, 0x10, 0x65, 0x11, 0x90,
328         0x00, 0x0a, 0x82, 0x04, 0xc3, 0x04, 0x60, 0x48, 0x24, 0x04, 0x92, 0x02,
329         0x44, 0x88, 0x80, 0x40, 0x18, 0x06, 0x29, 0x80, 0x10, 0x01, 0x00, 0x00,
330         0x44, 0xc8, 0x10, 0x21, 0x89, 0x30, 0x00, 0x4b, 0xa0, 0x01, 0x10, 0x14,
331         0x00, 0x02, 0x94, 0x40, 0x00, 0x20, 0x65, 0x00, 0xa2, 0x0c, 0x40, 0x22,
332         0x20, 0x81, 0x12, 0x20, 0x82, 0x04, 0x01, 0x10, 0x00, 0x08, 0x88, 0x00,
333         0x00, 0x11, 0x80, 0x04, 0x42, 0x80, 0x40, 0x41, 0x14, 0x00, 0x40, 0x32,
334         0x2c, 0x80, 0x24, 0x04, 0x19, 0x00, 0x00, 0x91, 0x00, 0x20, 0x83, 0x00,
335         0x05, 0x40, 0x20, 0x09, 0x01, 0x84, 0x40, 0x40, 0x20, 0x20, 0x11, 0x00,
336         0x40, 0x41, 0x90, 0x20, 0x00, 0x00, 0x40, 0x90, 0x92, 0x48, 0x18, 0x06,
337         0x08, 0x81, 0x80, 0x48, 0x01, 0x34, 0x24, 0x10, 0x20, 0x04, 0x00, 0x20,
338         0x04, 0x18, 0x06, 0x2d, 0x90, 0x10, 0x01, 0x00, 0x90, 0x00, 0x0a, 0x22,
339         0x01, 0x00, 0x22, 0x00, 0x11, 0x84, 0x01, 0x01, 0x00, 0x20, 0x88, 0x00,
340         0x44, 0x00, 0x22, 0x01, 0x00, 0xa6, 0x40, 0x02, 0x06, 0x20, 0x11, 0x00,
341         0x01, 0xc8, 0xa0, 0x04, 0x8a, 0x00, 0x28, 0x19, 0x80, 0x00, 0x52, 0xa0,
342         0x24, 0x12, 0x12, 0x09, 0x08, 0x24, 0x01, 0x48, 0x00, 0x04, 0x00, 0x24,
343         0x40, 0x02, 0x84, 0x08, 0x00, 0x04, 0x48, 0x40, 0x90, 0x60, 0x0a, 0x22,
344         0x01, 0x88, 0x14, 0x08, 0x01, 0x02, 0x08, 0xd3, 0x00, 0x20, 0xc0, 0x90,
345         0x24, 0x10, 0x00, 0x00, 0x01, 0xb0, 0x08, 0x0a, 0xa0, 0x00, 0x80, 0x00,
346         0x01, 0x09, 0x00, 0x20, 0x52, 0x02, 0x25, 0x00, 0x24, 0x04, 0x02, 0x84,
347         0x24, 0x10, 0x92, 0x40, 0x02, 0xa0, 0x40, 0x00, 0x22, 0x08, 0x11, 0x04,
348         0x08, 0x01, 0x22, 0x00, 0x42, 0x14, 0x00, 0x09, 0x90, 0x21, 0x00, 0x30,
349         0x6c, 0x00, 0x00, 0x0c, 0x00, 0x22, 0x09, 0x90, 0x10, 0x28, 0x40, 0x00,
350         0x20, 0xc0, 0x20, 0x00, 0x90, 0x00, 0x40, 0x01, 0x82, 0x05, 0x12, 0x12,
351         0x09, 0xc1, 0x04, 0x61, 0x80, 0x02, 0x28, 0x81, 0x24, 0x00, 0x49, 0x04,
352         0x08, 0x10, 0x86, 0x29, 0x41, 0x80, 0x21, 0x0a, 0x30, 0x49, 0x88, 0x90,
353         0x00, 0x41, 0x04, 0x29, 0x81, 0x80, 0x41, 0x09, 0x00, 0x40, 0x12, 0x10,
354         0x40, 0x00, 0x10, 0x40, 0x48, 0x02, 0x05, 0x80, 0x02, 0x21, 0x40, 0x20,
355         0x00, 0x58, 0x20, 0x60, 0x00, 0x90, 0x48, 0x00, 0x80, 0x28, 0xc0, 0x80,
356         0x48, 0x00, 0x00, 0x44, 0x80, 0x02, 0x00, 0x09, 0x06, 0x00, 0x12, 0x02,
357         0x01, 0x00, 0x10, 0x08, 0x83, 0x10, 0x45, 0x12, 0x00, 0x2c, 0x08, 0x04,
358         0x44, 0x00, 0x20, 0x20, 0xc0, 0x10, 0x20, 0x01, 0x00, 0x05, 0xc8, 0x20,
359         0x04, 0x98, 0x10, 0x08, 0x10, 0x00, 0x24, 0x02, 0x16, 0x40, 0x88, 0x00,
```

```
360      0x61, 0x88, 0x12, 0x24, 0x80, 0xa6, 0x00, 0x42, 0x00, 0x08, 0x10, 0x06,
361      0x48, 0x40, 0xa0, 0x00, 0x50, 0x20, 0x04, 0x81, 0xa4, 0x40, 0x18, 0x00,
362      0x08, 0x10, 0x80, 0x01, 0x01};
363
364  #if RSA_KEY_SIEVE && SIMULATION && RSA_INSTRUMENT
365  UINT32  PrimeIndex = 0;
366  UINT32  failedAtIteration[10] = {0};
367  UINT32  PrimeCounts[3] = {0};
368  UINT32  MillerRabinTrials[3] = {0};
369  UINT32  totalFieldsSieved[3] = {0};
370  UINT32  bitsInFieldAfterSieve[3] = {0};
371  UINT32  emptyFieldsSieved[3] = {0};
372  UINT32  noPrimeFields[3] = {0};
373  UINT32  primesChecked[3] = {0};
374  UINT16  lastSievePrime = 0;
375  #endif
```

10.2.21 RsaKeyCache.c

10.2.21.1 Introduction

This file contains the functions to implement the RSA key cache that can be used to speed up simulation.

Only one key is created for each supported key size and it is returned whenever a key of that size is requested.

If desired, the key cache can be populated from a file. This allows multiple TPM to run with the same RSA keys. Also, when doing simulation, the DRBG will use preset sequences so it is not too hard to repeat sequences for debug or profile or stress.

When the key cache is enabled, a call to CryptRsaGenerateKey() will call the GetCachedRsaKey(). If the cache is enabled and populated, then the cached key of the requested size is returned. If a key of the requested size is not available, the no key is loaded and the requested key will need to be generated. If the cache is not populated, the TPM will open a file that has the appropriate name for the type of keys required (CRT or no-CRT). If the file is the right size, it is used. If the file doesn't exist or the file does not have the correct size, the TMP will populate the cache with new keys of the required size and write the cache data to the file so that they will be available the next time.

Currently, if two simulations are being run with TPM's that have different RSA key sizes (e.g,, one with 1024 and 2048 and another with 2048 and 3072, then the files will not match for the both of them and they will both try to overwrite the other's cache file. I may try to do something about this if necessary.

10.2.21.2 Includes, Types, Locals, and Defines

```
1   #include "Tpm.h"
2   #if USE_RSA_KEY_CACHE
3   #include  <stdio.h>
4   #include "RsaKeyCache_fp.h"
5   #if CRT_FORMAT_RSA == YES
6   #define CACHE_FILE_NAME "RsaKeyCacheCrt.data"
7   #else
8   #define CACHE_FILE_NAME "RsaKeyCacheNoCrt.data"
9   #endif
10  typedef struct _RSA_KEY_CACHE_
11  {
12      TPM2B_PUBLIC_KEY_RSA        publicModulus;
13      TPM2B_PRIVATE_KEY_RSA       privateExponent;
14  } RSA_KEY_CACHE;
```

Determine the number of RSA key sizes for the cache

```
15  TPMI_RSA_KEY_BITS       SupportedRsaKeySizes[] = {
16  #if RSA_1024
17      1024,
18  #endif
19  #if RSA_2048
20      2048,
21  #endif
22  #if RSA_3072
23      3072,
24  #endif
25  #if RSA_4096
26      4096,
27  #endif
28      0
29  };
30
31  #define RSA_KEY_CACHE_ENTRIES (RSA_1024 + RSA_2048 + RSA_3072 + RSA_4096)
```


 Copyright © TCG 2006-2020

The key cache holds one entry for each of the supported key sizes

```
32  RSA_KEY_CACHE          s_rsaKeyCache[RSA_KEY_CACHE_ENTRIES];
33  // Indicates if the key cache is loaded. It can be loaded and enabled or disabled.
34  BOOL                   s_keyCacheLoaded = 0;
35
36  // Indicates if the key cache is enabled
37  int                    s_rsaKeyCacheEnabled = FALSE;
38
39  //*** RsaKeyCacheControl()
40  // Used to enable and disable the RSA key cache.
41  LIB_EXPORT void
42  RsaKeyCacheControl(
43      int                state
44      )
45  {
46      s_rsaKeyCacheEnabled = state;
47  }
```

10.2.21.2.1 InitializeKeyCache()

This will initialize the key cache and attempt to write it to a file for later use.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
48  static BOOL
49  InitializeKeyCache(
50      TPMT_PUBLIC         *publicArea,
51      TPMT_SENSITIVE      *sensitive,
52      RAND_STATE          *rand               // IN: if not NULL, the deterministic
53                                              //     RNG state
54      )
55  {
56      int                  index;
57      TPM_KEY_BITS         keySave = publicArea->parameters.rsaDetail.keyBits;
58      BOOL                 OK = TRUE;
59  //
60      s_rsaKeyCacheEnabled = FALSE;
61      for(index = 0; OK && index < RSA_KEY_CACHE_ENTRIES; index++)
62      {
63          publicArea->parameters.rsaDetail.keyBits
64              = SupportedRsaKeySizes[index];
65          OK = (CryptRsaGenerateKey(publicArea, sensitive, rand) == TPM_RC_SUCCESS);
66          if(OK)
67          {
68              s_rsaKeyCache[index].publicModulus = publicArea->unique.rsa;
69              s_rsaKeyCache[index].privateExponent = sensitive->sensitive.rsa;
70          }
71      }
72      publicArea->parameters.rsaDetail.keyBits = keySave;
73      s_keyCacheLoaded = OK;
74  #if SIMULATION && USE_RSA_KEY_CACHE && USE_KEY_CACHE_FILE
75      if(OK)
76      {
77          FILE                *cacheFile;
78          const char          *fn = CACHE_FILE_NAME;
79
80  #if defined _MSC_VER
81          if(fopen_s(&cacheFile, fn, "w+b") != 0)
82  #else
```

```
83          cacheFile = fopen(fn, "w+b");
84          if(NULL == cacheFile)
85  #endif
86          {
87              printf("Can't open %s for write.\n", fn);
88          }
89          else
90          {
91              fseek(cacheFile, 0, SEEK_SET);
92              if(fwrite(s_rsaKeyCache, 1, sizeof(s_rsaKeyCache), cacheFile)
93                 != sizeof(s_rsaKeyCache))
94              {
95                  printf("Error writing cache to %s.", fn);
96              }
97          }
98          if(cacheFile)
99              fclose(cacheFile);
100     }
101 #endif
102     return s_keyCacheLoaded;
103 }
```

10.2.21.2.2 KeyCacheLoaded()

Checks that key cache is loaded.

Return Value	Meaning
TRUE(1)	cache loaded
FALSE(0)	cache not loaded

```
104 static BOOL
105 KeyCacheLoaded(
106     TPMT_PUBLIC         *publicArea,
107     TPMT_SENSITIVE      *sensitive,
108     RAND_STATE          *rand               // IN: if not NULL, the deterministic
109                                             //     RNG state
110     )
111 {
112 #if SIMULATION && USE_RSA_KEY_CACHE && USE_KEY_CACHE_FILE
113     if(!s_keyCacheLoaded)
114     {
115         FILE            *cacheFile;
116         const char *     fn = CACHE_FILE_NAME;
117 #if defined _MSC_VER && 1
118         if(fopen_s(&cacheFile, fn, "r+b") == 0)
119 #else
120         cacheFile = fopen(fn, "r+b");
121         if(NULL != cacheFile)
122 #endif
123         {
124             fseek(cacheFile, 0L, SEEK_END);
125             if(ftell(cacheFile) == sizeof(s_rsaKeyCache))
126             {
127                 fseek(cacheFile, 0L, SEEK_SET);
128                 s_keyCacheLoaded = (
129                     fread(&s_rsaKeyCache, 1, sizeof(s_rsaKeyCache), cacheFile)
130                     == sizeof(s_rsaKeyCache));
131             }
132             fclose(cacheFile);
133         }
134     }
135 #endif
```

```
136     if(!s_keyCacheLoaded)
137         s_rsaKeyCacheEnabled = InitializeKeyCache(publicArea, sensitive, rand);
138     return s_keyCacheLoaded;
139 }
```

10.2.21.2.3 GetCachedRsaKey()

Return Value	Meaning
TRUE(1)	key loaded
FALSE(0)	key not loaded

```
140 BOOL
141 GetCachedRsaKey(
142     TPMT_PUBLIC         *publicArea,
143     TPMT_SENSITIVE      *sensitive,
144     RAND_STATE          *rand               // IN: if not NULL, the deterministic
145                                             //     RNG state
146     )
147 {
148     int                      keyBits = publicArea->parameters.rsaDetail.keyBits;
149     int                      index;
150 //
151     if(KeyCacheLoaded(publicArea, sensitive, rand))
152     {
153         for(index = 0; index < RSA_KEY_CACHE_ENTRIES; index++)
154         {
155             if((s_rsaKeyCache[index].publicModulus.t.size * 8) == keyBits)
156             {
157                 publicArea->unique.rsa = s_rsaKeyCache[index].publicModulus;
158                 sensitive->sensitive.rsa = s_rsaKeyCache[index].privateExponent;
159                 return TRUE;
160             }
161         }
162         return FALSE;
163     }
164     return s_keyCacheLoaded;
165 }
166 #endif  // defined SIMULATION && defined USE_RSA_KEY_CACHE
```

10.2.22 Ticket.c

10.2.22.1 Introduction

This clause contains the functions used for ticket computations.

10.2.22.2 Includes

```
1   #include "Tpm.h"
```

10.2.22.3 Functions

10.2.22.3.1 TicketIsSafe()

This function indicates if producing a ticket is safe. It checks if the leading bytes of an input buffer is TPM_GENERATED_VALUE or its substring of canonical form. If so, it is not safe to produce ticket for an input buffer claiming to be TPM generated buffer

Return Value	Meaning
TRUE(1)	safe to produce ticket
FALSE(0)	not safe to produce ticket

```
2   BOOL
3   TicketIsSafe(
4       TPM2B           *buffer
5       )
6   {
7       TPM_GENERATED   valueToCompare = TPM_GENERATED_VALUE;
8       BYTE            bufferToCompare[sizeof(valueToCompare)];
9       BYTE            *marshalBuffer;
10  //
11      // If the buffer size is less than the size of TPM_GENERATED_VALUE, assume
12      // it is not safe to generate a ticket
13      if(buffer->size < sizeof(valueToCompare))
14          return FALSE;
15      marshalBuffer = bufferToCompare;
16      TPM_GENERATED_Marshal(&valueToCompare, &marshalBuffer, NULL);
17      if(MemoryEqual(buffer->buffer, bufferToCompare, sizeof(valueToCompare)))
18          return FALSE;
19      else
20          return TRUE;
21  }
```

10.2.22.3.2 TicketComputeVerified()

This function creates a TPMT_TK_VERIFIED ticket.

```
22  void
23  TicketComputeVerified(
24      TPMI_RH_HIERARCHY    hierarchy,     // IN: hierarchy constant for ticket
25      TPM2B_DIGEST        *digest,        // IN: digest
26      TPM2B_NAME          *keyName,       // IN: name of key that signed the values
27      TPMT_TK_VERIFIED    *ticket         // OUT: verified ticket
28      )
29  {
30      TPM2B_PROOF         *proof;
31      HMAC_STATE           hmacState;
```


 Copyright © TCG 2006-2020

```
32  //
33      // Fill in ticket fields
34      ticket->tag = TPM_ST_VERIFIED;
35      ticket->hierarchy = hierarchy;
36      proof = HierarchyGetProof(hierarchy);
37
38      // Start HMAC using the proof value of the hierarchy as the HMAC key
39      ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
40                                               &proof->b);
41          //  TPM_ST_VERIFIED
42      CryptDigestUpdateInt(&hmacState, sizeof(TPM_ST), ticket->tag);
43          //  digest
44      CryptDigestUpdate2B(&hmacState.hashState, &digest->b);
45          // key name
46      CryptDigestUpdate2B(&hmacState.hashState, &keyName->b);
47          // done
48      CryptHmacEnd2B(&hmacState, &ticket->digest.b);
49
50      return;
51  }
```

10.2.22.3.3 TicketComputeAuth()

This function creates a TPMT_TK_AUTH ticket.

```
52  void
53  TicketComputeAuth(
54      TPM_ST               type,          // IN: the type of ticket.
55      TPMI_RH_HIERARCHY    hierarchy,     // IN: hierarchy constant for ticket
56      UINT64               timeout,       // IN: timeout
57      BOOL                 expiresOnReset,// IN: flag to indicate if ticket expires on
58                                          //      TPM Reset
59      TPM2B_DIGEST        *cpHashA,       // IN: input cpHashA
60      TPM2B_NONCE         *policyRef,     // IN: input policyRef
61      TPM2B_NAME          *entityName,    // IN: name of entity
62      TPMT_TK_AUTH        *ticket         // OUT: Created ticket
63      )
64  {
65      TPM2B_PROOF         *proof;
66      HMAC_STATE           hmacState;
67  //
68      // Get proper proof
69      proof = HierarchyGetProof(hierarchy);
70
71      // Fill in ticket fields
72      ticket->tag = type;
73      ticket->hierarchy = hierarchy;
74
75      // Start HMAC with hierarchy proof as the HMAC key
76      ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
77                                               &proof->b);
78          //  TPM_ST_AUTH_SECRET or TPM_ST_AUTH_SIGNED,
79      CryptDigestUpdateInt(&hmacState, sizeof(UINT16), ticket->tag);
80      // cpHash
81      CryptDigestUpdate2B(&hmacState.hashState, &cpHashA->b);
82          //  policyRef
83      CryptDigestUpdate2B(&hmacState.hashState, &policyRef->b);
84          //  keyName
85      CryptDigestUpdate2B(&hmacState.hashState, &entityName->b);
86          //  timeout
87      CryptDigestUpdateInt(&hmacState, sizeof(timeout), timeout);
88      if(timeout != 0)
89      {
90              //  epoch
```

 Copyright © TCG 2006-2020

```
91          CryptDigestUpdateInt(&hmacState.hashState, sizeof(CLOCK_NONCE),
92                               g_timeEpoch);
93              // reset count
94          if(expiresOnReset)
95              CryptDigestUpdateInt(&hmacState.hashState, sizeof(gp.totalResetCount),
96                                   gp.totalResetCount);
97      }
98          // done
99      CryptHmacEnd2B(&hmacState, &ticket->digest.b);
100
101     return;
102 }
```

10.2.22.3.4 TicketComputeHashCheck()

This function creates a TPMT_TK_HASHCHECK ticket.

```
103 void
104 TicketComputeHashCheck(
105     TPMI_RH_HIERARCHY    hierarchy,     // IN: hierarchy constant for ticket
106     TPM_ALG_ID           hashAlg,       // IN: the hash algorithm for 'digest'
107     TPM2B_DIGEST        *digest,        // IN: input digest
108     TPMT_TK_HASHCHECK   *ticket         // OUT: Created ticket
109     )
110 {
111     TPM2B_PROOF         *proof;
112     HMAC_STATE           hmacState;
113 //
114     // Get proper proof
115     proof = HierarchyGetProof(hierarchy);
116
117     // Fill in ticket fields
118     ticket->tag = TPM_ST_HASHCHECK;
119     ticket->hierarchy = hierarchy;
120
121     // Start HMAC using hierarchy proof as HMAC key
122     ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
123                                              &proof->b);
124         //  TPM_ST_HASHCHECK
125     CryptDigestUpdateInt(&hmacState, sizeof(TPM_ST), ticket->tag);
126         //  hash algorithm
127     CryptDigestUpdateInt(&hmacState, sizeof(hashAlg), hashAlg);
128         //  digest
129     CryptDigestUpdate2B(&hmacState.hashState, &digest->b);
130         // done
131     CryptHmacEnd2B(&hmacState, &ticket->digest.b);
132
133     return;
134 }
```

10.2.22.3.5 TicketComputeCreation()

This function creates a TPMT_TK_CREATION ticket.

```
135 void
136 TicketComputeCreation(
137     TPMI_RH_HIERARCHY    hierarchy,     // IN: hierarchy for ticket
138     TPM2B_NAME          *name,          // IN: object name
139     TPM2B_DIGEST        *creation,      // IN: creation hash
140     TPMT_TK_CREATION    *ticket         // OUT: created ticket
141     )
142 {
143     TPM2B_PROOF         *proof;
```

```
144     HMAC_STATE           hmacState;
145
146     // Get proper proof
147     proof = HierarchyGetProof(hierarchy);
148
149     // Fill in ticket fields
150     ticket->tag = TPM_ST_CREATION;
151     ticket->hierarchy = hierarchy;
152
153     // Start HMAC using hierarchy proof as HMAC key
154     ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
155                                              &proof->b);
156         //  TPM_ST_CREATION
157     CryptDigestUpdateInt(&hmacState, sizeof(TPM_ST), ticket->tag);
158         //  name if provided
159     if(name != NULL)
160         CryptDigestUpdate2B(&hmacState.hashState, &name->b);
161         //  creation hash
162     CryptDigestUpdate2B(&hmacState.hashState, &creation->b);
163         // Done
164     CryptHmacEnd2B(&hmacState, &ticket->digest.b);
165
166     return;
167 }
```

 Copyright © TCG 2006-2020

10.2.23 TpmAsn1.c

10.2.23.1 Includes

```
1   #include "Tpm.h"
2   #define _OIDS_
3   #include "OIDs.h"
4   #include "TpmASN1.h"
5   #include "TpmASN1_fp.h"
```

10.2.23.2 Unmarshaling Functions

10.2.23.2.1 ASN1UnmarshalContextInitialize()

Function does standard initialization of a context.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
6   BOOL
7   ASN1UnmarshalContextInitialize(
8       ASN1UnmarshalContext    *ctx,
9       INT16                    size,
10      BYTE                    *buffer
11  )
12  {
13      VERIFY(buffer != NULL);
14      VERIFY(size > 0);
15      ctx->buffer = buffer;
16      ctx->size = size;
17      ctx->offset = 0;
18      ctx->tag = 0xFF;
19      return TRUE;
20  Error:
21      return FALSE;
22  }
```

10.2.23.2.2 ASN1DecodeLength()

This function extracts the length of an element from *buffer* starting at *offset*.

Return Value	Meaning
>=0	the extracted length
<0	an error

```
23  INT16
24  ASN1DecodeLength(
25      ASN1UnmarshalContext        *ctx
26  )
27  {
28      BYTE                first;                  // Next octet in buffer
29      INT16               value;
30  //
31      VERIFY(ctx->offset < ctx->size);
32      first = NEXT_OCTET(ctx);
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
33      // If the number of octets of the entity is larger than 127, then the first octet
34      // is the number of octets in the length specifier.
35      if(first >= 0x80)
36      {
37          // Make sure that this length field is contained with the structure being
38          // parsed
39          CHECK_SIZE(ctx, (first & 0x7F));
40          if(first == 0x82)
41          {
42              // Two octets of size
43              // get the next value
44              value = (INT16)NEXT_OCTET(ctx);
45              // Make sure that the result will fit in an INT16
46              VERIFY(value < 0x0080);
47              // Shift up and add next octet
48              value = (value << 8) + NEXT_OCTET(ctx);
49          }
50          else if(first == 0x81)
51              value = NEXT_OCTET(ctx);
52          // Sizes larger than will fit in a INT16 are an error
53          else
54              goto Error;
55      }
56      else
57          value = first;
58      // Make sure that the size defined something within the current context
59      CHECK_SIZE(ctx, value);
60      return value;
61  Error:
62      ctx->size = -1;              // Makes everything fail from now on.
63      return -1;
64  }
```

10.2.23.2.3 ASN1NextTag()

This function extracts the next type from *buffer* starting at *offset*. It advances *offset* as it parses the type and the length of the type. It returns the length of the type. On return, the *length* octets starting at *offset* are the octets of the type.

Return Value	Meaning
>=0	the number of octets in *type*
<0	an error

```
65  INT16
66  ASN1NextTag(
67      ASN1UnmarshalContext    *ctx
68  )
69  {
70      // A tag to get?
71      VERIFY(ctx->offset < ctx->size);
72      // Get it
73      ctx->tag = NEXT_OCTET(ctx);
74      // Make sure that it is not an extended tag
75      VERIFY((ctx->tag & 0x1F) != 0x1F);
76      // Get the length field and return that
77      return ASN1DecodeLength(ctx);
78
79  Error:
80      // Attempt to read beyond the end of the context or an illegal tag
81      ctx->size = -1;          // Persistent failure
82      ctx->tag = 0xFF;
83      return -1;
```

 Copyright © TCG 2006-2020

```
84     }
```

10.2.23.2.4 ASN1GetBitStringValue()

Try to parse a bit string of up to 32 bits from a value that is expected to be a bit string. The bit string is left justified so that the MSb of the input is the MSb of the returned value. If there is a general parsing error, the context->size is set to -1.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
85     BOOL
86     ASN1GetBitStringValue(
87         ASN1UnmarshalContext        *ctx,
88         UINT32                      *val
89     )
90     {
91         int                     shift;
92         INT16                   length;
93         UINT32                  value = 0;
94         int                     inputBits;
95     //
96         length = ASN1NextTag(ctx);
97         VERIFY(length >= 1);
98         VERIFY(ctx->tag == ASN1_BITSTRING);
99         // Get the shift value for the bit field (how many bits to lop off of the end)
100        shift = NEXT_OCTET(ctx);
101        length--;
102        // Get the number of bits in the input
103        inputBits = (8 * length) - shift;
104        // the shift count has to make sense
105        VERIFY((shift < 8) && ((length > 0) || (shift == 0)));
106        // if there are any bytes left
107        for(; length > 1; length--)
108        {
109
110            // for all but the last octet, just shift and add the new octet
111            VERIFY((value & 0xFF000000) == 0); // can't loose significant bits
112            value = (value << 8) + NEXT_OCTET(ctx);
113
114        }
115        if(length == 1)
116        {
117            // for the last octet, just shift the accumulated value enough to
118            // accept the significant bits in the last octet and shift the last
119            // octet down
120            VERIFY(((value & (0xFF000000 << (8 - shift)))) == 0);
121            value = (value << (8 - shift)) + (NEXT_OCTET(ctx) >> shift);
122
123        }
124        // 'Left justify' the result
125        if(inputBits > 0)
126            value <<= (32 - inputBits);
127        *val = value;
128        return TRUE;
129    Error:
130        ctx->size = -1;
131        return FALSE;
132    }
```

10.2.23.3 Marshaling Functions

10.2.23.3.1 Introduction

Marshaling of an ASN.1 structure is accomplished from the bottom up. That is, the things that will be at the end of the structure are added last. To manage the collecting of the relative sizes, start a context for the outermost container, if there is one, and then placing items in from the bottom up. If the bottom-most item is also within a structure, create a nested context by calling ASN1StartMarshalingContext().

The context control structure contains a *buffer* pointer, an *offset*, an *end* and a stack. *offset* is the offset from the start of the buffer of the last added byte. When *offset* reaches 0, the buffer is full. *offset* is a signed value so that, when it becomes negative, there is an overflow. Only two functions are allowed to move bytes into the buffer: ASN1PushByte() and ASN1PushBytes(). These functions make sure that no data is written beyond the end of the buffer.

When a new context is started, the current value of *end* is pushed on the stack and *end* is set to 'offset. As bytes are added, offset gets smaller. At any time, the count of bytes in the current context is simply *end* - *offset*.

Since starting a new context involves setting *end* = *offset*, the number of bytes in the context starts at 0. The nominal way of ending a context is to use *end* - *offset* to set the length value, and then a tag is added to the buffer. Then the previous *end* value is popped meaning that the context just ended becomes a member of the now current context.

The nominal strategy for building a completed ASN.1 structure is to push everything into the buffer and then move everything to the start of the buffer. The move is simple as the size of the move is the initial *end* value minus the final *offset* value. The destination is *buffer* and the source is *buffer* + *offset*. As Skippy would say "Easy peasy, Joe."

It is not necessary to provide a buffer into which the data is placed. If no buffer is provided, then the marshaling process will return values needed for marshaling. On strategy for filling the buffer would be to execute the process for building the structure without using a buffer. This would return the overall size of the structure. Then that amount of data could be allocated for the buffer and the fill process executed again with the data going into the buffer. At the end, the data would be in its final resting place.

10.2.23.3.2 ASN1InitialializeMarshalContext()

This creates a structure for handling marshaling of an ASN.1 formatted data structure.

```
133 void
134 ASN1InitialializeMarshalContext(
135     ASN1MarshalContext      *ctx,
136     INT16                    length,
137     BYTE                    *buffer
138 )
139 {
140     ctx->buffer = buffer;
141     if(buffer)
142         ctx->offset = length;
143     else
144         ctx->offset = INT16_MAX;
145     ctx->end = ctx->offset;
146     ctx->depth = -1;
147 }
```

10.2.23.3.3 ASN1StartMarshalContext()

This starts a new constructed element. It is constructed on *top* of the value that was previously placed in the structure.

 Copyright © TCG 2006-2020

```
148 void
149 ASN1StartMarshalContext(
150     ASN1MarshalContext      *ctx
151 )
152 {
153     pAssert((ctx->depth + 1) < MAX_DEPTH);
154     ctx->depth++;
155     ctx->ends[ctx->depth] = ctx->end;
156     ctx->end = ctx->offset;
157 }
```

10.2.23.3.4 ASN1EndMarshalContext()

This function restores the end pointer for an encapsulating structure.

Return Value	Meaning
0	the size of the encapsulated structure that was just ended
0	an error

```
158 INT16
159 ASN1EndMarshalContext(
160     ASN1MarshalContext      *ctx
161 )
162 {
163     INT16                   length;
164     pAssert(ctx->depth >= 0);
165     length = ctx->end - ctx->offset;
166     ctx->end = ctx->ends[ctx->depth--];
167     if((ctx->depth == -1) && (ctx->buffer))
168     {
169         MemoryCopy(ctx->buffer, ctx->buffer + ctx->offset, ctx->end - ctx->offset);
170     }
171     return length;
172 }
```

10.2.23.3.5 ASN1EndEncapsulation()

This function puts a tag and length in the buffer. In this function, an embedded BIT_STRING is assumed to be a collection of octets. To indicate that all bits are used, a byte of zero is prepended. If a raw bit-string is needed, a new function like ASN1PushInteger() would be needed.

Return Value	Meaning
0	number of octets in the encapsulation
0	failure

```
173 UINT16
174 ASN1EndEncapsulation(
175     ASN1MarshalContext          *ctx,
176     BYTE                         tag
177 )
178 {
179     // only add a leading zero for an encapsulated BIT STRING
180     if (tag == ASN1_BITSTRING)
181         ASN1PushByte(ctx, 0);
182     ASN1PushTagAndLength(ctx, tag, ctx->end - ctx->offset);
183     return ASN1EndMarshalContext(ctx);
184 }
```

 Copyright © TCG 2006-2020

10.2.23.3.6 ASN1PushByte()

```
185 BOOL
186 ASN1PushByte(
187     ASN1MarshalContext          *ctx,
188     BYTE                         b
189 )
190 {
191     if(ctx->offset > 0)
192     {
193         ctx->offset -= 1;
194         if(ctx->buffer)
195             ctx->buffer[ctx->offset] = b;
196         return TRUE;
197     }
198     ctx->offset = -1;
199     return FALSE;
200 }
```

10.2.23.3.7 ASN1PushBytes()

Push some raw bytes onto the buffer. *count* cannot be zero.

Return Value	Meaning
0	count bytes
0	failure unless count was zero

```
201 INT16
202 ASN1PushBytes(
203     ASN1MarshalContext          *ctx,
204     INT16                        count,
205     const BYTE                  *buffer
206 )
207 {
208     // make sure that count is not negative which would mess up the math; and that
209     // if there is a count, there is a buffer
210     VERIFY((count >= 0) && ((buffer != NULL) || (count == 0)));
211     // back up the offset to determine where the new octets will get pushed
212     ctx->offset -= count;
213     // can't go negative
214     VERIFY(ctx->offset >= 0);
215     // if there are buffers, move the data, otherwise, assume that this is just a
216     // test.
217     if(count && buffer && ctx->buffer)
218         MemoryCopy(&ctx->buffer[ctx->offset], buffer, count);
219     return count;
220 Error:
221     ctx->offset = -1;
222     return 0;
223 }
```

10.2.23.3.8 ASN1PushNull()

Return Value	Meaning
0	count bytes
0	failure unless count was zero

```
224 INT16
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
225 ASN1PushNull(
226     ASN1MarshalContext      *ctx
227 )
228 {
229     ASN1PushByte(ctx, 0);
230     ASN1PushByte(ctx, ASN1_NULL);
231     return (ctx->offset >= 0) ? 2 : 0;
232 }
```

10.2.23.3.9 ASN1PushLength()

Push a length value. This will only handle length values that fit in an INT16.

Return Value	Meaning
0	number of bytes added
0	failure

```
233 INT16
234 ASN1PushLength(
235     ASN1MarshalContext          *ctx,
236     INT16                        len
237 )
238 {
239     UINT16                       start = ctx->offset;
240     VERIFY(len >= 0);
241     if(len <= 127)
242         ASN1PushByte(ctx, (BYTE)len);
243     else
244     {
245         ASN1PushByte(ctx, (BYTE)(len & 0xFF));
246         len >>= 8;
247         if(len == 0)
248             ASN1PushByte(ctx, 0x81);
249         else
250         {
251             ASN1PushByte(ctx, (BYTE)(len));
252             ASN1PushByte(ctx, 0x82);
253         }
254     }
255     goto Exit;
256 Error:
257     ctx->offset = -1;
258 Exit:
259     return (ctx->offset > 0) ? start - ctx->offset : 0;
260 }
```

10.2.23.3.10 ASN1PushTagAndLength()

Return Value	Meaning
0	number of bytes added
0	failure

```
261 INT16
262 ASN1PushTagAndLength(
263     ASN1MarshalContext          *ctx,
264     BYTE                         tag,
265     INT16                        length
266 )
267 {
```

```
268     INT16       bytes;
269     bytes = ASN1PushLength(ctx, length);
270     bytes += (INT16)ASN1PushByte(ctx, tag);
271     return (ctx->offset < 0) ? 0 : bytes;
272 }
```

10.2.23.3.11 ASN1PushTaggedOctetString()

This function will push a random octet string.

Return Value	Meaning
0	number of bytes added
0	failure

```
273 INT16
274 ASN1PushTaggedOctetString(
275     ASN1MarshalContext          *ctx,
276     INT16                        size,
277     const BYTE                  *string,
278     BYTE                         tag
279 )
280 {
281     ASN1PushBytes(ctx, size, string);
282     // PushTagAndLenght just tells how many octets it added so the total size of this
283     // element is the sum of those octets and input size.
284     size += ASN1PushTagAndLength(ctx, tag, size);
285     return size;
286 }
```

10.2.23.3.12 ASN1PushUINT()

This function pushes an native-endian integer value. This just changes a native-endian integer into a big-endian byte string and calls ASN1PushInteger(). That function will remove leading zeros and make sure that the number is positive.

Return Value	Meaning
0	count bytes
0	failure unless count was zero

```
287 INT16
288 ASN1PushUINT(
289     ASN1MarshalContext      *ctx,
290     UINT32                   integer
291 )
292 {
293     BYTE                    marshaled[4];
294     UINT32_TO_BYTE_ARRAY(integer, marshaled);
295     return ASN1PushInteger(ctx, 4, marshaled);
296 }
```

10.2.23.3.13 ASN1PushInteger

Push a big-endian integer on the end of the buffer

 Copyright © TCG 2006-2020

Return Value	Meaning
0	the number of bytes marshaled for the integer
0	failure

```
297     INT16
298     ASN1PushInteger(
299         ASN1MarshalContext  *ctx,           // IN/OUT: buffer context
300         INT16                iLen,          // IN: octets of the integer
301         BYTE                *integer        // IN: big-endian integer
302     )
303     {
304         // no leading 0's
305         while((*integer == 0) && (--iLen > 0))
306             integer++;
307         // Move the bytes to the buffer
308         ASN1PushBytes(ctx, iLen, integer);
309         // if needed, add a leading byte of 0 to make the number positive
310         if(*integer & 0x80)
311             iLen += (INT16)ASN1PushByte(ctx, 0);
312         // PushTagAndLenght just tells how many octets it added so the total size of this
313         // element is the sum of those octets and the adjusted input size.
314         iLen +=  ASN1PushTagAndLength(ctx, ASN1_INTEGER, iLen);
315         return iLen;
316     }
```

10.2.23.3.14 ASN1PushOID()

This function is used to add an OID. An OID is 0x06 followed by a byte of size followed by size bytes. This is used to avoid having to do anything special in the definition of an OID.

Return Value	Meaning
0	the number of bytes marshaled for the integer
0	failure

```
317     INT16
318     ASN1PushOID(
319         ASN1MarshalContext          *ctx,
320         const BYTE                  *OID
321     )
322     {
323         if((*OID == ASN1_OBJECT_IDENTIFIER) && ((OID[1] & 0x80) == 0))
324         {
325             return ASN1PushBytes(ctx, OID[1] + 2, OID);
326         }
327         ctx->offset = -1;
328         return 0;
329     }
```

10.2.24 X509_ECC.c

10.2.24.1 Includes

```
1  #include "Tpm.h"
2  #include "X509.h"
3  #include "OIDs.h"
4  #include "TpmASN1_fp.h"
5  #include "X509_spt_fp.h"
6  #include "CryptHash_fp.h"
```

10.2.24.2 Functions

10.2.24.2.1 X509PushPoint()

This seems like it might be used more than once so...

Return Value	Meaning
0	number of bytes added
0	failure

```
7   INT16
8   X509PushPoint(
9       ASN1MarshalContext      *ctx,
10      TPMS_ECC_POINT          *p
11  )
12  {
13      // Push a bit string containing the public key. For now, push the x, and y
14      // coordinates of the public point, bottom up
15      ASN1StartMarshalContext(ctx); // BIT STRING
16      {
17          ASN1PushBytes(ctx, p->y.t.size, p->y.t.buffer);
18          ASN1PushBytes(ctx, p->x.t.size, p->x.t.buffer);
19          ASN1PushByte(ctx, 0x04);
20      }
21      return ASN1EndEncapsulation(ctx, ASN1_BITSTRING); // Ends BIT STRING
22  }
```

10.2.24.2.2 X509AddSigningAlgorithmECC()

This creates the singing algorithm data.

Return Value	Meaning
0	number of bytes added
0	failure

```
23  INT16
24  X509AddSigningAlgorithmECC(
25      OBJECT              *signKey,
26      TPMT_SIG_SCHEME     *scheme,
27      ASN1MarshalContext  *ctx
28  )
29  {
30      PHASH_DEF            hashDef = CryptGetHashDef(scheme->details.any.hashAlg);
31  //
32      NOT_REFERENCED(signKey);
```

 Copyright © TCG 2006-2020

```
33      // If the desired hashAlg definition wasn't found...
34      if(hashDef->hashAlg != scheme->details.any.hashAlg)
35          return 0;
36
37      switch(scheme->scheme)
38      {
39          case ALG_ECDSA_VALUE:
40              // Make sure that we have an OID for this hash and ECC
41              if((hashDef->ECDSA)[0] != ASN1_OBJECT_IDENTIFIER)
42                  break;
43              // if this is just an implementation check, indicate that this
44              // combination is supported
45              if(!ctx)
46                  return 1;
47              ASN1StartMarshalContext(ctx);
48              ASN1PushOID(ctx, hashDef->ECDSA);
49              return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
50          default:
51              break;
52      }
53      return 0;
54  }
```

10.2.24.2.3 X509AddPublicECC()

This function will add the *publicKey* description to the DER data. If ctx is NULL, then no data is transferred and this function will indicate if the TPM has the values for DER-encoding of the public key.

Return Value	Meaning
0	number of bytes added
0	failure

```
55  INT16
56  X509AddPublicECC(
57      OBJECT                  *object,
58      ASN1MarshalContext      *ctx
59  )
60  {
61      const BYTE      *curveOid =
62          CryptEccGetOID(object->publicArea.parameters.eccDetail.curveID);
63      if((curveOid == NULL) || (*curveOid != ASN1_OBJECT_IDENTIFIER))
64          return 0;
65  //
66  //
67  //  SEQUENCE (2 elem) 1st
68  //    SEQUENCE (2 elem) 2nd
69  //      OBJECT IDENTIFIER 1.2.840.10045.2.1 ecPublicKey (ANSI X9.62 public key type)
70  //      OBJECT IDENTIFIER 1.2.840.10045.3.1.7 prime256v1 (ANSI X9.62 named curve)
71  //    BIT STRING (520 bit) 000001001010000111010101010111001001101101000100000010...
72  //
73      // If this is a check to see if the key can be encoded, it can.
74      // Need to mark the end sequence
75      if(ctx == NULL)
76          return 1;
77      ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 1st
78      {
79          X509PushPoint(ctx, &object->publicArea.unique.ecc); // BIT STRING
80          ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 2nd
81          {
82              ASN1PushOID(ctx, curveOid); // curve dependent
83              ASN1PushOID(ctx, OID_ECC_PUBLIC); // (1.2.840.10045.2.1)
84          }
```

 Copyright © TCG 2006-2020

```
85          ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE); // Ends SEQUENCE 2nd
86      }
87      return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE); // Ends SEQUENCE 1st
88  }
```

 Copyright © TCG 2006-2020

10.2.25 X509_RSA.c

10.2.25.1 Includes

```
1   #include "Tpm.h"
2   #include "X509.h"
3   #include "TpmASN1_fp.h"
4   #include "X509_spt_fp.h"
5   #include "CryptHash_fp.h"
6   #include "CryptRsa_fp.h"
```

10.2.25.2 Functions

```
7   #if ALG_RSA
```

10.2.25.2.1 X509AddSigningAlgorithmRSA()

This creates the singing algorithm data.

Return Value	Meaning
0	number of bytes added
0	failure

```
8   INT16
9   X509AddSigningAlgorithmRSA(
10      OBJECT               *signKey,
11      TPMT_SIG_SCHEME      *scheme,
12      ASN1MarshalContext   *ctx
13  )
14  {
15      TPM_ALG_ID            hashAlg = scheme->details.any.hashAlg;
16      PHASH_DEF             hashDef = CryptGetHashDef(hashAlg);
17  //
18      NOT_REFERENCED(signKey);
19      // return failure if hash isn't implemented
20      if(hashDef->hashAlg != hashAlg)
21          return 0;
22      switch(scheme->scheme)
23      {
24          case ALG_RSASSA_VALUE:
25          {
26              // if the hash is implemented but there is no PKCS1 OID defined
27              // then this is not a valid signing combination.
28              if(hashDef->PKCS1[0] != ASN1_OBJECT_IDENTIFIER)
29                  break;
30              if(ctx == NULL)
31                  return 1;
32              return X509PushAlgorithmIdentifierSequence(ctx, hashDef->PKCS1);
33          }
34          case ALG_RSAPSS_VALUE:
35              // leave if this is just an implementation check
36              if(ctx == NULL)
37                  return 1;
38              // In the case of SHA1, everything is default and RFC4055 says that
39              // implementations that do signature generation MUST omit the parameter
40              // when defaults are used. )-:
41              if(hashDef->hashAlg == ALG_SHA1_VALUE)
42              {
43                  return X509PushAlgorithmIdentifierSequence(ctx, OID_RSAPSS);
44              }
```

 Copyright © TCG 2006-2020

```
45              else
46              {
47                  // Going to build something that looks like:
48                  //   SEQUENCE (2 elem)
49                  //      OBJECT IDENTIFIER 1.2.840.113549.1.1.10 rsaPSS (PKCS #1)
50                  //      SEQUENCE (3 elem)
51                  //        [0] (1 elem)
52                  //          SEQUENCE (2 elem)
53                  //            OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256
54                  //            NULL
55                  //        [1] (1 elem)
56                  //          SEQUENCE (2 elem)
57                  //            OBJECT IDENTIFIER 1.2.840.113549.1.1.8 pkcs1-MGF
58                  //            SEQUENCE (2 elem)
59                  //              OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256
60                  //              NULL
61                  //        [2] (1 elem)  salt length
62                  //          INTEGER 32
63
64                  // The indentation is just to keep track of where we are in the
65                  // structure
66                  ASN1StartMarshalContext(ctx); // SEQUENCE (2 elements)
67                  {
68                      ASN1StartMarshalContext(ctx);   // SEQUENCE (3 elements)
69                      {
70                          // [2] (1 elem)  salt length
71                          //    INTEGER 32
72                          ASN1StartMarshalContext(ctx);
73                          {
74                              INT16       saltSize =
75                                  CryptRsaPssSaltSize((INT16)hashDef->digestSize,
76                                  (INT16)signKey->publicArea.unique.rsa.t.size);
77                              ASN1PushUINT(ctx, saltSize);
78                          }
79                          ASN1EndEncapsulation(ctx, ASN1_APPLICAIION_SPECIFIC + 2);
80
81                          // Add the mask generation algorithm
82                          // [1] (1 elem)
83                          //    SEQUENCE (2 elem) 1st
84                          //      OBJECT IDENTIFIER 1.2.840.113549.1.1.8 pkcs1-MGF
85                          //      SEQUENCE (2 elem) 2nd
86                          //        OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256
87                          //        NULL
88                          ASN1StartMarshalContext(ctx);   // mask context [1] (1 elem)
89                          {
90                              ASN1StartMarshalContext(ctx);   // SEQUENCE (2 elem) 1st
91                              // Handle the 2nd Sequence (sequence (object, null))
92                              {
93                                  // This adds a NULL, then an OID and a SEQUENCE
94                                  // wrapper.
95                                  X509PushAlgorithmIdentifierSequence(ctx,
96                                      hashDef->OID);
97                                  // add the pkcs1-MGF OID
98                                  ASN1PushOID(ctx, OID_MGF1);
99                              }
100                             // End outer sequence
101                             ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
102                         }
103                         // End the [1]
104                         ASN1EndEncapsulation(ctx, ASN1_APPLICAIION_SPECIFIC + 1);
105
106                         // Add the hash algorithm
107                         // [0] (1 elem)
108                         //   SEQUENCE (2 elem) (done by
109                         //              X509PushAlgorithmIdentifierSequence)
110                         //     OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256 (NIST)
```

 Copyright © TCG 2006-2020

```
111                         //     NULL
112                         ASN1StartMarshalContext(ctx); // [0] (1 elem)
113                         {
114                             X509PushAlgorithmIdentifierSequence(ctx, hashDef->OID);
115                         }
116                         ASN1EndEncapsulation(ctx, (ASN1_APPLICAIION_SPECIFIC + 0));
117                     }
118                     //  SEQUENCE (3 elements) end
119                     ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
120
121                     // RSA PSS OID
122                     // OBJECT IDENTIFIER 1.2.840.113549.1.1.10 rsaPSS (PKCS #1)
123                     ASN1PushOID(ctx, OID_RSAPSS);
124                 }
125                 // End Sequence (2 elements)
126                 return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
127             }
128         default:
129             break;
130     }
131     return 0;
132 }
```

10.2.25.2.2 X509AddPublicRSA()

This function will add the *publicKey* description to the DER data. If *fillPtr* is NULL, then no data is transferred and this function will indicate if the TPM has the values for DER-encoding of the public key.

Return Value	Meaning
0	number of bytes added
0	failure

```
133 INT16
134 X509AddPublicRSA(
135     OBJECT                  *object,
136     ASN1MarshalContext     *ctx
137 )
138 {
139     UINT32          exp = object->publicArea.parameters.rsaDetail.exponent;
140 //
141 /*
142     SEQUENCE (2 elem) 1st
143       SEQUENCE (2 elem) 2nd
144         OBJECT IDENTIFIER 1.2.840.113549.1.1.1 rsaEncryption (PKCS #1)
145         NULL
146       BIT STRING (1 elem)
147         SEQUENCE (2 elem) 3rd
148           INTEGER (2048 bit) 2197304513741227955725834199357401
149           INTEGER 65537
150 */
151     // If this is a check to see if the key can be encoded, it can.
152     // Need to mark the end sequence
153     if(ctx == NULL)
154         return 1;
155     ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 1st
156     ASN1StartMarshalContext(ctx); // BIT STRING
157     ASN1StartMarshalContext(ctx); // SEQUENCE *(2 elem) 3rd
158
159     // Get public exponent in big-endian byte order.
160     if(exp == 0)
161         exp = RSA_DEFAULT_PUBLIC_EXPONENT;
162
```

```
163     // Push a 4 byte integer. This might get reduced if there are leading zeros or
164     // extended if the high order byte is negative.
165     ASN1PushUINT(ctx, exp);
166     // Push the public key as an integer
167     ASN1PushInteger(ctx, object->publicArea.unique.rsa.t.size,
168                          object->publicArea.unique.rsa.t.buffer);
169     // Embed this in a SEQUENCE tag and length in for the key, exponent sequence
170     ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE); // SEQUENCE (3rd)
171
172     // Embed this in a BIT STRING
173     ASN1EndEncapsulation(ctx, ASN1_BITSTRING);
174
175     // Now add the formatted SEQUENCE for the RSA public key OID. This is a
176     // fully constructed value so it doesn't need to have a context started
177     X509PushAlgorithmIdentifierSequence(ctx, OID_PKCS1_PUB);
178
179     return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
180 }
181 #endif // ALG_RSA
```

10.2.26 X509_spt.c

10.2.26.1 Includes

```
1  #include "Tpm.h"
2  #include "TpmASN1.h"
3  #include "TpmASN1_fp.h"
4  #define _X509_SPT_
5  #include "X509.h"
6  #include "X509_spt_fp.h"
7  #if ALG_RSA
8  #   include "X509_RSA_fp.h"
9  #endif // ALG_RSA
10 #if ALG_ECC
11 #   include "X509_ECC_fp.h"
12 #endif // ALG_ECC
13 #if ALG_SM2
14 //#   include "X509_SM2_fp.h"
15 #endif // ALG_RSA
```

10.2.26.2 Unmarshaling Functions

10.2.26.2.1 X509FindExtensionByOID()

This will search a list of X509 extensions to find an extension with the requested OID. If the extension is found, the output context (*ctx*) is set up to point to the OID in the extension.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure (could be catastrophic)

```
16 BOOL
17 X509FindExtensionByOID(
18     ASN1UnmarshalContext    *ctxIn,         // IN: the context to search
19     ASN1UnmarshalContext    *ctx,           // OUT: the extension context
20     const BYTE              *OID            // IN: oid to search for
21 )
22 {
23     INT16                length;
24 //
25     pAssert(ctxIn != NULL);
26     // Make the search non-destructive of the input if ctx provided. Otherwise, use
27     // the provided context.
28     if (ctx == NULL)
29         ctx = ctxIn;
30     // if the provide search context is different from the context of the extension,
31     // then copy the search context to the search context.
32     else if(ctx != ctxIn)
33         *ctx = *ctxIn;
34     // Now, search in the extension context
35     for(;ctx->size > ctx->offset; ctx->offset += length)
36     {
37         VERIFY((length = ASN1NextTag(ctx)) >= 0);
38         // If this is not a constructed sequence, then it doesn't belong
39         // in the extensions.
40         VERIFY(ctx->tag == ASN1_CONSTRUCTED_SEQUENCE);
41         // Make sure that this entry could hold the OID
42         if (length >= OID_SIZE(OID))
43         {
44             // See if this is a match for the provided object identifier.
```


 Copyright © TCG 2006-2020

```
45              if (MemoryEqual(OID, &(ctx->buffer[ctx->offset]), OID_SIZE(OID)))
46              {
47                  // Return with ' ctx' set to point to the start of the OID with the
size
48                  // set to be the size of the SEQUENCE
49                  ctx->buffer += ctx->offset;
50                  ctx->offset = 0;
51                  ctx->size = length;
52                  return TRUE;
53              }
54          }
55      }
56      VERIFY(ctx->offset == ctx->size);
57      return FALSE;
58  Error:
59      ctxIn->size = -1;
60      ctx->size = -1;
61      return FALSE;
62  }
```

10.2.26.2.2 X509GetExtensionBits()

This function will extract a bit field from an extension. If the extension doesn't contain a bit string, it will fail.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
63  UINT32
64  X509GetExtensionBits(
65      ASN1UnmarshalContext            *ctx,
66      UINT32                          *value
67  )
68  {
69      INT16               length;
70  //
71      while(((length = ASN1NextTag(ctx)) > 0) && (ctx->size > ctx->offset))
72      {
73          // Since this is an extension, the extension value will be in an OCTET STRING
74          if(ctx->tag == ASN1_OCTET_STRING)
75          {
76              return ASN1GetBitStringValue(ctx, value);
77          }
78          ctx->offset += length;
79      }
80      ctx->size = -1;
81      return FALSE;
82  }
```

10.2.26.2.3 X509ProcessExtensions()

This function is used to process the TPMA_OBJECT and KeyUsage() extensions. It is not in the CertifyX509.c code because it makes the code harder to follow.

Error Returns	Meaning
TPM_RCS_ATTRIBUTES	the attributes of object are not consistent with the extension setting
TPM_RC_VALUE	problem parsing the extensions

```
83   TPM_RC
84   X509ProcessExtensions(
85       OBJECT              *object,        // IN: The object with the attributes to
86                                           //      check
87       stringRef           *extension      // IN: The start and length of the extensions
88   )
89   {
90       ASN1UnmarshalContext     ctx;
91       ASN1UnmarshalContext     extensionCtx;
92       INT16                    length;
93       UINT32                   value;
94       TPMA_OBJECT              attributes = object->publicArea.objectAttributes;
95   //
96       if(!ASN1UnmarshalContextInitialize(&ctx, extension->len, extension->buf)
97          || ((length = ASN1NextTag(&ctx)) < 0)
98          || (ctx.tag != X509_EXTENSIONS))
99           return TPM_RCS_VALUE;
100      if( ((length = ASN1NextTag(&ctx)) < 0)
101         || (ctx.tag != (ASN1_CONSTRUCTED_SEQUENCE)))
102          return TPM_RCS_VALUE;
103
104      // Get the extension for the TPMA_OBJECT if there is one
105      if(X509FindExtensionByOID(&ctx, &extensionCtx, OID_TCG_TPMA_OBJECT) &&
106          X509GetExtensionBits(&extensionCtx, &value))
107      {
108          // If an keyAttributes extension was found, it must be exactly the same as the
109          // attributes of the object.
110          // NOTE: MemoryEqual() is used rather than a simple UINT32 compare to avoid
111          // type-punned pointer warning/error.
112          if(!MemoryEqual(&value, &attributes, sizeof(value)))
113              return TPM_RCS_ATTRIBUTES;
114      }
115      // Make sure the failure to find the value wasn't because of a fatal error
116      else if(extensionCtx.size < 0)
117          return TPM_RCS_VALUE;
118
119      // Get the keyUsage extension. This one is required
120      if(X509FindExtensionByOID(&ctx, &extensionCtx, OID_KEY_USAGE_EXTENSION) &&
121          X509GetExtensionBits(&extensionCtx, &value))
122      {
123          x509KeyUsageUnion   keyUsage;
124          BOOL                bad;
125      //
126          keyUsage.integer = value;
127          // For KeyUsage:
128          // 1) 'sign' is SET if Key Usage includes signing
129          bad = (KEY_USAGE_SIGN.integer & keyUsage.integer) != 0
130                && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign);
131          // 2) 'decrypt' is SET if Key Usage includes decryption uses
132          bad = bad || (KEY_USAGE_DECRYPT.integer & keyUsage.integer) != 0
133                       && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt);
134          // 3) 'fixedTPM' is SET if Key Usage is non-repudiation
135          bad = bad || IS_ATTRIBUTE(keyUsage.x509, TPMA_X509_KEY_USAGE, nonrepudiation)
136                        && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM);
137          // 4)'restricted' is SET if Key Usage is for key agreement.
138          bad = bad || IS_ATTRIBUTE(keyUsage.x509, TPMA_X509_KEY_USAGE, keyAgreement)
139                       && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted);
140          if(bad)
141              return TPM_RCS_VALUE;
```

```
142         }
143         else
144             // The KeyUsage extension is required
145             return TPM_RCS_VALUE;
146
147         return TPM_RC_SUCCESS;
148     }
```

10.2.26.3 Marshaling Functions

10.2.26.3.1 X509AddSigningAlgorithm()

This creates the singing algorithm data.

Return Value	Meaning
0	number of octets added
0	failure

```
149     INT16
150     X509AddSigningAlgorithm(
151         ASN1MarshalContext  *ctx,
152         OBJECT              *signKey,
153         TPMT_SIG_SCHEME     *scheme
154     )
155     {
156         switch(signKey->publicArea.type)
157         {
158     #if ALG_RSA
159             case ALG_RSA_VALUE:
160                 return X509AddSigningAlgorithmRSA(signKey, scheme, ctx);
161     #endif // ALG_RSA
162     #if ALG_ECC
163             case ALG_ECC_VALUE:
164                 return X509AddSigningAlgorithmECC(signKey, scheme, ctx);
165     #endif // ALG_ECC
166     #if ALG_SM2
167             case ALG_SM2:
168                 return X509AddSigningAlgorithmSM2(signKey, scheme,ctx);
169     #endif // ALG_SM2
170             default:
171                 break;
172         }
173         return 0;
174     }
```

10.2.26.3.2 X509AddPublicKey()

This function will add the *publicKey* description to the DER data. If *fillPtr* is NULL, then no data is transferred and this function will indicate if the TPM has the values for DER-encoding of the public key.

Return Value	Meaning
0	number of octets added
0	failure

```
175     INT16
176     X509AddPublicKey(
177         ASN1MarshalContext  *ctx,
178         OBJECT              *object
```

```
179 )
180 {
181     switch(object->publicArea.type)
182     {
183 #if ALG_RSA
184         case ALG_RSA_VALUE:
185             return X509AddPublicRSA(object, ctx);
186 #endif
187 #if ALG_ECC
188         case ALG_ECC_VALUE:
189             return X509AddPublicECC(object, ctx);
190 #endif
191 #if ALG_SM2
192         case ALG_SM2_VALUE:
193             break;
194 #endif
195         default:
196             break;
197     }
198     return FALSE;
199 }
```

10.2.26.3.3 X509PushAlgorithmIdentifierSequence()

The function adds the algorithm identifier sequence.

Return Value	Meaning
0	number of bytes added
0	failure

```
200 INT16
201 X509PushAlgorithmIdentifierSequence(
202     ASN1MarshalContext          *ctx,
203     const BYTE                  *OID
204     )
205 {
206     ASN1StartMarshalContext(ctx);   // hash algorithm
207     ASN1PushNull(ctx);
208     ASN1PushOID(ctx, OID);
209     return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
210 }
```

10.2.27 AC_spt.c

10.2.27.1 Includes

```
1   #include "Tpm.h"
2   #include "AC_spt_fp.h"
3   #if 1 // This is the simulated AC data.
4   typedef struct {
5       TPMI_RH_AC              ac;
6       TPML_AC_CAPABILITIES   *acData;
7
8   } acCapabilities;
9   TPML_AC_CAPABILITIES acData0001 = {1,
10          {{TPM_AT_PV1, 0x01234567}}};
11
12  acCapabilities  ac[1] = { {0x0001, &acData0001} };
13
14  #define NUM_AC  (sizeof(ac) / sizeof(acCapabilities))
15  #endif // 1 The simulated AC data
```

10.2.27.1.1 AcToCapabilities()

This function returns a pointer to a list of AC capabilities.

```
16  TPML_AC_CAPABILITIES *
17  AcToCapabilities(
18      TPMI_RH_AC      component       // IN: component
19  )
20  {
21      UINT32          index;
22  //
23      for(index = 0; index < NUM_AC; index++)
24      {
25          if(ac[index].ac == component)
26              return ac[index].acData;
27      }
28      return NULL;
29  }
```

10.2.27.1.2 AcIsAccessible()

Function to determine if an AC handle references an actual AC

Return Value	Meaning

```
30  BOOL
31  AcIsAccessible(
32      TPM_HANDLE          acHandle
33      )
34  {
35      // In this implementation, the AC exists if there are some capabilities to go
36      // with the handle
37      return AcToCapabilities(acHandle) != NULL;
38  }
```

10.2.27.1.3 AcCapabilitiesGet()

This function returns a list of capabilities associated with an AC

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
39  TPMI_YES_NO
40  AcCapabilitiesGet(
41      TPMI_RH_AC              component,      // IN: the component
42      TPM_AT                  type,           // IN: start capability type
43      TPML_AC_CAPABILITIES   *capabilityList // OUT: list of handle
44  )
45  {
46      TPMI_YES_NO             more = NO;
47      UINT32                  i;
48      TPML_AC_CAPABILITIES   *capabilities = AcToCapabilities(component);
49
50      pAssert(HandleGetType(component) == TPM_HT_AC);
51
52      // Initialize output handle list
53      capabilityList->count = 0;
54
55      if(capabilities != NULL)
56      {
57          // Find the first capability less than or equal to type
58          for(i = 0; i < capabilities->count; i++)
59          {
60              if(capabilities->acCapabilities[i].tag >= type)
61              {
62                  // copy the capabilities until we run out or fill the list
63                  for(; (capabilityList->count < MAX_AC_CAPABILITIES)
64                      && (i < capabilities->count); i++)
65                  {
66                      capabilityList->acCapabilities[capabilityList->count]
67                          = capabilities->acCapabilities[i];
68                      capabilityList->count++;
69                  }
70                  more = i < capabilities->count;
71              }
72          }
73      }
74      return more;
75  }
```

10.2.27.1.4 AcSendObject()

Stub to handle sending of an AC object

Error Returns	Meaning

```
76  TPM_RC
77  AcSendObject(
78      TPM_HANDLE           acHandle,       // IN: Handle of AC receiving object
79      OBJECT              *object,         // IN: object structure to send
80      TPMS_AC_OUTPUT      *acDataOut       // OUT: results of operation
81  )
82  {
83      NOT_REFERENCED(object);
84      NOT_REFERENCED(acHandle);
85      acDataOut->tag = TPM_AT_ERROR;   // indicate that the response contains an
86                                       // error code
87      acDataOut->data = TPM_AE_NONE;   // but there is no error.
88
```

 Copyright © TCG 2006-2020

```
89          return TPM_RC_SUCCESS;
90      }
```

Annex A
(informative)
Implementation Dependent

A.1 Introduction

This header file contains definitions that are used to define a TPM profile. The values are chosen by the manufacturer. The values here are chosen to represent a full featured TPM so that all of the TPM's capabilities can be simulated and tested. This file would change based on the implementation.

The file listed below was generated by an automated tool using three documents as inputs. They are:

1) The TCG_Algorithm Registery,
2) Part 2 of this specification, and
3) A purpose-built document that contains vendor-specific information in tables.

All of the values in this file have #ifdef 'guards' so that they may be defined in a command line.Additionally, TpmBuildSwitches.h allows an additional file to be specified in the compiler command line and preset any of these values.

A.2 TpmProfile.h

```
1  #ifndef _TPM_PROFILE_H_
2  #define _TPM_PROFILE_H_
```

Table 2:4 - Defines for Logic Values

```
3  #undef TRUE
4  #define TRUE                1
5  #undef FALSE
6  #define FALSE               0
7  #undef YES
8  #define YES                 1
9  #undef NO
10 #define NO                  0
11 #undef SET
12 #define SET                 1
13 #undef CLEAR
14 #define CLEAR               0
```

Table 0:1 - Defines for Processor Values

```
15 #ifndef BIG_ENDIAN_TPM
16 #define BIG_ENDIAN_TPM              NO
17 #endif
18 #ifndef LITTLE_ENDIAN_TPM
19 #define LITTLE_ENDIAN_TPM           !BIG_ENDIAN_TPM
20 #endif
21 #ifndef MOST_SIGNIFICANT_BIT_0
22 #define MOST_SIGNIFICANT_BIT_0      NO
23 #endif
24 #ifndef LEAST_SIGNIFICANT_BIT_0
25 #define LEAST_SIGNIFICANT_BIT_0     !MOST_SIGNIFICANT_BIT_0
26 #endif
27 #ifndef AUTO_ALIGN
28 #define AUTO_ALIGN                  NO
29 #endif
```

Table 0:4 - Defines for Implemented Curves

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
30  #ifndef ECC_NIST_P192
31  #define ECC_NIST_P192                   NO
32  #endif
33  #ifndef ECC_NIST_P224
34  #define ECC_NIST_P224                   NO
35  #endif
36  #ifndef ECC_NIST_P256
37  #define ECC_NIST_P256                   YES
38  #endif
39  #ifndef ECC_NIST_P384
40  #define ECC_NIST_P384                   YES
41  #endif
42  #ifndef ECC_NIST_P521
43  #define ECC_NIST_P521                   NO
44  #endif
45  #ifndef ECC_BN_P256
46  #define ECC_BN_P256                     YES
47  #endif
48  #ifndef ECC_BN_P638
49  #define ECC_BN_P638                     NO
50  #endif
51  #ifndef ECC_SM2_P256
52  #define ECC_SM2_P256                    NO
53  #endif
```

Table 0:6 - Defines for Implemented ACT

```
54  #ifndef RH_ACT_0
55  #define RH_ACT_0                        YES
56  #endif
57  #ifndef RH_ACT_1
58  #define RH_ACT_1                        NO
59  #endif
60  #ifndef RH_ACT_A
61  #define RH_ACT_A                        YES
62  #endif
```

Table 0:7 - Defines for Implementation Values

```
63  #ifndef FIELD_UPGRADE_IMPLEMENTED
64  #define FIELD_UPGRADE_IMPLEMENTED       NO
65  #endif
66  #ifndef HASH_LIB
67  #define HASH_LIB                        Ossl
68  #endif
69  #ifndef SYM_LIB
70  #define SYM_LIB                         Ossl
71  #endif
72  #ifndef MATH_LIB
73  #define MATH_LIB                        Ossl
74  #endif
75  #ifndef IMPLEMENTATION_PCR
76  #define IMPLEMENTATION_PCR              24
77  #endif
78  #ifndef PLATFORM_PCR
79  #define PLATFORM_PCR                    24
80  #endif
81  #ifndef DRTM_PCR
82  #define DRTM_PCR                        17
83  #endif
84  #ifndef HCRTM_PCR
85  #define HCRTM_PCR                       0
86  #endif
87  #ifndef NUM_LOCALITIES
88  #define NUM_LOCALITIES                  5
```

```
 89  #endif
 90  #ifndef MAX_HANDLE_NUM
 91  #define MAX_HANDLE_NUM                    3
 92  #endif
 93  #ifndef MAX_ACTIVE_SESSIONS
 94  #define MAX_ACTIVE_SESSIONS               64
 95  #endif
 96  #ifndef CONTEXT_SLOT
 97  #define CONTEXT_SLOT                      UINT16
 98  #endif
 99  #ifndef MAX_LOADED_SESSIONS
100  #define MAX_LOADED_SESSIONS               3
101  #endif
102  #ifndef MAX_SESSION_NUM
103  #define MAX_SESSION_NUM                   3
104  #endif
105  #ifndef MAX_LOADED_OBJECTS
106  #define MAX_LOADED_OBJECTS                3
107  #endif
108  #ifndef MIN_EVICT_OBJECTS
109  #define MIN_EVICT_OBJECTS                 2
110  #endif
111  #ifndef NUM_POLICY_PCR_GROUP
112  #define NUM_POLICY_PCR_GROUP              1
113  #endif
114  #ifndef NUM_AUTHVALUE_PCR_GROUP
115  #define NUM_AUTHVALUE_PCR_GROUP           1
116  #endif
117  #ifndef MAX_CONTEXT_SIZE
118  #define MAX_CONTEXT_SIZE                  1264
119  #endif
120  #ifndef MAX_DIGEST_BUFFER
121  #define MAX_DIGEST_BUFFER                 1024
122  #endif
123  #ifndef MAX_NV_INDEX_SIZE
124  #define MAX_NV_INDEX_SIZE                 2048
125  #endif
126  #ifndef MAX_NV_BUFFER_SIZE
127  #define MAX_NV_BUFFER_SIZE                1024
128  #endif
129  #ifndef MAX_CAP_BUFFER
130  #define MAX_CAP_BUFFER                    1024
131  #endif
132  #ifndef NV_MEMORY_SIZE
133  #define NV_MEMORY_SIZE                    16384
134  #endif
135  #ifndef MIN_COUNTER_INDICES
136  #define MIN_COUNTER_INDICES               8
137  #endif
138  #ifndef NUM_STATIC_PCR
139  #define NUM_STATIC_PCR                    16
140  #endif
141  #ifndef MAX_ALG_LIST_SIZE
142  #define MAX_ALG_LIST_SIZE                 64
143  #endif
144  #ifndef PRIMARY_SEED_SIZE
145  #define PRIMARY_SEED_SIZE                 32
146  #endif
147  #ifndef CONTEXT_ENCRYPT_ALGORITHM
148  #define CONTEXT_ENCRYPT_ALGORITHM         AES
149  #endif
150  #ifndef NV_CLOCK_UPDATE_INTERVAL
151  #define NV_CLOCK_UPDATE_INTERVAL          12
152  #endif
153  #ifndef NUM_POLICY_PCR
154  #define NUM_POLICY_PCR                    1
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
155     #endif
156     #ifndef MAX_COMMAND_SIZE
157     #define MAX_COMMAND_SIZE                4096
158     #endif
159     #ifndef MAX_RESPONSE_SIZE
160     #define MAX_RESPONSE_SIZE               4096
161     #endif
162     #ifndef ORDERLY_BITS
163     #define ORDERLY_BITS                    8
164     #endif
165     #ifndef MAX_SYM_DATA
166     #define MAX_SYM_DATA                    128
167     #endif
168     #ifndef MAX_RNG_ENTROPY_SIZE
169     #define MAX_RNG_ENTROPY_SIZE            64
170     #endif
171     #ifndef RAM_INDEX_SPACE
172     #define RAM_INDEX_SPACE                 512
173     #endif
174     #ifndef RSA_DEFAULT_PUBLIC_EXPONENT
175     #define RSA_DEFAULT_PUBLIC_EXPONENT     0x00010001
176     #endif
177     #ifndef ENABLE_PCR_NO_INCREMENT
178     #define ENABLE_PCR_NO_INCREMENT         YES
179     #endif
180     #ifndef CRT_FORMAT_RSA
181     #define CRT_FORMAT_RSA                  YES
182     #endif
183     #ifndef VENDOR_COMMAND_COUNT
184     #define VENDOR_COMMAND_COUNT            0
185     #endif
186     #ifndef MAX_VENDOR_BUFFER_SIZE
187     #define MAX_VENDOR_BUFFER_SIZE          1024
188     #endif
189     #ifndef MAX_DERIVATION_BITS
190     #define MAX_DERIVATION_BITS             8192
191     #endif
192     #ifndef SIZE_OF_X509_SERIAL_NUMBER
193     #define SIZE_OF_X509_SERIAL_NUMBER      20
194     #endif
195     #ifndef PRIVATE_VENDOR_SPECIFIC_BYTES
196     #define PRIVATE_VENDOR_SPECIFIC_BYTES   RSA_PRIVATE_SIZE
197     #endif
```

Table 0:2 - Defines for Implemented Algorithms

```
198     #ifndef ALG_AES
199     #define ALG_AES                         ALG_YES
200     #endif
201     #ifndef ALG_CAMELLIA
202     #define ALG_CAMELLIA                    ALG_YES
203     #endif
204     #ifndef ALG_CBC
205     #define ALG_CBC                         ALG_YES
206     #endif
207     #ifndef ALG_CFB
208     #define ALG_CFB                         ALG_YES
209     #endif
210     #ifndef ALG_CMAC
211     #define ALG_CMAC                        ALG_YES
212     #endif
213     #ifndef ALG_CTR
214     #define ALG_CTR                         ALG_YES
215     #endif
216     #ifndef ALG_ECB
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
217     #define ALG_ECB                         ALG_YES
218     #endif
219     #ifndef ALG_ECC
220     #define ALG_ECC                         ALG_YES
221     #endif
222     #ifndef ALG_ECDAA
223     #define ALG_ECDAA                       (ALG_YES && ALG_ECC)
224     #endif
225     #ifndef ALG_ECDH
226     #define ALG_ECDH                        (ALG_YES && ALG_ECC)
227     #endif
228     #ifndef ALG_ECDSA
229     #define ALG_ECDSA                       (ALG_YES && ALG_ECC)
230     #endif
231     #ifndef ALG_ECMQV
232     #define ALG_ECMQV                       (ALG_NO && ALG_ECC)
233     #endif
234     #ifndef ALG_ECSCHNORR
235     #define ALG_ECSCHNORR                   (ALG_YES && ALG_ECC)
236     #endif
237     #ifndef ALG_HMAC
238     #define ALG_HMAC                        ALG_YES
239     #endif
240     #ifndef ALG_KDF1_SP800_108
241     #define ALG_KDF1_SP800_108              ALG_YES
242     #endif
243     #ifndef ALG_KDF1_SP800_56A
244     #define ALG_KDF1_SP800_56A              (ALG_YES && ALG_ECC)
245     #endif
246     #ifndef ALG_KDF2
247     #define ALG_KDF2                        ALG_NO
248     #endif
249     #ifndef ALG_KEYEDHASH
250     #define ALG_KEYEDHASH                   ALG_YES
251     #endif
252     #ifndef ALG_MGF1
253     #define ALG_MGF1                        ALG_YES
254     #endif
255     #ifndef ALG_OAEP
256     #define ALG_OAEP                        (ALG_YES && ALG_RSA)
257     #endif
258     #ifndef ALG_OFB
259     #define ALG_OFB                         ALG_YES
260     #endif
261     #ifndef ALG_RSA
262     #define ALG_RSA                         ALG_YES
263     #endif
264     #ifndef ALG_RSAES
265     #define ALG_RSAES                       (ALG_YES && ALG_RSA)
266     #endif
267     #ifndef ALG_RSAPSS
268     #define ALG_RSAPSS                      (ALG_YES && ALG_RSA)
269     #endif
270     #ifndef ALG_RSASSA
271     #define ALG_RSASSA                      (ALG_YES && ALG_RSA)
272     #endif
273     #ifndef ALG_SHA
274     #define ALG_SHA                         ALG_NO      /* Not specified by vendor */
275     #endif
276     #ifndef ALG_SHA1
277     #define ALG_SHA1                        ALG_YES
278     #endif
279     #ifndef ALG_SHA256
280     #define ALG_SHA256                      ALG_YES
281     #endif
282     #ifndef ALG_SHA384
```

```
283 #define ALG_SHA384                    ALG_YES
284 #endif
285 #ifndef ALG_SHA3_256
286 #define ALG_SHA3_256                  ALG_NO     /* Not specified by vendor */
287 #endif
288 #ifndef ALG_SHA3_384
289 #define ALG_SHA3_384                  ALG_NO     /* Not specified by vendor */
290 #endif
291 #ifndef ALG_SHA3_512
292 #define ALG_SHA3_512                  ALG_NO     /* Not specified by vendor */
293 #endif
294 #ifndef ALG_SHA512
295 #define ALG_SHA512                    ALG_NO
296 #endif
297 #ifndef ALG_SM2
298 #define ALG_SM2                       (ALG_NO && ALG_ECC)
299 #endif
300 #ifndef ALG_SM3_256
301 #define ALG_SM3_256                   ALG_NO
302 #endif
303 #ifndef ALG_SM4
304 #define ALG_SM4                       ALG_YES
305 #endif
306 #ifndef ALG_SYMCIPHER
307 #define ALG_SYMCIPHER                 ALG_YES
308 #endif
309 #ifndef ALG_TDES
310 #define ALG_TDES                      ALG_NO
311 #endif
312 #ifndef ALG_XOR
313 #define ALG_XOR                       ALG_YES
314 #endif
```

Table 1:3 - Defines for RSA Asymmetric Cipher Algorithm Constants

```
315 #ifndef RSA_1024
316 #define RSA_1024                  (ALG_RSA && YES)
317 #endif
318 #ifndef RSA_2048
319 #define RSA_2048                  (ALG_RSA && YES)
320 #endif
321 #ifndef RSA_3072
322 #define RSA_3072                  (ALG_RSA && NO)
323 #endif
324 #ifndef RSA_4096
325 #define RSA_4096                  (ALG_RSA && NO)
326 #endif
```

Table 1:21 - Defines for AES Symmetric Cipher Algorithm Constants

```
327 #ifndef AES_128
328 #define AES_128                   (ALG_AES && YES)
329 #endif
330 #ifndef AES_192
331 #define AES_192                   (ALG_AES && NO)
332 #endif
333 #ifndef AES_256
334 #define AES_256                   (ALG_AES && YES)
335 #endif
```

Table 1:22 - Defines for SM4 Symmetric Cipher Algorithm Constants

```
336 #ifndef SM4_128
337 #define SM4_128                   (ALG_SM4 && YES)
```

```
338     #endif
```

Table 1:23 - Defines for CAMELLIA Symmetric Cipher Algorithm Constants

```
339     #ifndef CAMELLIA_128
340     #define CAMELLIA_128                        (ALG_CAMELLIA && YES)
341     #endif
342     #ifndef CAMELLIA_192
343     #define CAMELLIA_192                        (ALG_CAMELLIA && NO)
344     #endif
345     #ifndef CAMELLIA_256
346     #define CAMELLIA_256                        (ALG_CAMELLIA && YES)
347     #endif
```

Table 1:24 - Defines for TDES Symmetric Cipher Algorithm Constants

```
348     #ifndef TDES_128
349     #define TDES_128                    (ALG_TDES && YES)
350     #endif
351     #ifndef TDES_192
352     #define TDES_192                    (ALG_TDES && YES)
353     #endif
```

Table 0:5 - Defines for Implemented Commands

```
354     #ifndef CC_ACT_SetTimeout
355     #define CC_ACT_SetTimeout                         CC_YES
356     #endif
357     #ifndef CC_AC_GetCapability
358     #define CC_AC_GetCapability                       CC_YES
359     #endif
360     #ifndef CC_AC_Send
361     #define CC_AC_Send                                CC_YES
362     #endif
363     #ifndef CC_ActivateCredential
364     #define CC_ActivateCredential                     CC_YES
365     #endif
366     #ifndef CC_Certify
367     #define CC_Certify                                CC_YES
368     #endif
369     #ifndef CC_CertifyCreation
370     #define CC_CertifyCreation                        CC_YES
371     #endif
372     #ifndef CC_CertifyX509
373     #define CC_CertifyX509                            CC_YES
374     #endif
375     #ifndef CC_ChangeEPS
376     #define CC_ChangeEPS                              CC_YES
377     #endif
378     #ifndef CC_ChangePPS
379     #define CC_ChangePPS                              CC_YES
380     #endif
381     #ifndef CC_Clear
382     #define CC_Clear                                  CC_YES
383     #endif
384     #ifndef CC_ClearControl
385     #define CC_ClearControl                           CC_YES
386     #endif
387     #ifndef CC_ClockRateAdjust
388     #define CC_ClockRateAdjust                        CC_YES
389     #endif
390     #ifndef CC_ClockSet
391     #define CC_ClockSet                               CC_YES
392     #endif
```

```
393 #ifndef CC_Commit
394 #define CC_Commit                          (CC_YES && ALG_ECC)
395 #endif
396 #ifndef CC_ContextLoad
397 #define CC_ContextLoad                     CC_YES
398 #endif
399 #ifndef CC_ContextSave
400 #define CC_ContextSave                     CC_YES
401 #endif
402 #ifndef CC_Create
403 #define CC_Create                          CC_YES
404 #endif
405 #ifndef CC_CreateLoaded
406 #define CC_CreateLoaded                    CC_YES
407 #endif
408 #ifndef CC_CreatePrimary
409 #define CC_CreatePrimary                   CC_YES
410 #endif
411 #ifndef CC_DictionaryAttackLockReset
412 #define CC_DictionaryAttackLockReset       CC_YES
413 #endif
414 #ifndef CC_DictionaryAttackParameters
415 #define CC_DictionaryAttackParameters      CC_YES
416 #endif
417 #ifndef CC_Duplicate
418 #define CC_Duplicate                       CC_YES
419 #endif
420 #ifndef CC_ECC_Parameters
421 #define CC_ECC_Parameters                  (CC_YES && ALG_ECC)
422 #endif
423 #ifndef CC_ECDH_KeyGen
424 #define CC_ECDH_KeyGen                     (CC_YES && ALG_ECC)
425 #endif
426 #ifndef CC_ECDH_ZGen
427 #define CC_ECDH_ZGen                       (CC_YES && ALG_ECC)
428 #endif
429 #ifndef CC_EC_Ephemeral
430 #define CC_EC_Ephemeral                    (CC_YES && ALG_ECC)
431 #endif
432 #ifndef CC_EncryptDecrypt
433 #define CC_EncryptDecrypt                  CC_YES
434 #endif
435 #ifndef CC_EncryptDecrypt2
436 #define CC_EncryptDecrypt2                 CC_YES
437 #endif
438 #ifndef CC_EventSequenceComplete
439 #define CC_EventSequenceComplete           CC_YES
440 #endif
441 #ifndef CC_EvictControl
442 #define CC_EvictControl                    CC_YES
443 #endif
444 #ifndef CC_FieldUpgradeData
445 #define CC_FieldUpgradeData                CC_NO
446 #endif
447 #ifndef CC_FieldUpgradeStart
448 #define CC_FieldUpgradeStart               CC_NO
449 #endif
450 #ifndef CC_FirmwareRead
451 #define CC_FirmwareRead                    CC_NO
452 #endif
453 #ifndef CC_FlushContext
454 #define CC_FlushContext                    CC_YES
455 #endif
456 #ifndef CC_GetCapability
457 #define CC_GetCapability                   CC_YES
458 #endif
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
459 #ifndef CC_GetCommandAuditDigest
460 #define CC_GetCommandAuditDigest             CC_YES
461 #endif
462 #ifndef CC_GetRandom
463 #define CC_GetRandom                         CC_YES
464 #endif
465 #ifndef CC_GetSessionAuditDigest
466 #define CC_GetSessionAuditDigest             CC_YES
467 #endif
468 #ifndef CC_GetTestResult
469 #define CC_GetTestResult                     CC_YES
470 #endif
471 #ifndef CC_GetTime
472 #define CC_GetTime                           CC_YES
473 #endif
474 #ifndef CC_HMAC
475 #define CC_HMAC                              (CC_YES && !ALG_CMAC)
476 #endif
477 #ifndef CC_HMAC_Start
478 #define CC_HMAC_Start                        (CC_YES && !ALG_CMAC)
479 #endif
480 #ifndef CC_Hash
481 #define CC_Hash                              CC_YES
482 #endif
483 #ifndef CC_HashSequenceStart
484 #define CC_HashSequenceStart                 CC_YES
485 #endif
486 #ifndef CC_HierarchyChangeAuth
487 #define CC_HierarchyChangeAuth               CC_YES
488 #endif
489 #ifndef CC_HierarchyControl
490 #define CC_HierarchyControl                  CC_YES
491 #endif
492 #ifndef CC_Import
493 #define CC_Import                            CC_YES
494 #endif
495 #ifndef CC_IncrementalSelfTest
496 #define CC_IncrementalSelfTest               CC_YES
497 #endif
498 #ifndef CC_Load
499 #define CC_Load                              CC_YES
500 #endif
501 #ifndef CC_LoadExternal
502 #define CC_LoadExternal                      CC_YES
503 #endif
504 #ifndef CC_MAC
505 #define CC_MAC                               (CC_YES && ALG_CMAC)
506 #endif
507 #ifndef CC_MAC_Start
508 #define CC_MAC_Start                         (CC_YES && ALG_CMAC)
509 #endif
510 #ifndef CC_MakeCredential
511 #define CC_MakeCredential                    CC_YES
512 #endif
513 #ifndef CC_NV_Certify
514 #define CC_NV_Certify                        CC_YES
515 #endif
516 #ifndef CC_NV_ChangeAuth
517 #define CC_NV_ChangeAuth                     CC_YES
518 #endif
519 #ifndef CC_NV_DefineSpace
520 #define CC_NV_DefineSpace                    CC_YES
521 #endif
522 #ifndef CC_NV_Extend
523 #define CC_NV_Extend                         CC_YES
524 #endif
```

```
525 #ifndef CC_NV_GlobalWriteLock
526 #define CC_NV_GlobalWriteLock                CC_YES
527 #endif
528 #ifndef CC_NV_Increment
529 #define CC_NV_Increment                      CC_YES
530 #endif
531 #ifndef CC_NV_Read
532 #define CC_NV_Read                           CC_YES
533 #endif
534 #ifndef CC_NV_ReadLock
535 #define CC_NV_ReadLock                       CC_YES
536 #endif
537 #ifndef CC_NV_ReadPublic
538 #define CC_NV_ReadPublic                     CC_YES
539 #endif
540 #ifndef CC_NV_SetBits
541 #define CC_NV_SetBits                        CC_YES
542 #endif
543 #ifndef CC_NV_UndefineSpace
544 #define CC_NV_UndefineSpace                  CC_YES
545 #endif
546 #ifndef CC_NV_UndefineSpaceSpecial
547 #define CC_NV_UndefineSpaceSpecial           CC_YES
548 #endif
549 #ifndef CC_NV_Write
550 #define CC_NV_Write                          CC_YES
551 #endif
552 #ifndef CC_NV_WriteLock
553 #define CC_NV_WriteLock                      CC_YES
554 #endif
555 #ifndef CC_ObjectChangeAuth
556 #define CC_ObjectChangeAuth                  CC_YES
557 #endif
558 #ifndef CC_PCR_Allocate
559 #define CC_PCR_Allocate                      CC_YES
560 #endif
561 #ifndef CC_PCR_Event
562 #define CC_PCR_Event                         CC_YES
563 #endif
564 #ifndef CC_PCR_Extend
565 #define CC_PCR_Extend                        CC_YES
566 #endif
567 #ifndef CC_PCR_Read
568 #define CC_PCR_Read                          CC_YES
569 #endif
570 #ifndef CC_PCR_Reset
571 #define CC_PCR_Reset                         CC_YES
572 #endif
573 #ifndef CC_PCR_SetAuthPolicy
574 #define CC_PCR_SetAuthPolicy                 CC_YES
575 #endif
576 #ifndef CC_PCR_SetAuthValue
577 #define CC_PCR_SetAuthValue                  CC_YES
578 #endif
579 #ifndef CC_PP_Commands
580 #define CC_PP_Commands                       CC_YES
581 #endif
582 #ifndef CC_PolicyAuthValue
583 #define CC_PolicyAuthValue                   CC_YES
584 #endif
585 #ifndef CC_PolicyAuthorize
586 #define CC_PolicyAuthorize                   CC_YES
587 #endif
588 #ifndef CC_PolicyAuthorizeNV
589 #define CC_PolicyAuthorizeNV                 CC_YES
590 #endif
```

```
591 #ifndef CC_PolicyCommandCode
592 #define CC_PolicyCommandCode                CC_YES
593 #endif
594 #ifndef CC_PolicyCounterTimer
595 #define CC_PolicyCounterTimer               CC_YES
596 #endif
597 #ifndef CC_PolicyCpHash
598 #define CC_PolicyCpHash                     CC_YES
599 #endif
600 #ifndef CC_PolicyDuplicationSelect
601 #define CC_PolicyDuplicationSelect          CC_YES
602 #endif
603 #ifndef CC_PolicyGetDigest
604 #define CC_PolicyGetDigest                  CC_YES
605 #endif
606 #ifndef CC_PolicyLocality
607 #define CC_PolicyLocality                   CC_YES
608 #endif
609 #ifndef CC_PolicyNV
610 #define CC_PolicyNV                         CC_YES
611 #endif
612 #ifndef CC_PolicyNameHash
613 #define CC_PolicyNameHash                   CC_YES
614 #endif
615 #ifndef CC_PolicyNvWritten
616 #define CC_PolicyNvWritten                  CC_YES
617 #endif
618 #ifndef CC_PolicyOR
619 #define CC_PolicyOR                         CC_YES
620 #endif
621 #ifndef CC_PolicyPCR
622 #define CC_PolicyPCR                        CC_YES
623 #endif
624 #ifndef CC_PolicyPassword
625 #define CC_PolicyPassword                   CC_YES
626 #endif
627 #ifndef CC_PolicyPhysicalPresence
628 #define CC_PolicyPhysicalPresence           CC_YES
629 #endif
630 #ifndef CC_PolicyRestart
631 #define CC_PolicyRestart                    CC_YES
632 #endif
633 #ifndef CC_PolicySecret
634 #define CC_PolicySecret                     CC_YES
635 #endif
636 #ifndef CC_PolicySigned
637 #define CC_PolicySigned                     CC_YES
638 #endif
639 #ifndef CC_PolicyTemplate
640 #define CC_PolicyTemplate                   CC_YES
641 #endif
642 #ifndef CC_PolicyTicket
643 #define CC_PolicyTicket                     CC_YES
644 #endif
645 #ifndef CC_Policy_AC_SendSelect
646 #define CC_Policy_AC_SendSelect             CC_YES
647 #endif
648 #ifndef CC_Quote
649 #define CC_Quote                            CC_YES
650 #endif
651 #ifndef CC_RSA_Decrypt
652 #define CC_RSA_Decrypt                      (CC_YES && ALG_RSA)
653 #endif
654 #ifndef CC_RSA_Encrypt
655 #define CC_RSA_Encrypt                      (CC_YES && ALG_RSA)
656 #endif
```

```
657 #ifndef CC_ReadClock
658 #define CC_ReadClock                        CC_YES
659 #endif
660 #ifndef CC_ReadPublic
661 #define CC_ReadPublic                       CC_YES
662 #endif
663 #ifndef CC_Rewrap
664 #define CC_Rewrap                           CC_YES
665 #endif
666 #ifndef CC_SelfTest
667 #define CC_SelfTest                         CC_YES
668 #endif
669 #ifndef CC_SequenceComplete
670 #define CC_SequenceComplete                 CC_YES
671 #endif
672 #ifndef CC_SequenceUpdate
673 #define CC_SequenceUpdate                   CC_YES
674 #endif
675 #ifndef CC_SetAlgorithmSet
676 #define CC_SetAlgorithmSet                  CC_YES
677 #endif
678 #ifndef CC_SetCommandCodeAuditStatus
679 #define CC_SetCommandCodeAuditStatus        CC_YES
680 #endif
681 #ifndef CC_SetPrimaryPolicy
682 #define CC_SetPrimaryPolicy                 CC_YES
683 #endif
684 #ifndef CC_Shutdown
685 #define CC_Shutdown                         CC_YES
686 #endif
687 #ifndef CC_Sign
688 #define CC_Sign                             CC_YES
689 #endif
690 #ifndef CC_StartAuthSession
691 #define CC_StartAuthSession                 CC_YES
692 #endif
693 #ifndef CC_Startup
694 #define CC_Startup                          CC_YES
695 #endif
696 #ifndef CC_StirRandom
697 #define CC_StirRandom                       CC_YES
698 #endif
699 #ifndef CC_TestParms
700 #define CC_TestParms                        CC_YES
701 #endif
702 #ifndef CC_Unseal
703 #define CC_Unseal                           CC_YES
704 #endif
705 #ifndef CC_Vendor_TCG_Test
706 #define CC_Vendor_TCG_Test                  CC_YES
707 #endif
708 #ifndef CC_VerifySignature
709 #define CC_VerifySignature                  CC_YES
710 #endif
711 #ifndef CC_ZGen_2Phase
712 #define CC_ZGen_2Phase                      (CC_YES && ALG_ECC)
713 #endif
714 #endif // _TPM_PROFILE_H_
```

A.3 TpmSizeChecks.c

A.3.1. Includes, Defines, and Types

```
1 #include    "Tpm.h"
```

 Copyright © TCG 2006-2020

```
2   #include    <stdio.h>
3   #include    <assert.h>
4   #if RUNTIME_SIZE_CHECKS
5   #if TABLE_DRIVEN_MARSHAL
6   extern uint32_t    MarshalDataSize;
7   #endif
8
9   static      int once = 0;
10
11  //** TpmSizeChecks()
12  // This function is used during the development process to make sure that the
13  // vendor-specific values result in a consistent implementation. When possible,
14  // the code contains #if to do compile-time checks. However, in some cases, the
15  // values require the use of "sizeof()" and that can't be used in an #if.
16  BOOL
17  TpmSizeChecks(
18      void
19      )
20  {
21      BOOL        PASS = TRUE;
22  #if DEBUG
23  //
24      if(once++ != 0)
25          return 1;
26      {
27          UINT32      maxAsymSecurityStrength = MAX_ASYM_SECURITY_STRENGTH;
28          UINT32      maxHashSecurityStrength = MAX_HASH_SECURITY_STRENGTH;
29          UINT32      maxSymSecurityStrength = MAX_SYM_SECURITY_STRENGTH;
30          UINT32      maxSecurityStrengthBits = MAX_SECURITY_STRENGTH_BITS;
31          UINT32      proofSize = PROOF_SIZE;
32          UINT32      compliantProofSize = COMPLIANT_PROOF_SIZE;
33          UINT32      compliantPrimarySeedSize = COMPLIANT_PRIMARY_SEED_SIZE;
34          UINT32      primarySeedSize = PRIMARY_SEED_SIZE;
35
36          UINT32      cmacState = sizeof(tpmCmacState_t);
37          UINT32      hashState = sizeof(HASH_STATE);
38          UINT32      keyScheduleSize = sizeof(tpmCryptKeySchedule_t);
39      //
40          NOT_REFERENCED(cmacState);
41          NOT_REFERENCED(hashState);
42          NOT_REFERENCED(keyScheduleSize);
43          NOT_REFERENCED(maxAsymSecurityStrength);
44          NOT_REFERENCED(maxHashSecurityStrength);
45          NOT_REFERENCED(maxSymSecurityStrength);
46          NOT_REFERENCED(maxSecurityStrengthBits);
47          NOT_REFERENCED(proofSize);
48          NOT_REFERENCED(compliantProofSize);
49          NOT_REFERENCED(compliantPrimarySeedSize);
50          NOT_REFERENCED(primarySeedSize);
51
52          {
53              TPMT_SENSITIVE           *p;
54              // This assignment keeps compiler from complaining about a conditional
55              // comparison being between two constants
56              UINT16                    max_rsa_key_bytes = MAX_RSA_KEY_BYTES;
57              if((max_rsa_key_bytes / 2) != (sizeof(p->sensitive.rsa.t.buffer) / 5))
58              {
59                  printf("Sensitive part of TPMT_SENSITIVE is undersized. May be caused"
60                         " by use of wrong version of Part 2.\n");
61                  PASS = FALSE;
62              }
63          }
64  #if TABLE_DRIVEN_MARSHAL
65          printf("sizeof(MarshalData) = %zu\n", sizeof(MarshalData_st));
66  #endif
67
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
68          printf("Size of OBJECT = %zu\n", sizeof(OBJECT));
69          printf("Size of components in TPMT_SENSITIVE = %zu\n",
    sizeof(TPMT_SENSITIVE));
70          printf("    TPMI_ALG_PUBLIC                 %zu\n", sizeof(TPMI_ALG_PUBLIC));
71          printf("    TPM2B_AUTH                      %zu\n", sizeof(TPM2B_AUTH));
72          printf("    TPM2B_DIGEST                    %zu\n", sizeof(TPM2B_DIGEST));
73          printf("    TPMU_SENSITIVE_COMPOSITE        %zu\n",
74                 sizeof(TPMU_SENSITIVE_COMPOSITE));
75      }
76          // Make sure that the size of the context blob is large enough for the largest
77          // context
78          // TPMS_CONTEXT_DATA contains two TPM2B values. That is not how this is
79          // implemented. Rather, the size field of the TPM2B_CONTEXT_DATA is used to
80          // determine the amount of data in the encrypted data. That part is not
81          // independently sized. This makes the actual size 2 bytes smaller than
82          // calculated using Part 2. Since this is opaque to the caller, it is not
83          // necessary to fix. The actual size is returned by TPM2_GetCapabilties().
84
85          // Initialize output handle.  At the end of command action, the output
86          // handle of an object will be replaced, while the output handle
87          // for a session will be the same as input
88
89          // Get the size of fingerprint in context blob.  The sequence value in
90          // TPMS_CONTEXT structure is used as the fingerprint
91      {
92          UINT32  fingerprintSize = sizeof(UINT64);
93          UINT32  integritySize = sizeof(UINT16)
94              + CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
95          UINT32  biggestObject = MAX(MAX(sizeof(HASH_OBJECT), sizeof(OBJECT)),
96                                      sizeof(SESSION));
97          UINT32  biggestContext = fingerprintSize + integritySize + biggestObject;
98
99          // round required size up to nearest 8 byte boundary.
100         biggestContext = 8 * ((biggestContext + 7) / 8);
101
102         if(MAX_CONTEXT_SIZE != biggestContext)
103         {
104             printf("MAX_CONTEXT_SIZE should be changed to %d (%d)\n",
105                    biggestContext, MAX_CONTEXT_SIZE);
106             PASS = FALSE;
107         }
108     }
109     {
110         union u
111         {
112             TPMA_OBJECT             attributes;
113             UINT32                  uint32Value;
114         } u;
115         // these are defined so that compiler doesn't complain about conditional
116         // expressions comparing two constants.
117         int                         aSize = sizeof(u.attributes);
118         int                         uSize = sizeof(u.uint32Value);
119         u.uint32Value = 0;
120         SET_ATTRIBUTE(u.attributes, TPMA_OBJECT, Reserved_bit_at_0);
121         if(u.uint32Value != 1)
122         {
123             printf("The bit allocation in a TPMA_OBJECT is not as expected");
124             PASS = FALSE;
125         }
126         if(aSize != uSize)  // comparison of two sizeof() values annoys compiler
127         {
128             printf("A TPMA_OBJECT is not the expected size.");
129             PASS = FALSE;
130         }
131     }
132     // Check that the platform implements each of the ACT that the TPM thinks
```

```
133         {
134             uint32_t            act;
135             for(act = 0; act < 16; act++)
136             {
137                 switch(act)
138                 {
139                     FOR_EACH_ACT(CASE_ACT_NUMBER)
140                         if(!_plat__ACT_GetImplemented(act))
141                         {
142                             printf("TPM_RH_ACT_%1X is not implemented by platform\n",
143                                    act);
144                             PASS = FALSE;
145                         }
146                     default:
147                         break;
148                 }
149             }
150         }
151     #endif // DEBUG
152         return (PASS);
153     }
154     #endif // RUNTIME_SIZE_CHECKS
```

Annex B
(informative)
Library-Specific

B.1 Introduction

This clause contains the files that are specific to a cryptographic library used by the TPM code.

Three categories are defined for cryptographic functions:

1) big number math (asymmetric cryptography),
2) symmetric ciphers, and
3) hash functions.

The code is structured to make it possible to use different libraries for different categories. For example, one might choose to use OpenSSL for its math library, but use a different library for hashing and symmetric cryptography. Since OpenSSL supports all three categories, it might be more typical to combine libraries of specific functions; that is, one library might only contain block ciphers while another supports big number math.

 Copyright © TCG 2006-2020

B.2 OpenSSL-Specific Files

B.2.1. Introduction

The following files are specific to a port that uses the OpenSSL library for cryptographic functions.

B.2.2. Header Files

B.2.2.1. TpmToOsslHash.h

B.2.2.1.1. Introduction

This header file is used to *splice* the OpenSSL hash code into the TPM code.

```
1   #ifndef HASH_LIB_DEFINED
2   #define HASH_LIB_DEFINED
3   #define HASH_LIB_OSSL
4   #include <openssl/evp.h>
5   #include <openssl/sha.h>
6   #include <openssl/sm3.h>
7   #include <openssl/ossl_typ.h>
```

B.2.2.1.2. Links to the OpenSSL HASH code

Redefine the internal name used for each of the hash state structures to the name used by the library. These defines need to be known in all parts of the TPM so that the structure sizes can be properly computed when needed.

```
8   #define tpmHashStateSHA1_t         SHA_CTX
9   #define tpmHashStateSHA256_t       SHA256_CTX
10  #define tpmHashStateSHA384_t       SHA512_CTX
11  #define tpmHashStateSHA512_t       SHA512_CTX
12  #define tpmHashStateSM3_256_t      SM3_CTX
```

The defines below are only needed when compiling CryptHash.c or CryptSmac.c. This isolation is primarily to avoid name space collision. However, if there is a real collision, it will likely show up when the linker tries to put things together.

```
13  #ifdef _CRYPT_HASH_C_
14  typedef BYTE          *PBYTE;
15  typedef const BYTE    *PCBYTE;
```

Define the interface between CryptHash.c to the functions provided by the library. For each method, define the calling parameters of the method and then define how the method is invoked in CryptHash.c.

All hashes are required to have the same calling sequence. If they don't, create a simple adaptation function that converts from the **standard** form of the call to the form used by the specific hash (and then send a nasty letter to the person who wrote the hash function for the library).

The macro that calls the method also defines how the parameters get swizzled between the default form (in CryptHash.c)and the library form.

```
16  #define HASH_ALIGNMENT  RADIX_BYTES
```

Initialize the hash context

```
17  #define HASH_START_METHOD_DEF   void (HASH_START_METHOD)(PANY_HASH_STATE state)
18  #define HASH_START(hashState)                                                  \
19                  ((hashState)->def->method.start)(&(hashState)->state);
```

Add data to the hash

```
20  #define HASH_DATA_METHOD_DEF                                                   \
21                  void (HASH_DATA_METHOD)(PANY_HASH_STATE state,                 \
22                                          PCBYTE buffer,                         \
23                                          size_t size)
24  #define HASH_DATA(hashState, dInSize, dIn)                                     \
25                  ((hashState)->def->method.data)(&(hashState)->state, dIn, dInSize)
```

Finalize the hash and get the digest

```
26  #define HASH_END_METHOD_DEF                                                    \
27                  void (HASH_END_METHOD)(BYTE *buffer, PANY_HASH_STATE state)
28  #define HASH_END(hashState, buffer)                                            \
29                  ((hashState)->def->method.end)(buffer, &(hashState)->state)
```

Copy the hash context

NOTE: For import, export, and copy, memcpy() is used since there is no reformatting necessary between the internal and external forms.

```
30  #define HASH_STATE_COPY_METHOD_DEF                                             \
31                  void (HASH_STATE_COPY_METHOD)(PANY_HASH_STATE to,              \
32                                                PCANY_HASH_STATE from,           \
33                                                size_t size)
34  #define HASH_STATE_COPY(hashStateOut, hashStateIn)                             \
35                  ((hashStateIn)->def->method.copy)(&(hashStateOut)->state,      \
36                                                &(hashStateIn)->state,           \
37                                                (hashStateIn)->def->contextSize)
```

Copy (with reformatting when necessary) an internal hash structure to an external blob

```
38  #define  HASH_STATE_EXPORT_METHOD_DEF                                          \
39                  void (HASH_STATE_EXPORT_METHOD)(BYTE *to,                      \
40                                              PCANY_HASH_STATE from,             \
41                                              size_t size)
42  #define  HASH_STATE_EXPORT(to, hashStateFrom)                                  \
43                  ((hashStateFrom)->def->method.copyOut)                         \
44                          (&(((BYTE *)(to))[offsetof(HASH_STATE, state)]),       \
45                           &(hashStateFrom)->state,                              \
46                           (hashStateFrom)->def->contextSize)
```

Copy from an external blob to an internal formate (with reformatting when necessary

```
47  #define  HASH_STATE_IMPORT_METHOD_DEF                                          \
48                  void (HASH_STATE_IMPORT_METHOD)(PANY_HASH_STATE to,            \
49                                                  const BYTE *from,              \
50                                                   size_t size)
51  #define  HASH_STATE_IMPORT(hashStateTo, from)                                  \
52                  ((hashStateTo)->def->method.copyIn)                            \
53                          (&(hashStateTo)->state,                                \
54                           &(((const BYTE *)(from))[offsetof(HASH_STATE, state)]),\
55                           (hashStateTo)->def->contextSize)
```

Function aliases. The code in CryptHash.c uses the internal designation for the functions. These need to be translated to the function names of the library.

```
56  #define tpmHashStart_SHA1           SHA1_Init   // external name of the
57                                                  // initialization method
```

```
58   #define tpmHashData_SHA1              SHA1_Update
59   #define tpmHashEnd_SHA1               SHA1_Final
60   #define tpmHashStateCopy_SHA1         memcpy
61   #define tpmHashStateExport_SHA1       memcpy
62   #define tpmHashStateImport_SHA1       memcpy
63   #define tpmHashStart_SHA256           SHA256_Init
64   #define tpmHashData_SHA256            SHA256_Update
65   #define tpmHashEnd_SHA256             SHA256_Final
66   #define tpmHashStateCopy_SHA256       memcpy
67   #define tpmHashStateExport_SHA256     memcpy
68   #define tpmHashStateImport_SHA256     memcpy
69   #define tpmHashStart_SHA384           SHA384_Init
70   #define tpmHashData_SHA384            SHA384_Update
71   #define tpmHashEnd_SHA384             SHA384_Final
72   #define tpmHashStateCopy_SHA384       memcpy
73   #define tpmHashStateExport_SHA384     memcpy
74   #define tpmHashStateImport_SHA384     memcpy
75   #define tpmHashStart_SHA512           SHA512_Init
76   #define tpmHashData_SHA512            SHA512_Update
77   #define tpmHashEnd_SHA512             SHA512_Final
78   #define tpmHashStateCopy_SHA512       memcpy
79   #define tpmHashStateExport_SHA512     memcpy
80   #define tpmHashStateImport_SHA_512    memcpy
81   #define tpmHashStart_SM3_256          sm3_init
82   #define tpmHashData_SM3_256           sm3_update
83   #define tpmHashEnd_SM3_256            sm3_final
84   #define tpmHashStateCopy_SM3_256      memcpy
85   #define tpmHashStateExport_SM3_256    memcpy
86   #define tpmHashStateImport_SM3_256    memcpy
87   #endif // _CRYPT_HASH_C_
88   #define LibHashInit()
```

This definition would change if there were something to report

```
89   #define HashLibSimulationEnd()
90   #endif // HASH_LIB_DEFINED
```

B.2.2.2. TpmToOsslMath.h

B.2.2.2.1. Introduction

This file contains the structure definitions used for ECC in the LibTomCrypt() version of the code. These definitions would change, based on the library. The ECC-related structures that cross the TPM interface are defined in TpmTypes.h

```
1   #ifndef MATH_LIB_DEFINED
2   #define MATH_LIB_DEFINED
3   #define MATH_LIB_OSSL
4   #include <openssl/evp.h>
5   #include <openssl/ec.h>
6   #if OPENSSL_VERSION_NUMBER >= 0x10200000L
7       // Check the bignum_st definition in crypto/bn/bn_lcl.h and either update the
8       // version check or provide the new definition for this version.
9   #   error Untested OpenSSL version
10  #elif OPENSSL_VERSION_NUMBER >= 0x10100000L
11      // from crypto/bn/bn_lcl.h
12      struct bignum_st {
13          BN_ULONG *d;                /* Pointer to an array of 'BN_BITS2' bit
14                                      * chunks. */
15          int top;                    /* Index of last used d +1. */
16                                      /* The next are internal book keeping for
bn_expand. */
17          int dmax;                   /* Size of the d array. */
18          int neg;                    /* one if the number is negative */
19          int flags;
20      };
21  #endif // OPENSSL_VERSION_NUMBER
22  #include <openssl/bn.h>
```

B.2.2.2.2. Macros and Defines

Make sure that the library is using the correct size for a crypt word

```
23  #if    defined THIRTY_TWO_BIT && (RADIX_BITS != 32)   \
24      || ((defined SIXTY_FOUR_BIT_LONG || defined SIXTY_FOUR_BIT) \
25          && (RADIX_BITS != 64))
26  #   error Ossl library is using different radix
27  #endif
```

Allocate a local BIGNUM value. For the allocation, a *bigNum* structure is created as is a local BIGNUM. The *bigNum* is initialized and then the BIGNUM is set to reference the local value.

```
28  #define BIG_VAR(name, bits)                                         \
29      BN_VAR(name##Bn, (bits));                                       \
30      BIGNUM          _##name;                                        \
31      BIGNUM          *name = BigInitialized(&_##name,                \
32                                  BnInit(name##Bn,                    \
33                                  BYTES_TO_CRYPT_WORDS(sizeof(_##name##Bn.d))))
```

Allocate a BIGNUM and initialize with the values in a *bigNum* initializer

```
34  #define BIG_INITIALIZED(name, initializer)                      \
35      BIGNUM           _##name;                                   \
36      BIGNUM          *name = BigInitialized(&_##name, initializer)
37  typedef struct
38  {
39      const ECC_CURVE_DATA    *C;     // the TPM curve values
40      EC_GROUP                *G;     // group parameters
```

```
41      BN_CTX                  *CTX;   // the context for the math (this might not be
42                                      // the context in which the curve was created>;
43  } OSSL_CURVE_DATA;
44  typedef OSSL_CURVE_DATA      *bigCurve;
45  #define AccessCurveData(E)      ((E)->C)
46  #include "TpmToOsslSupport_fp.h"
```

Start and end a context within which the OpenSSL memory management works

```
47  #define OSSL_ENTER()     BN_CTX          *CTX = OsslContextEnter()
48  #define OSSL_LEAVE()     OsslContextLeave(CTX)
```

Start and end a context that spans multiple ECC functions. This is used so that the group for the curve can persist across multiple frames.

```
49  #define CURVE_INITIALIZED(name, initializer)                        \
50      OSSL_CURVE_DATA      _##name;                                   \
51      bigCurve             name =  BnCurveInitialize(&_##name, initializer)
52  #define CURVE_FREE(name)               BnCurveFree(name)
```

Start and end a local stack frame within the context of the curve frame

```
53  #define ECC_ENTER()      BN_CTX          *CTX = OsslPushContext(E->CTX)
54  #define ECC_LEAVE()      OsslPopContext(CTX)
55  #define BN_NEW()         BnNewVariable(CTX)
```

This definition would change if there were something to report

```
56  #define MathLibSimulationEnd()
57  #endif // MATH_LIB_DEFINED
```

 Copyright © TCG 2006-2020

B.2.2.3. TpmToOsslSym.h

B.2.2.3.1. Introduction

This header file is used to *splice* the OpenSSL library into the TPM code.

The support required of a library are a hash module, a block cipher module and portions of a big number library. All of the library-dependent headers should have the same guard to that only the first one gets defined.

```
1   #ifndef SYM_LIB_DEFINED
2   #define SYM_LIB_DEFINED
3   #define SYM_LIB_OSSL
4   #include <openssl/aes.h>
5   #include <openssl/des.h>
6   #include <openssl/sm4.h>
7   #include <openssl/camellia.h>
8   #include <openssl/bn.h>
9   #include <openssl/ossl_typ.h>
```

B.2.2.3.2. Links to the OpenSSL symmetric algorithms

The Crypt functions that call the block encryption function use the parameters in the order:

a) *keySchedule*

b) in buffer

c) out buffer Since open SSL uses the order in *encryptoCall_t* above, need to swizzle the values to the order required by the library.

```
10  #define SWIZZLE(keySchedule, in, out)                                          \
11      (const BYTE *)(in), (BYTE *)(out), (void *)(keySchedule)
```

Define the order of parameters to the library functions that do block encryption and decryption.

```
12  typedef void(*TpmCryptSetSymKeyCall_t)(
13      const BYTE  *in,
14      BYTE        *out,
15      void *keySchedule
16      );
17  #define SYM_ALIGNMENT   RADIX_BYTES
```

B.2.2.3.3. Links to the OpenSSL AES code

Macros to set up the encryption/decryption key schedules

AES:

```
18  #define TpmCryptSetEncryptKeyAES(key, keySizeInBits, schedule)                 \
19      AES_set_encrypt_key((key), (keySizeInBits), (tpmKeyScheduleAES *)(schedule))
20  #define TpmCryptSetDecryptKeyAES(key, keySizeInBits, schedule)                 \
21      AES_set_decrypt_key((key), (keySizeInBits), (tpmKeyScheduleAES *)(schedule))
```

Macros to alias encryption calls to specific algorithms. This should be used sparingly. Currently, only used by CryptSym.c and CryptRand.c

When using these calls, to call the AES block encryption code, the caller should use: TpmCryptEncryptAES(SWIZZLE(*keySchedule*, in, out));

```
22    #define TpmCryptEncryptAES            AES_encrypt
23    #define TpmCryptDecryptAES            AES_decrypt
24    #define tpmKeyScheduleAES             AES_KEY
```

B.2.2.3.4. Links to the OpenSSL DES code

```
25    #if ALG_TDES
26    #include "TpmToOsslDesSupport_fp.h"
27    #endif
28    #define TpmCryptSetEncryptKeyTDES(key, keySizeInBits, schedule)                      \
29        TDES_set_encrypt_key((key), (keySizeInBits), (tpmKeyScheduleTDES *)(schedule))
30    #define TpmCryptSetDecryptKeyTDES(key, keySizeInBits, schedule)                      \
31        TDES_set_encrypt_key((key), (keySizeInBits), (tpmKeyScheduleTDES *)(schedule))
```

Macros to alias encryption calls to specific algorithms. This should be used sparingly. Currently, only used by CryptRand.c

```
32    #define TpmCryptEncryptTDES           TDES_encrypt
33    #define TpmCryptDecryptTDES           TDES_decrypt
34    #define tpmKeyScheduleTDES            DES_key_schedule
```

B.2.2.3.5. Links to the OpenSSL SM4 code

Macros to set up the encryption/decryption key schedules

```
35    #define TpmCryptSetEncryptKeySM4(key, keySizeInBits, schedule)                       \
36        SM4_set_key((key), (tpmKeyScheduleSM4 *)(schedule))
37    #define TpmCryptSetDecryptKeySM4(key, keySizeInBits, schedule)                       \
38        SM4_set_key((key), (tpmKeyScheduleSM4 *)(schedule))
```

Macros to alias encryption calls to specific algorithms. This should be used sparingly.

```
39    #define TpmCryptEncryptSM4            SM4_encrypt
40    #define TpmCryptDecryptSM4            SM4_decrypt
41    #define tpmKeyScheduleSM4             SM4_KEY
```

B.2.2.3.6. Links to the OpenSSL CAMELLIA code

Macros to set up the encryption/decryption key schedules

```
42    #define TpmCryptSetEncryptKeyCAMELLIA(key, keySizeInBits, schedule)                  \
43        Camellia_set_key((key), (keySizeInBits), (tpmKeyScheduleCAMELLIA *)(schedule))
44    #define TpmCryptSetDecryptKeyCAMELLIA(key, keySizeInBits, schedule)                  \
45        Camellia_set_key((key), (keySizeInBits), (tpmKeyScheduleCAMELLIA *)(schedule))
```

Macros to alias encryption calls to specific algorithms. This should be used sparingly.

```
46    #define TpmCryptEncryptCAMELLIA            Camellia_encrypt
47    #define TpmCryptDecryptCAMELLIA            Camellia_decrypt
48    #define tpmKeyScheduleCAMELLIA             CAMELLIA_KEY
```

Forward reference

```
49    typedef union tpmCryptKeySchedule_t tpmCryptKeySchedule_t;
```

This definition would change if there were something to report

```
50    #define SymLibSimulationEnd()
51    #endif // SYM_LIB_DEFINED
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

B.2.3. Source Files

B.2.3.1. TpmToOsslDesSupport.c

B.2.3.1.1. Introduction

The functions in this file are used for initialization of the interface to the OpenSSL library.

B.2.3.1.2. Defines and Includes

```
1   #include "Tpm.h"
2   #if (defined SYM_LIB_OSSL) && ALG_TDES
```

B.2.3.1.3. Functions

B.2.3.1.3.1. TDES_set_encyrpt_key()

This function makes creation of a TDES key look like the creation of a key for any of the other OpenSSL block ciphers. It will create three key schedules, one for each of the DES keys. If there are only two keys, then the third schedule is a copy of the first.

```
3   void
4   TDES_set_encrypt_key(
5       const BYTE                    *key,
6       UINT16                         keySizeInBits,
7       tpmKeyScheduleTDES            *keySchedule
8       )
9   {
10      DES_set_key_unchecked((const DES_cblock *)key, &keySchedule[0]);
11      DES_set_key_unchecked((const DES_cblock *)&key[8], &keySchedule[1]);
12      // If is two-key, copy the schedule for K1 into K3, otherwise, compute the
13      // the schedule for K3
14      if(keySizeInBits == 128)
15          keySchedule[2] = keySchedule[0];
16      else
17          DES_set_key_unchecked((const DES_cblock *)&key[16],
18                                &keySchedule[2]);
19  }
```

B.2.3.1.3.2. TDES_encyrpt()

The TPM code uses one key schedule. For TDES, the schedule contains three schedules. OpenSSL wants the schedules referenced separately. This function does that.

```
20  void TDES_encrypt(
21      const BYTE                *in,
22      BYTE                      *out,
23      tpmKeyScheduleTDES        *ks
24      )
25  {
26      DES_ecb3_encrypt((const DES_cblock *)in, (DES_cblock *)out,
27                       &ks[0], &ks[1], &ks[2],
28                       DES_ENCRYPT);
29  }
```

 Copyright © TCG 2006-2020

B.2.3.1.3.3. TDES_decrypt()

As with TDES_encypt() this function bridges between the TPM single schedule model and the OpenSSL three schedule model.

```
30  void TDES_decrypt(
31      const BYTE          *in,
32      BYTE                *out,
33      tpmKeyScheduleTDES   *ks
34      )
35  {
36      DES_ecb3_encrypt((const_DES_cblock *)in, (DES_cblock *)out,
37                       &ks[0], &ks[1], &ks[2],
38                       DES_DECRYPT);
39  }
40  #endif // SYM_LIB_OSSL
```

B.2.3.2. TpmToOsslMath.c

B.2.3.2.1. Introduction

The functions in this file provide the low-level interface between the TPM code and the big number and elliptic curve math routines in OpenSSL.

Most math on big numbers require a context. The context contains the memory in which OpenSSL creates and manages the big number values. When a OpenSSL math function will be called that modifies a BIGNUM value, that value must be created in an OpenSSL context. The first line of code in such a function must be: OSSL_ENTER(); and the last operation before returning must be OSSL_LEAVE(). OpenSSL variables can then be created with BnNewVariable(). Constant values to be used by OpenSSL are created from the *bigNum* values passed to the functions in this file. Space for the BIGNUM control block is allocated in the stack of the function and then it is initialized by calling BigInitialized(). That function sets up the values in the BIGNUM structure and sets the data pointer to point to the data in the bignum_t. This is only used when the value is known to be a constant in the called function.

Because the allocations of constants is on the local stack and the OSSL_ENTER()/OSSL_LEAVE() pair flushes everything created in OpenSSL memory, there should be no chance of a memory leak.

B.2.3.2.2. Includes and Defines

```
1  #include "Tpm.h"
2  #ifdef MATH_LIB_OSSL
3  #include "TpmToOsslMath_fp.h"
```

B.2.3.2.3. Functions

B.2.3.2.3.1. OsslToTpmBn()

This function converts an OpenSSL BIGNUM to a TPM bignum. In this implementation it is assumed that OpenSSL uses a different control structure but the same data layout -- an array of native-endian words in little-endian order.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure because value will not fit or OpenSSL variable doesn't exist

```
4  BOOL
5  OsslToTpmBn(
6      bigNum          bn,
7      BIGNUM          *osslBn
8      )
9  {
10     VERIFY(osslBn != NULL);
11     // If the bn is NULL, it means that an output value pointer was NULL meaning that
12     // the results is simply to be discarded.
13     if(bn != NULL)
14     {
15         int         i;
16     //
17         VERIFY((unsigned)osslBn->top <= BnGetAllocated(bn));
18         for(i = 0; i < osslBn->top; i++)
19             bn->d[i] = osslBn->d[i];
20         BnSetTop(bn, osslBn->top);
21     }
22     return TRUE;
```

 Copyright © TCG 2006-2020

```
23  Error:
24      return FALSE;
25  }
```

B.2.3.2.3.2. BigInitialized()

This function initializes an OSSL BIGNUM from a TPM *bigConst*. Do not use this for values that are passed to OpenSLL when they are not declared as const in the function prototype. Instead, use BnNewVariable().

```
26  BIGNUM *
27  BigInitialized(
28      BIGNUM              *toInit,
29      bigConst             initializer
30      )
31  {
32      if(initializer == NULL)
33          FAIL(FATAL_ERROR_PARAMETER);
34      if(toInit == NULL || initializer == NULL)
35          return NULL;
36      toInit->d = (BN_ULONG *)&initializer->d[0];
37      toInit->dmax = (int)initializer->allocated;
38      toInit->top = (int)initializer->size;
39      toInit->neg = 0;
40      toInit->flags = 0;
41      return toInit;
42  }
43  #ifndef OSSL_DEBUG
44  #   define BIGNUM_PRINT(label, bn, eol)
45  #   define DEBUG_PRINT(x)
46  #else
47  #   define DEBUG_PRINT(x)   printf("%s", x)
48  #   define BIGNUM_PRINT(label, bn, eol) BIGNUM_print((label), (bn), (eol))
```

B.2.3.2.3.3. BIGNUM_print()

```
49  static void
50  BIGNUM_print(
51      const char      *label,
52      const BIGNUM    *a,
53      BOOL             eol
54      )
55  {
56      BN_ULONG        *d;
57      int              i;
58      int              notZero = FALSE;
59
60      if(label != NULL)
61          printf("%s", label);
62      if(a == NULL)
63      {
64          printf("NULL");
65          goto done;
66      }
67      if (a->neg)
68          printf("-");
69      for(i = a->top, d = &a->d[i - 1]; i > 0; i--)
70      {
71          int         j;
72          BN_ULONG    l = *d--;
73          for(j = BN_BITS2 - 8; j >= 0; j -= 8)
74          {
75              BYTE    b = (BYTE)((l >> j) & 0xFF);
```

```
76                notZero = notZero || (b != 0);
77                if(notZero)
78                    printf("%02x", b);
79            }
80            if(!notZero)
81                printf("0");
82        }
83    done:
84        if(eol)
85            printf("\n");
86        return;
87    }
88    #endif
```

B.2.3.2.3.4. BnNewVariable()

This function allocates a new variable in the provided context. If the context does not exist or the allocation fails, it is a catastrophic failure.

```
89    static BIGNUM *
90    BnNewVariable(
91        BN_CTX          *CTX
92    )
93    {
94        BIGNUM          *new;
95    //
96        // This check is intended to protect against calling this function without
97        // having initialized the CTX.
98        if((CTX == NULL) || ((new = BN_CTX_get(CTX)) == NULL))
99            FAIL(FATAL_ERROR_ALLOCATION);
100       return new;
101   }
102   #if LIBRARY_COMPATIBILITY_CHECK
```

B.2.3.2.3.5. MathLibraryCompatibilityCheck()

```
103   BOOL
104   MathLibraryCompatibilityCheck(
105       void
106       )
107   {
108       OSSL_ENTER();
109       BIGNUM              *osslTemp = BnNewVariable(CTX);
110       crypt_uword_t        i;
111       BYTE                 test[] = {0x1F, 0x1E, 0x1D, 0x1C, 0x1B, 0x1A, 0x19, 0x18,
112                                      0x17, 0x16, 0x15, 0x14, 0x13, 0x12, 0x11, 0x10,
113                                      0x0F, 0x0E, 0x0D, 0x0C, 0x0B, 0x0A, 0x09, 0x08,
114                                      0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00};
115       BN_VAR(tpmTemp, sizeof(test) * 8); // allocate some space for a test value
116   //
117       // Convert the test data to a bigNum
118       BnFromBytes(tpmTemp, test, sizeof(test));
119       // Convert the test data to an OpenSSL BIGNUM
120       BN_bin2bn(test, sizeof(test), osslTemp);
121       // Make sure the values are consistent
122       VERIFY(osslTemp->top == (int)tpmTemp->size);
123       for(i = 0; i < tpmTemp->size; i++)
124           VERIFY(osslTemp->d[i] == tpmTemp->d[i]);
125       OSSL_LEAVE();
126       return 1;
127   Error:
128       return 0;
129   }
```


November 8, 2019 Copyright © TCG 2006-2020 Level 00 Revision 01.59

```
130 #endif
```

B.2.3.2.3.6. BnModMult()

This function does a modular multiply. It first does a multiply and then a divide and returns the remainder of the divide.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```
131 LIB_EXPORT BOOL
132 BnModMult(
133     bigNum              result,
134     bigConst            op1,
135     bigConst            op2,
136     bigConst            modulus
137     )
138 {
139     OSSL_ENTER();
140     BOOL                OK = TRUE;
141     BIGNUM              *bnResult = BN_NEW();
142     BIGNUM              *bnTemp = BN_NEW();
143     BIG_INITIALIZED(bnOp1, op1);
144     BIG_INITIALIZED(bnOp2, op2);
145     BIG_INITIALIZED(bnMod, modulus);
146 //
147     VERIFY(BN_mul(bnTemp, bnOp1, bnOp2, CTX));
148     VERIFY(BN_div(NULL, bnResult, bnTemp, bnMod, CTX));
149     VERIFY(OsslToTpmBn(result, bnResult));
150     goto Exit;
151 Error:
152     OK = FALSE;
153 Exit:
154     OSSL_LEAVE();
155     return OK;
156 }
```

B.2.3.2.3.7. BnMult()

Multiplies two numbers

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```
157 LIB_EXPORT BOOL
158 BnMult(
159     bigNum               result,
160     bigConst             multiplicand,
161     bigConst             multiplier
162     )
163 {
164     OSSL_ENTER();
165     BIGNUM              *bnTemp = BN_NEW();
166     BOOL                 OK = TRUE;
167     BIG_INITIALIZED(bnA, multiplicand);
168     BIG_INITIALIZED(bnB, multiplier);
169 //
```

```
170         VERIFY(BN_mul(bnTemp, bnA, bnB, CTX));
171         VERIFY(OsslToTpmBn(result, bnTemp));
172         goto Exit;
173     Error:
174         OK = FALSE;
175     Exit:
176         OSSL_LEAVE();
177         return OK;
178     }
```

B.2.3.2.3.8. BnDiv()

This function divides two *bigNum* values. The function returns FALSE if there is an error in the operation.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```
179     LIB_EXPORT BOOL
180     BnDiv(
181         bigNum                 quotient,
182         bigNum                 remainder,
183         bigConst               dividend,
184         bigConst               divisor
185         )
186     {
187         OSSL_ENTER();
188         BIGNUM                *bnQ = BN_NEW();
189         BIGNUM                *bnR = BN_NEW();
190         BOOL                   OK = TRUE;
191         BIG_INITIALIZED(bnDend, dividend);
192         BIG_INITIALIZED(bnSor, divisor);
193     //
194         if(BnEqualZero(divisor))
195             FAIL(FATAL_ERROR_DIVIDE_ZERO);
196         VERIFY(BN_div(bnQ, bnR, bnDend, bnSor, CTX));
197         VERIFY(OsslToTpmBn(quotient, bnQ));
198         VERIFY(OsslToTpmBn(remainder, bnR));
199         DEBUG_PRINT("In BnDiv:\n");
200         BIGNUM_PRINT("   bnDividend: ", bnDend, TRUE);
201         BIGNUM_PRINT("    bnDivisor: ", bnSor, TRUE);
202         BIGNUM_PRINT("   bnQuotient: ", bnQ, TRUE);
203         BIGNUM_PRINT("  bnRemainder: ", bnR, TRUE);
204         goto Exit;
205     Error:
206         OK = FALSE;
207     Exit:
208         OSSL_LEAVE();
209         return OK;
210     }
211     #if ALG_RSA
```

B.2.3.2.3.9. BnGcd()

Get the greatest common divisor of two numbers

 Copyright © TCG 2006-2020

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```
212 LIB_EXPORT BOOL
213 BnGcd(
214     bigNum      gcd,            // OUT: the common divisor
215     bigConst    number1,        // IN:
216     bigConst    number2         // IN:
217     )
218 {
219     OSSL_ENTER();
220     BIGNUM              *bnGcd = BN_NEW();
221     BOOL                 OK = TRUE;
222     BIG_INITIALIZED(bn1, number1);
223     BIG_INITIALIZED(bn2, number2);
224 //
225     VERIFY(BN_gcd(bnGcd, bn1, bn2, CTX));
226     VERIFY(OsslToTpmBn(gcd, bnGcd));
227     goto Exit;
228 Error:
229     OK = FALSE;
230 Exit:
231     OSSL_LEAVE();
232     return OK;
233 }
```

B.2.3.2.3.10. BnModExp()

Do modular exponentiation using *bigNum* values. The conversion from a bignum_t to a *bigNum* is trivial as they are based on the same structure

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```
234 LIB_EXPORT BOOL
235 BnModExp(
236     bigNum               result,         // OUT: the result
237     bigConst             number,         // IN: number to exponentiate
238     bigConst             exponent,       // IN:
239     bigConst             modulus         // IN:
240     )
241 {
242     OSSL_ENTER();
243     BIGNUM              *bnResult = BN_NEW();
244     BOOL                 OK = TRUE;
245     BIG_INITIALIZED(bnN, number);
246     BIG_INITIALIZED(bnE, exponent);
247     BIG_INITIALIZED(bnM, modulus);
248 //
249     VERIFY(BN_mod_exp(bnResult, bnN, bnE, bnM, CTX));
250     VERIFY(OsslToTpmBn(result, bnResult));
251     goto Exit;
252 Error:
253     OK = FALSE;
254 Exit:
255     OSSL_LEAVE();
256     return OK;
257 }
```

 Copyright © TCG 2006-2020

B.2.3.2.3.11. BnModInverse()

Modular multiplicative inverse

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```
258  LIB_EXPORT BOOL
259  BnModInverse(
260      bigNum              result,
261      bigConst            number,
262      bigConst            modulus
263      )
264  {
265      OSSL_ENTER();
266      BIGNUM             *bnResult = BN_NEW();
267      BOOL               OK = TRUE;
268      BIG_INITIALIZED(bnN, number);
269      BIG_INITIALIZED(bnM, modulus);
270  //
271      VERIFY(BN_mod_inverse(bnResult, bnN, bnM, CTX) != NULL);
272      VERIFY(OsslToTpmBn(result, bnResult));
273      goto Exit;
274  Error:
275      OK = FALSE;
276  Exit:
277      OSSL_LEAVE();
278      return OK;
279  }
280  #endif // ALG_RSA
281  #if ALG_ECC
```

B.2.3.2.3.12. PointFromOssl()

Function to copy the point result from an OSSL function to a *bigNum*

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```
282  static BOOL
283  PointFromOssl(
284      bigPoint          pOut,      // OUT: resulting point
285      EC_POINT         *pIn,       // IN: the point to return
286      bigCurve          E          // IN: the curve
287      )
288  {
289      BIGNUM          *x = NULL;
290      BIGNUM          *y = NULL;
291      BOOL             OK;
292      BN_CTX_start(E->CTX);
293  //
294      x = BN_CTX_get(E->CTX);
295      y = BN_CTX_get(E->CTX);
296
297      if(y == NULL)
298          FAIL(FATAL_ERROR_ALLOCATION);
299      // If this returns false, then the point is at infinity
300      OK = EC_POINT_get_affine_coordinates(E->G, pIn, x, y, E->CTX);
```

 Copyright © TCG 2006-2020

```
301     if(OK)
302     {
303         OsslToTpmBn(pOut->x, x);
304         OsslToTpmBn(pOut->y, y);
305         BnSetWord(pOut->z, 1);
306     }
307     else
308         BnSetWord(pOut->z, 0);
309     BN_CTX_end(E->CTX);
310     return OK;
311 }
```

B.2.3.2.3.13. EcPointInitialized()

Allocate and initialize a point.

```
312 static EC_POINT *
313 EcPointInitialized(
314     pointConst          initializer,
315     bigCurve            E
316     )
317 {
318     EC_POINT            *P = NULL;
319
320     if(initializer != NULL)
321     {
322         BIG_INITIALIZED(bnX, initializer->x);
323         BIG_INITIALIZED(bnY, initializer->y);
324         P = EC_POINT_new(E->G);
325         if(E == NULL)
326             FAIL(FATAL_ERROR_ALLOCATION);
327         if(!EC_POINT_set_affine_coordinates(E->G, P, bnX, bnY, E->CTX))
328             P = NULL;
329     }
330     return P;
331 }
```

B.2.3.2.3.14. BnCurveInitialize()

This function initializes the OpenSSL curve information structure. This structure points to the TPM-defined values for the curve, to the context for the number values in the frame, and to the OpenSSL-defined group values.

Return Value	Meaning
NULL	the TPM_ECC_CURVE is not valid or there was a problem in in initializing the curve data
non-NULL	points to *E*

```
332 LIB_EXPORT bigCurve
333 BnCurveInitialize(
334     bigCurve          E,            // IN: curve structure to initialize
335     TPM_ECC_CURVE     curveId       // IN: curve identifier
336 )
337 {
338     const ECC_CURVE_DATA    *C = GetCurveData(curveId);
339     if(C == NULL)
340         E = NULL;
341     if(E != NULL)
342     {
343         // This creates the OpenSSL memory context that stays in effect as long as the
344         // curve (E) is defined.
```

```
345         OSSL_ENTER();                          // if the allocation fails, the TPM fails
346         EC_POINT                 *P = NULL;
347         BIG_INITIALIZED(bnP, C->prime);
348         BIG_INITIALIZED(bnA, C->a);
349         BIG_INITIALIZED(bnB, C->b);
350         BIG_INITIALIZED(bnX, C->base.x);
351         BIG_INITIALIZED(bnY, C->base.y);
352         BIG_INITIALIZED(bnN, C->order);
353         BIG_INITIALIZED(bnH, C->h);
354     //
355         E->C = C;
356         E->CTX = CTX;
357
358         // initialize EC group, associate a generator point and initialize the point
359         // from the parameter data
360         // Create a group structure
361         E->G = EC_GROUP_new_curve_GFp(bnP, bnA, bnB, CTX);
362         VERIFY(E->G != NULL);
363
364         // Allocate a point in the group that will be used in setting the
365         // generator. This is not needed after the generator is set.
366         P = EC_POINT_new(E->G);
367         VERIFY(P != NULL);
368
369         // Need to use this in case Montgomery method is being used
370         VERIFY(EC_POINT_set_affine_coordinates(E->G, P, bnX, bnY, CTX));
371         // Now set the generator
372         VERIFY(EC_GROUP_set_generator(E->G, P, bnN, bnH));
373
374         EC_POINT_free(P);
375         goto Exit;
376 Error:
377         EC_POINT_free(P);
378         BnCurveFree(E);
379         E = NULL;
380     }
381 Exit:
382     return E;
383 }
```

B.2.3.2.3.15. BnCurveFree()

This function will free the allocated components of the curve and end the frame in which the curve data exists

```
384 LIB_EXPORT void
385 BnCurveFree(
386     bigCurve                     E
387 )
388 {
389     if(E)
390     {
391         EC_GROUP_free(E->G);
392         OsslContextLeave(E->CTX);
393     }
394 }
```

B.2.3.2.3.16. BnEccModMult()

This function does a point multiply of the form R = [d]S

Copyright © TCG 2006-2020

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation; treat as result being point at infinity

```
395 LIB_EXPORT BOOL
396 BnEccModMult(
397     bigPoint            R,          // OUT: computed point
398     pointConst          S,          // IN: point to multiply by 'd' (optional)
399     bigConst            d,          // IN: scalar for [d]S
400     bigCurve            E
401     )
402 {
403     EC_POINT           *pR = EC_POINT_new(E->G);
404     EC_POINT           *pS = EcPointInitialized(S, E);
405     BIG_INITIALIZED(bnD, d);
406
407     if(S == NULL)
408         EC_POINT_mul(E->G, pR, bnD, NULL, NULL, E->CTX);
409     else
410         EC_POINT_mul(E->G, pR, NULL, pS, bnD, E->CTX);
411     PointFromOssl(R, pR, E);
412     EC_POINT_free(pR);
413     EC_POINT_free(pS);
414     return !BnEqualZero(R->z);
415 }
```

B.2.3.2.3.17. BnEccModMult2()

This function does a point multiply of the form R = [d]G + [u]Q

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation; treat as result being point at infinity

```
416 LIB_EXPORT BOOL
417 BnEccModMult2(
418     bigPoint            R,          // OUT: computed point
419     pointConst          S,          // IN: optional point
420     bigConst            d,          // IN: scalar for [d]S or [d]G
421     pointConst          Q,          // IN: second point
422     bigConst            u,          // IN: second scalar
423     bigCurve            E           // IN: curve
424     )
425 {
426     EC_POINT           *pR = EC_POINT_new(E->G);
427     EC_POINT           *pS = EcPointInitialized(S, E);
428     BIG_INITIALIZED(bnD, d);
429     EC_POINT           *pQ = EcPointInitialized(Q, E);
430     BIG_INITIALIZED(bnU, u);
431
432     if(S == NULL || S == (pointConst)&(AccessCurveData(E)->base))
433         EC_POINT_mul(E->G, pR, bnD, pQ, bnU, E->CTX);
434     else
435     {
436         const EC_POINT        *points[2];
437         const BIGNUM          *scalars[2];
438         points[0] = pS;
439         points[1] = pQ;
440         scalars[0] = bnD;
441         scalars[1] = bnU;
```

```
442             EC_POINTs_mul(E->G, pR, NULL, 2, points, scalars, E->CTX);
443         }
444         PointFromOssl(R, pR, E);
445         EC_POINT_free(pR);
446         EC_POINT_free(pS);
447         EC_POINT_free(pQ);
448         return !BnEqualZero(R->z);
449     }
```

B.2.3.2.4. BnEccAdd()

This function does addition of two points.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation; treat as result being point at infinity

```
450     LIB_EXPORT BOOL
451     BnEccAdd(
452         bigPoint             R,          // OUT: computed point
453         pointConst           S,          // IN: point to multiply by 'd'
454         pointConst           Q,          // IN: second point
455         bigCurve             E           // IN: curve
456         )
457     {
458         EC_POINT            *pR = EC_POINT_new(E->G);
459         EC_POINT            *pS = EcPointInitialized(S, E);
460         EC_POINT            *pQ = EcPointInitialized(Q, E);
461     //
462         EC_POINT_add(E->G, pR, pS, pQ, E->CTX);
463
464         PointFromOssl(R, pR, E);
465         EC_POINT_free(pR);
466         EC_POINT_free(pS);
467         EC_POINT_free(pQ);
468         return !BnEqualZero(R->z);
469     }
470     #endif // ALG_ECC
471     #endif // MATHLIB OSSL
```

 Copyright © TCG 2006-2020

B.2.3.3. TpmToOsslSupport.c

B.2.3.3.1. Introduction

The functions in this file are used for initialization of the interface to the OpenSSL library.

B.2.3.3.2. Defines and Includes

```
1    #include "Tpm.h"
2    #if defined(HASH_LIB_OSSL) || defined(MATH_LIB_OSSL) || defined(SYM_LIB_OSSL)
```

Used to pass the pointers to the correct sub-keys

```
3    typedef const BYTE *desKeyPointers[3];
```

B.2.3.3.2.1. SupportLibInit()

This does any initialization required by the support library.

```
4    LIB_EXPORT int
5    SupportLibInit(
6        void
7        )
8    {
9        return TRUE;
10   }
```

B.2.3.3.2.2. OsslContextEnter()

This function is used to initialize an OpenSSL context at the start of a function that will call to an OpenSSL math function.

```
11   BN_CTX *
12   OsslContextEnter(
13       void
14       )
15   {
16       BN_CTX              *CTX = BN_CTX_new();
17   //
18       return OsslPushContext(CTX);
19   }
```

B.2.3.3.2.3. OsslContextLeave()

This is the companion function to OsslContextEnter().

```
20   void
21   OsslContextLeave(
22       BN_CTX          *CTX
23       )
24   {
25       OsslPopContext(CTX);
26       BN_CTX_free(CTX);
27   }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

B.2.3.3.2.4. OsslPushContext()

This function is used to create a frame in a context. All values allocated within this context after the frame is started will be automatically freed when the context (OsslPopContext()

```
28  BN_CTX *
29  OsslPushContext(
30      BN_CTX          *CTX
31      )
32  {
33      if(CTX == NULL)
34          FAIL(FATAL_ERROR_ALLOCATION);
35      BN_CTX_start(CTX);
36      return CTX;
37  }
```

B.2.3.3.2.5. OsslPopContext()

This is the companion function to OsslPushContext().

```
38  void
39  OsslPopContext(
40      BN_CTX          *CTX
41      )
42  {
43      // BN_CTX_end can't be called with NULL. It will blow up.
44      if(CTX != NULL)
45          BN_CTX_end(CTX);
46  }
47  #endif // HASH_LIB_OSSL || MATH_LIB_OSSL || SYM_LIB_OSSL
```

 Copyright © TCG 2006-2020

Annex C
(informative)
Simulation Environment

C.1 Introduction

These files are used to simulate some of the implementation-dependent hardware of a TPM. These files are provided to allow creation of a simulation environment for the TPM. These files are not expected to be part of a hardware TPM implementation.

C.2 Cancel.c

C.2.1. Description

This module simulates the cancel pins on the TPM.

C.2.2. Includes, Typedefs, Structures, and Defines

```
1   #include "Platform.h"
```

C.2.3. Functions

C.2.3.1. _plat__IsCanceled()

Check if the cancel flag is set

Return Value	Meaning
TRUE(1)	if cancel flag is set
FALSE(0)	if cancel flag is not set

```
2   LIB_EXPORT int
3   _plat__IsCanceled(
4       void
5       )
6   {
7       // return cancel flag
8       return s_isCanceled;
9   }
```

C.2.3.2. _plat__SetCancel()

Set cancel flag.

```
10  LIB_EXPORT void
11  _plat__SetCancel(
12      void
13      )
14  {
15      s_isCanceled = TRUE;
16      return;
17  }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

C.2.3.3. _plat__ClearCancel()

Clear cancel flag

```
18  LIB_EXPORT void
19  _plat__ClearCancel(
20      void
21      )
22  {
23      s_isCanceled = FALSE;
24      return;
25  }
```

C.3 Clock.c

C.3.1. Description

This file contains the routines that are used by the simulator to mimic a hardware clock on a TPM.

In this implementation, all the time values are measured in millisecond. However, the precision of the clock functions may be implementation dependent.

C.3.2. Includes and Data Definitions

```
1   #include <assert.h>
2   #include "Platform.h"
3   #include "TpmFail_fp.h"
```

C.3.3. Simulator Functions

C.3.3.1. Introduction

This set of functions is intended to be called by the simulator environment in order to simulate hardware events.

C.3.3.2. _plat__TimerReset()

This function sets current system clock time as t0 for counting TPM time. This function is called at a power on event to reset the clock. When the clock is reset, the indication that the clock was stopped is also set.

```
4   LIB_EXPORT void
5   _plat__TimerReset(
6       void
7       )
8   {
9       s_lastSystemTime = 0;
10      s_tpmTime = 0;
11      s_adjustRate = CLOCK_NOMINAL;
12      s_timerReset = TRUE;
13      s_timerStopped = TRUE;
14      return;
15  }
```

C.3.3.3. _plat__TimerRestart()

This function should be called in order to simulate the restart of the timer should it be stopped while power is still applied.

```
16  LIB_EXPORT void
17  _plat__TimerRestart(
18      void
19      )
20  {
21      s_timerStopped = TRUE;
22      return;
23  }
```

C.3.4. Functions Used by TPM

C.3.4.1. Introduction

These functions are called by the TPM code. They should be replaced by appropriated hardware functions.

```
24  #include <time.h>
25  clock_t     debugTime;
26
27  //*** _plat__RealTime()
28  // This is another, probably futile, attempt to define a portable function
29  // that will return a 64-bit clock value that has mSec resolution.
30  LIB_EXPORT uint64_t
31  _plat__RealTime(
32      void
33  )
34  {
35      clock64_t           time;
36  #ifdef _MSC_VER
37      struct _timeb       sysTime;
38  //
39      _ftime_s(&sysTime);
40      time = (clock64_t)(sysTime.time) * 1000 + sysTime.millitm;
41      // set the time back by one hour if daylight savings
42      if(sysTime.dstflag)
43          time -= 1000 * 60 * 60;  // mSec/sec * sec/min * min/hour = ms/hour
44  #else
45      // hopefully, this will work with most UNIX systems
46      struct timespec     systime;
47  //
48      clock_gettime(CLOCK_MONOTONIC, &systime);
49      time = (clock64_t)systime.tv_sec * 1000 + (systime.tv_nsec / 1000000);
50  #endif
51      return time;
52  }
```

C.3.4.2. _plat__TimerRead()

This function provides access to the tick timer of the platform. The TPM code uses this value to drive the TPM Clock.

The tick timer is supposed to run when power is applied to the device. This timer should not be reset by time events including _TPM_Init(). It should only be reset when TPM power is re-applied.

If the TPM is run in a protected environment, that environment may provide the tick time to the TPM as long as the time provided by the environment is not allowed to go backwards. If the time provided by the system can go backwards during a power discontinuity, then the _plat__Signal_PowerOn() should call _plat__TimerReset().

```
53  LIB_EXPORT uint64_t
54  _plat__TimerRead(
55      void
56      )
57  {
58  #ifdef HARDWARE_CLOCK
59  #error      "need a defintion for reading the hardware clock"
60      return HARDWARE_CLOCK
61  #else
62      clock64_t          timeDiff;
63      clock64_t          adjustedTimeDiff;
64      clock64_t          timeNow;
```

 Copyright © TCG 2006-2020

```
65      clock64_t          readjustedTimeDiff;
66
67      // This produces a timeNow that is basically locked to the system clock.
68      timeNow = _plat__RealTime();
69
70      // if this hasn't been initialized, initialize it
71      if(s_lastSystemTime == 0)
72      {
73          s_lastSystemTime = timeNow;
74          debugTime = clock();
75          s_lastReportedTime = 0;
76          s_realTimePrevious = 0;
77      }
78      // The system time can bounce around and that's OK as long as we don't allow
79      // time to go backwards. When the time does appear to go backwards, set
80      // lastSystemTime to be the new value and then update the reported time.
81      if(timeNow < s_lastReportedTime)
82          s_lastSystemTime = timeNow;
83      s_lastReportedTime = s_lastReportedTime + timeNow - s_lastSystemTime;
84      s_lastSystemTime = timeNow;
85      timeNow = s_lastReportedTime;
86
87      // The code above produces a timeNow that is similar to the value returned
88      // by Clock(). The difference is that timeNow does not max out, and it is
89      // at a ms. rate rather than at a CLOCKS_PER_SEC rate. The code below
90      // uses that value and does the rate adjustment on the time value.
91      // If there is no difference in time, then skip all the computations
92      if(s_realTimePrevious >= timeNow)
93          return s_tpmTime;
94      // Compute the amount of time since the last update of the system clock
95      timeDiff = timeNow - s_realTimePrevious;
96
97      // Do the time rate adjustment and conversion from CLOCKS_PER_SEC to mSec
98      adjustedTimeDiff = (timeDiff * CLOCK_NOMINAL) / ((uint64_t)s_adjustRate);
99
100     // update the TPM time with the adjusted timeDiff
101     s_tpmTime += (clock64_t)adjustedTimeDiff;
102
103     // Might have some rounding error that would loose CLOCKS. See what is not
104     // being used. As mentioned above, this could result in putting back more than
105     // is taken out. Here, we are trying to recreate timeDiff.
106     readjustedTimeDiff = (adjustedTimeDiff * (uint64_t)s_adjustRate )
107                                 / CLOCK_NOMINAL;
108
109     // adjusted is now converted back to being the amount we should advance the
110     // previous sampled time. It should always be less than or equal to timeDiff.
111     // That is, we could not have use more time than we started with.
112     s_realTimePrevious = s_realTimePrevious + readjustedTimeDiff;
113
114 #ifdef  DEBUGGING_TIME
115     // Put this in so that TPM time will pass much faster than real time when
116     // doing debug.
117     // A value of 1000 for DEBUG_TIME_MULTIPLER will make each ms into a second
118     // A good value might be 100
119     return (s_tpmTime * DEBUG_TIME_MULTIPLIER);
120 #endif
121     return s_tpmTime;
122 #endif
123 }
```

C.3.4.3. _plat__TimerWasReset()

This function is used to interrogate the flag indicating if the tick timer has been reset.

If the *resetFlag* parameter is SET, then the flag will be CLEAR before the function returns.

 Copyright © TCG 2006-2020

```
124 LIB_EXPORT int
125 _plat__TimerWasReset(
126    void
127     )
128 {
129     int           retVal = s_timerReset;
130     s_timerReset = FALSE;
131     return retVal;
132 }
```

C.3.4.4. _plat__TimerWasStopped()

This function is used to interrogate the flag indicating if the tick timer has been stopped. If so, this is typically a reason to roll the nonce.

This function will CLEAR the *s_timerStopped* flag before returning. This provides functionality that is similar to status register that is cleared when read. This is the model used here because it is the one that has the most impact on the TPM code as the flag can only be accessed by one entity in the TPM. Any other implementation of the hardware can be made to look like a read-once register.

```
133 LIB_EXPORT int
134 _plat__TimerWasStopped(
135     void
136     )
137 {
138     int           retVal = s_timerStopped;
139     s_timerStopped = FALSE;
140     return retVal;
141 }
```

C.3.4.5. _plat__ClockAdjustRate()

Adjust the clock rate

```
142 LIB_EXPORT void
143 _plat__ClockAdjustRate(
144     int                adjust         // IN: the adjust number.  It could be positive
145                                       //     or negative
146     )
147 {
148     // We expect the caller should only use a fixed set of constant values to
149     // adjust the rate
150     switch(adjust)
151     {
152         case CLOCK_ADJUST_COARSE:
153             s_adjustRate += CLOCK_ADJUST_COARSE;
154             break;
155         case -CLOCK_ADJUST_COARSE:
156             s_adjustRate -= CLOCK_ADJUST_COARSE;
157             break;
158         case CLOCK_ADJUST_MEDIUM:
159             s_adjustRate += CLOCK_ADJUST_MEDIUM;
160             break;
161         case -CLOCK_ADJUST_MEDIUM:
162             s_adjustRate -= CLOCK_ADJUST_MEDIUM;
163             break;
164         case CLOCK_ADJUST_FINE:
165             s_adjustRate += CLOCK_ADJUST_FINE;
166             break;
167         case -CLOCK_ADJUST_FINE:
168             s_adjustRate -= CLOCK_ADJUST_FINE;
169             break;
```

 Copyright © TCG 2006-2020

```
170             default:
171                 // ignore any other values;
172                 break;
173         }
174
175         if(s_adjustRate > (CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT))
176             s_adjustRate = CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT;
177         if(s_adjustRate < (CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT))
178             s_adjustRate = CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT;
179
180         return;
181     }
```

C.4 Entropy.c

C.4.1. Includes and Local Values

```
1   #define _CRT_RAND_S
2   #include <stdlib.h>
3   #include <memory.h>
4   #include <time.h>
5   #include "Platform.h"
6   #ifdef _MSC_VER
7   #include <process.h>
8   #else
9   #include <unistd.h>
10  #endif
```

This is the last 32-bits of hardware entropy produced. We have to check to see that two consecutive 32-bit values are not the same because (according to FIPS 140-2, annex C

"If each call to a RNG produces blocks of n bits (where n > 15), the first n-bit block generated after power-up, initialization, or reset shall not be used, but shall be saved for comparison with the next n-bit block to be generated. Each subsequent generation of an n-bit block shall be compared with the previously generated block. The test shall fail if any two compared n-bit blocks are equal."

```
11  extern uint32_t        lastEntropy;
12
13  //** Functions
14
15  //*** rand32()
16  // Local function to get a 32-bit random number
17  static uint32_t
18  rand32(
19      void
20  )
21  {
22      uint32_t    rndNum = rand();
23  #if RAND_MAX < UINT16_MAX
24      // If the maximum value of the random number is a 15-bit number, then shift it up
25      // 15 bits, get 15 more bits, shift that up 2 and then XOR in another value to get
26      // a full 32 bits.
27      rndNum = (rndNum << 15) ^ rand();
28      rndNum = (rndNum << 2) ^ rand();
29  #elif RAND_MAX == UINT16_MAX
30      // If the maximum size is 16-bits, shift it and add another 16 bits
31      rndNum = (rndNum << 16) ^ rand();
32  #elif RAND_MAX < UINT32_MAX
33      // If 31 bits, then shift 1 and include another random value to get the extra bit
34      rndNum = (rndNum << 1) ^ rand();
35  #endif
36      return rndNum;
37  }
```

C.4.1.1. _plat__GetEntropy()

This function is used to get available hardware entropy. In a hardware implementation of this function, there would be no call to the system to get entropy.

 Copyright © TCG 2006-2020

Return Value	Meaning
0	hardware failure of the entropy generator, this is sticky
0	the returned amount of entropy (bytes)

```
38 LIB_EXPORT int32_t
39 _plat__GetEntropy(
40     unsigned char        *entropy,           // output buffer
41     uint32_t              amount             // amount requested
42 )
43 {
44     uint32_t             rndNum;
45     int32_t              ret;
46 //
47     if(amount == 0)
48     {
49         // Seed the platform entropy source if the entropy source is software. There
50         // is no reason to put a guard macro (#if or #ifdef) around this code because
51         // this code would not be here if someone was changing it for a system with
52         // actual hardware.
53         //
54         // NOTE 1: The following command does not provide proper cryptographic
55         // entropy. Its primary purpose to make sure that different instances of the
56         // simulator, possibly started by a script on the same machine, are seeded
57         // differently. Vendors of the actual TPMs need to ensure availability of
58         // proper entropy using their platform-specific means.
59         //
60         // NOTE 2: In debug builds by default the reference implementation will seed
61         // its RNG deterministically (without using any platform provided randomness).
62         // See the USE_DEBUG_RNG macro and DRBG_GetEntropy() function.
63 #ifdef _MSC_VER
64         srand((unsigned)_plat__RealTime() ^ _getpid());
65 #else
66         srand((unsigned)_plat__RealTime() ^ getpid());
67 #endif
68         lastEntropy = rand32();
69         ret = 0;
70     }
71     else
72     {
73         rndNum = rand32();
74         if(rndNum == lastEntropy)
75         {
76             ret = -1;
77         }
78         else
79         {
80             lastEntropy = rndNum;
81             // Each process will have its random number generator initialized
82             // according to the process id and the initialization time. This is not a
83             // lot of entropy so, to add a bit more, XOR the current time value into
84             // the returned entropy value.
85             // NOTE: the reason for including the time here rather than have it in
86             // in the value assigned to lastEntropy is that rand() could be broken and
87             // using the time would in the lastEntropy value would hide this.
88             rndNum ^= (uint32_t)_plat__RealTime();
89
90             // Only provide entropy 32 bits at a time to test the ability
91             // of the caller to deal with partial results.
92             ret = MIN(amount, sizeof(rndNum));
93             memcpy(entropy, &rndNum, ret);
94         }
95     }
96     return ret;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
97  }
```

C.5 LocalityPlat.c

C.5.1. Includes

```
1   #include "Platform.h"
```

C.5.2. Functions

C.5.2.1. _plat__LocalityGet()

Get the most recent command locality in locality value form. This is an integer value for locality and not a locality structure The locality can be 0-4 or 32-255. 5-31 is not allowed.

```
2   LIB_EXPORT unsigned char
3   _plat__LocalityGet(
4       void
5       )
6   {
7       return s_locality;
8   }
```

C.5.2.2. _plat__LocalitySet()

Set the most recent command locality in locality value form

```
9   LIB_EXPORT void
10  _plat__LocalitySet(
11      unsigned char    locality
12      )
13  {
14      if(locality > 4 && locality < 32)
15          locality = 0;
16      s_locality = locality;
17      return;
18  }
```


 Copyright © TCG 2006-2020

C.6 NVMem.c

C.6.1. Description

This file contains the NV read and write access methods. This implementation uses RAM/file and does not manage the RAM/file as NV blocks. The implementation may become more sophisticated over time.

C.6.2. Includes and Local

```
1  #include <memory.h>
2  #include <string.h>
3  #include <assert.h>
4  #include "Platform.h"
5  #if FILE_BACKED_NV
6  #   include          <stdio.h>
7  FILE                  *s_NvFile = NULL;
8  int                    s_NeedsManufacture = FALSE;
9  #endif
10
11 //**Functions
12
13 #if FILE_BACKED_NV
14
15 //*** NvFileOpen()
16 // This function opens the file used to hold the NV image.
17 //  Return Type: int
18 //  >= 0        success
19 //  -1          error
20 static int
21 NvFileOpen(
22     const char       *mode
23 )
24 {
25 #if defined(NV_FILE_PATH)
26 #   define TO_STRING(s) TO_STRING_IMPL(s)
27 #   define TO_STRING_IMPL(s) #s
28     const char* s_NvFilePath = TO_STRING(NV_FILE_PATH);
29 #   undef TO_STRING
30 #   undef TO_STRING_IMPL
31 #else
32     const char* s_NvFilePath = "NVChip";
33 #endif
34
35     // Try to open an exist NVChip file for read/write
36 #   if defined _MSC_VER && 1
37     if(fopen_s(&s_NvFile, s_NvFilePath, mode) != 0)
38         s_NvFile = NULL;
39 #   else
40     s_NvFile = fopen(s_NvFilePath, mode);
41 #   endif
42     return (s_NvFile == NULL) ? -1 : 0;
43 }
```

C.6.2.1. NvFileCommit()

Write all of the contents of the NV image to a file.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
44  static int
45  NvFileCommit(
46      void
47  )
48  {
49      int          OK;
50      // If NV file is not available, return failure
51      if(s_NvFile == NULL)
52          return 1;
53      // Write RAM data to NV
54      fseek(s_NvFile, 0, SEEK_SET);
55      OK = (NV_MEMORY_SIZE == fwrite(s_NV, 1, NV_MEMORY_SIZE, s_NvFile));
56      OK = OK && (0 == fflush(s_NvFile));
57      assert(OK);
58      return OK;
59  }
```

C.6.2.2. NvFileSize()

This function gets the size of the NV file and puts the file pointer were desired using the seek method values. SEEK_SET => beginning; SEEK_CUR => current position and SEEK_END => to the end of the file.

```
60  static long
61  NvFileSize(
62      int          leaveAt
63  )
64  {
65      long     fileSize;
66      long     filePos = ftell(s_NvFile);
67  //
68      assert(NULL != s_NvFile);
69
70      fseek(s_NvFile, 0, SEEK_END);
71      fileSize = ftell(s_NvFile);
72      switch(leaveAt)
73      {
74          case SEEK_SET:
75              filePos = 0;
76          case SEEK_CUR:
77              fseek(s_NvFile, filePos, SEEK_SET);
78              break;
79          case SEEK_END:
80              break;
81          default:
82              assert(FALSE);
83              break;
84      }
85      return fileSize;
86  }
87  #endif
```

C.6.2.3. _plat__NvErrors()

This function is used by the simulator to set the error flags in the NV subsystem to simulate an error in the NV loading process

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
88  LIB_EXPORT void
89  _plat__NvErrors(
90      int                 recoverable,
91      int                 unrecoverable
92      )
93  {
94      s_NV_unrecoverable = unrecoverable;
95      s_NV_recoverable = recoverable;
96  }
```

C.6.2.4. _plat__NVEnable()

Enable NV memory.

This version just pulls in data from a file. In a real TPM, with NV on chip, this function would verify the integrity of the saved context. If the NV memory was not on chip but was in something like RPMB, the NV state would be read in, decrypted and integrity checked.

The recovery from an integrity failure depends on where the error occurred. It it was in the state that is discarded by TPM Reset, then the error is recoverable if the TPM is reset. Otherwise, the TPM must go into failure mode.

Return Value	Meaning
0	if success
0	if receive recoverable error
<0	if unrecoverable error

```
97  LIB_EXPORT int
98  _plat__NVEnable(
99      void                *platParameter  // IN: platform specific parameters
100     )
101 {
102     NOT_REFERENCED(platParameter);          // to keep compiler quiet
103 //
104     // Start assuming everything is OK
105     s_NV_unrecoverable = FALSE;
106     s_NV_recoverable = FALSE;
107 #if FILE_BACKED_NV
108     if(s_NvFile != NULL)
109         return 0;
110     // Initialize all the bytes in the ram copy of the NV
111     _plat__NvMemoryClear(0, NV_MEMORY_SIZE);
112
113     // If the file exists
114     if(NvFileOpen("r+b") >= 0)
115     {
116         long    fileSize = NvFileSize(SEEK_SET);    // get the file size and leave the
117                                                     // file pointer at the start
118 //
119         // If the size is right, read the data
120         if (NV_MEMORY_SIZE == fileSize)
121         {
122             s_NeedsManufacture =
123                 fread(s_NV, 1, NV_MEMORY_SIZE, s_NvFile) != NV_MEMORY_SIZE;
124         }
125         else
126         {
127             NvFileCommit();     // for any other size, initialize it
128             s_NeedsManufacture = TRUE;
129         }
130     }
```

```
131         // If NVChip file does not exist, try to create it for read/write.
132         else if(NvFileOpen("w+b") >= 0)
133         {
134             NvFileCommit();              // Initialize the file
135             s_NeedsManufacture = TRUE;
136         }
137         assert(NULL != s_NvFile);        // Just in case we are broken for some reason.
138     #endif
139         // NV contents have been initialized and the error checks have been performed. For
140         // simulation purposes, use the signaling interface to indicate if an error is
141         // to be simulated and the type of the error.
142         if(s_NV_unrecoverable)
143             return -1;
144         return s_NV_recoverable;
145     }
```

C.6.2.5. _plat__NVDisable()

Disable NV memory

```
146     LIB_EXPORT void
147     _plat__NVDisable(
148         int                 delete           // IN: If TRUE, delete the NV contents.
149         )
150     {
151     #if  FILE_BACKED_NV
152         if(NULL != s_NvFile)
153         {
154             fclose(s_NvFile);    // Close NV file
155             // Alternative to deleting the file is to set its size to 0. This will not
156             // match the NV size so the TPM will need to be remanufactured.
157             if(delete)
158             {
159                 // Open for writing at the start. Sets the size to zero.
160                 if(NvFileOpen("w") >= 0)
161                 {
162                     fflush(s_NvFile);
163                     fclose(s_NvFile);
164                 }
165             }
166         }
167         s_NvFile = NULL;        // Set file handle to NULL
168     #endif
169         return;
170     }
```

C.6.2.6. _plat__IsNvAvailable()

Check if NV is available

Return Value	Meaning
0	NV is available
1	NV is not available due to write failure
2	NV is not available due to rate limit

```
171     LIB_EXPORT int
172     _plat__IsNvAvailable(
173         void
174         )
175     {
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
176     int         retVal = 0;
177     // NV is not available if the TPM is in failure mode
178     if(!s_NvIsAvailable)
179         retVal = 1;
180 #if FILE_BACKED_NV
181     else
182         retVal = (s_NvFile == NULL);
183 #endif
184     return retVal;
185 }
```

C.6.2.7. _plat__NvMemoryRead()

Function: Read a chunk of NV memory

```
186 LIB_EXPORT void
187 _plat__NvMemoryRead(
188     unsigned int     startOffset,    // IN: read start
189     unsigned int     size,           // IN: size of bytes to read
190     void            *data            // OUT: data buffer
191     )
192 {
193     assert(startOffset + size <= NV_MEMORY_SIZE);
194     memcpy(data, &s_NV[startOffset], size);      // Copy data from RAM
195     return;
196 }
```

C.6.2.8. _plat__NvIsDifferent()

This function checks to see if the NV is different from the test value. This is so that NV will not be written if it has not changed.

Return Value	Meaning
TRUE(1)	the NV location is different from the test value
FALSE(0)	the NV location is the same as the test value

```
197 LIB_EXPORT int
198 _plat__NvIsDifferent(
199     unsigned int     startOffset,    // IN: read start
200     unsigned int     size,           // IN: size of bytes to read
201     void            *data            // IN: data buffer
202     )
203 {
204     return (memcmp(&s_NV[startOffset], data, size) != 0);
205 }
```

C.6.2.9. _plat__NvMemoryWrite()

This function is used to update NV memory. The **write** is to a memory copy of NV. At the end of the current command, any changes are written to the actual NV memory.

NOTE: A useful optimization would be for this code to compare the current contents of NV with the local copy and note the blocks that have changed. Then only write those blocks when _plat__NvCommit() is called.

```
206 LIB_EXPORT int
207 _plat__NvMemoryWrite(
208     unsigned int     startOffset,    // IN: write start
209     unsigned int     size,           // IN: size of bytes to write
210     void            *data            // OUT: data buffer
```

 Copyright © TCG 2006-2020

```
211         )
212     {
213         if(startOffset + size <= NV_MEMORY_SIZE)
214         {
215             memcpy(&s_NV[startOffset], data, size);     // Copy the data to the NV image
216             return TRUE;
217         }
218         return FALSE;
219     }
```

C.6.2.10. _plat__NvMemoryClear()

Function is used to set a range of NV memory bytes to an implementation-dependent value. The value represents the erase state of the memory.

```
220     LIB_EXPORT void
221     _plat__NvMemoryClear(
222         unsigned int     start,          // IN: clear start
223         unsigned int     size            // IN: number of bytes to clear
224         )
225     {
226         assert(start + size <= NV_MEMORY_SIZE);
227         // In this implementation, assume that the erase value for NV is all 1s
228         memset(&s_NV[start], 0xff, size);
229     }
```

C.6.2.11. _plat__NvMemoryMove()

Function: Move a chunk of NV memory from source to destination This function should ensure that if there overlap, the original data is copied before it is written

```
230     LIB_EXPORT void
231     _plat__NvMemoryMove(
232         unsigned int     sourceOffset,   // IN: source offset
233         unsigned int     destOffset,     // IN: destination offset
234         unsigned int     size            // IN: size of data being moved
235         )
236     {
237         assert(sourceOffset + size <= NV_MEMORY_SIZE);
238         assert(destOffset + size <= NV_MEMORY_SIZE);
239         memmove(&s_NV[destOffset], &s_NV[sourceOffset], size);      // Move data in RAM
240         return;
241     }
```

C.6.2.12. _plat__NvCommit()

This function writes the local copy of NV to NV for permanent store. It will write NV_MEMORY_SIZE bytes to NV. If a file is use, the entire file is written.

Return Value	Meaning
0	NV write success
non-0	NV write fail

```
242     LIB_EXPORT int
243     _plat__NvCommit(
244         void
245         )
246     {
247     #if FILE_BACKED_NV
```

```
248         return (NvFileCommit() ? 0 : 1);
249     #else
250         return 0;
251     #endif
252     }
```

C.6.2.13. _plat__SetNvAvail()

Set the current NV state to available. This function is for testing purpose only. It is not part of the platform NV logic

```
253     LIB_EXPORT void
254     _plat__SetNvAvail(
255         void
256         )
257     {
258         s_NvIsAvailable = TRUE;
259         return;
260     }
```

C.6.2.14. _plat__ClearNvAvail()

Set the current NV state to unavailable. This function is for testing purpose only. It is not part of the platform NV logic

```
261     LIB_EXPORT void
262     _plat__ClearNvAvail(
263         void
264         )
265     {
266         s_NvIsAvailable = FALSE;
267         return;
268     }
```

C.6.2.15. _plat__NVNeedsManufacture()

This function is used by the simulator to determine when the TPM's NV state needs to be manufactured.

```
269     LIB_EXPORT int
270     _plat__NVNeedsManufacture(
271         void
272         )
273     {
274     #if FILE_BACKED_NV
275         return s_NeedsManufacture;
276     #else
277         return FALSE;
278     #endif
279     }
```

C.7 PowerPlat.c

C.7.1. Includes and Function Prototypes

```
1  #include     "Platform.h"
2  #include     "_TPM_Init_fp.h"
```

C.7.2. Functions

C.7.2.1. _plat__Signal_PowerOn()

Signal platform power on

```
3  LIB_EXPORT int
4  _plat__Signal_PowerOn(
5      void
6      )
7  {
8      // Reset the timer
9      _plat__TimerReset();
10
11    // Need to indicate that we lost power
12     s_powerLost = TRUE;
13
14     return 0;
15 }
```

C.7.2.2. _plat__WasPowerLost()

Test whether power was lost before a _TPM_Init().

This function will clear the **hardware** indication of power loss before return. This means that there can only be one spot in the TPM code where this value gets read. This method is used here as it is the most difficult to manage in the TPM code and, if the hardware actually works this way, it is hard to make it look like anything else. So, the burden is placed on the TPM code rather than the platform code

Return Value	Meaning
TRUE(1)	power was lost
FALSE(0)	power was not lost

```
16 LIB_EXPORT int
17 _plat__WasPowerLost(
18     void
19     )
20 {
21     int          retVal = s_powerLost;
22     s_powerLost = FALSE;
23     return retVal;
24 }
```

C.7.2.3. _plat_Signal_Reset()

This a TPM reset without a power loss.

```
25 LIB_EXPORT int
26 _plat__Signal_Reset(
```

 Copyright © TCG 2006-2020

```
27      void
28      )
29  {
30      // Initialize locality
31      s_locality = 0;
32
33      // Command cancel
34      s_isCanceled = FALSE;
35
36      _TPM_Init();
37
38      // if we are doing reset but did not have a power failure, then we should
39      // not need to reload NV ...
40
41      return 0;
42  }
```

C.7.2.4. _plat__Signal_PowerOff()

Signal platform power off

```
43  LIB_EXPORT void
44  _plat__Signal_PowerOff(
45      void
46      )
47  {
48      // Prepare NV memory for power off
49      _plat__NVDisable(0);
50
51      // Disable tick ACT tick processing
52      _plat__ACT_EnableTicks(FALSE);
53
54      return;
55  }
```

 Copyright © TCG 2006-2020

C.8 PlatformData.h

This file contains the instance data for the Platform module. It is collected in this file so that the state of the module is easier to manage.

```
1   #ifndef _PLATFORM_DATA_H_
2   #define _PLATFORM_DATA_H_
3   #ifdef  _PLATFORM_DATA_C_
4   #define EXTERN
5   #else
6   #define EXTERN  extern
7   #endif
```

From Cancel.c Cancel flag. It is initialized as FALSE, which indicate the command is not being canceled

```
8   EXTERN int      s_isCanceled;
9
10  #ifndef HARDWARE_CLOCK
11  typedef uint64_t     clock64_t;
12  // This is the value returned the last time that the system clock was read. This
13  // is only relevant for a simulator or virtual TPM.
14  EXTERN clock64_t        s_realTimePrevious;
15
16  // These values are used to try to synthesize a long lived version of clock().
17  EXTERN clock64_t         s_lastSystemTime;
18  EXTERN clock64_t         s_lastReportedTime;
19
20  // This is the rate adjusted value that is the equivalent of what would be read from
21  // a hardware register that produced rate adjusted time.
22  EXTERN clock64_t         s_tpmTime;
23  #endif // HARDWARE_CLOCK
24
25  // This value indicates that the timer was reset
26  EXTERN int                s_timerReset;
27  // This value indicates that the timer was stopped. It causes a clock discontinuity.
28  EXTERN int                s_timerStopped;
29
30  // This variable records the time when _plat__TimerReset is called.  This mechanism
31  // allow us to subtract the time when TPM is power off from the total
32  // time reported by clock() function
33  EXTERN uint64_t         s_initClock;
34
35  // This variable records the timer adjustment factor.
36  EXTERN unsigned int          s_adjustRate;
37
38  // For LocalityPlat.c
39  // Locality of current command
40  EXTERN unsigned char s_locality;
41
42  // For NVMem.c
43  // Choose if the NV memory should be backed by RAM or by file.
44  // If this macro is defined, then a file is used as NV.  If it is not defined,
45  // then RAM is used to back NV memory. Comment out to use RAM.
46
47  #if (!defined VTPM) || ((VTPM != NO) && (VTPM != YES))
48  #   undef VTPM
49  #   define      VTPM            YES                // Default: Either YES or NO
50  #endif
```

For a simulation, use a file to back up the NV

```
51  #if (!defined FILE_BACKED_NV) || ((FILE_BACKED_NV != NO) && (FILE_BACKED_NV != YES))
52  #   undef    FILE_BACKED_NV
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
53  #   define  FILE_BACKED_NV          (VTPM && YES)     // Default: Either YES or NO
54  #endif
55  #if SIMULATION
56  #   undef       FILE_BACKED_NV
57  #   define      FILE_BACKED_NV          YES
58  #endif // SIMULATION
59  EXTERN unsigned char     s_NV[NV_MEMORY_SIZE];
60  EXTERN int               s_NvIsAvailable;
61  EXTERN int               s_NV_unrecoverable;
62  EXTERN int               s_NV_recoverable;
63
64  // For PPPlat.c
65  // Physical presence.  It is initialized to FALSE
66  EXTERN int              s_physicalPresence;
67
68  // From Power
69  EXTERN int              s_powerLost;
70
71  // For Entropy.c
72  EXTERN uint32_t         lastEntropy;
73
74  #define DEFINE_ACT(N)   EXTERN ACT_DATA ACT_##N;
75      FOR_EACH_ACT(DEFINE_ACT)
76  EXTERN int              actTicksAllowed;
77
78  #endif // _PLATFORM_DATA_H_
```

 Copyright © TCG 2006-2020

C.9 PlatformData.c

C.9.1. Description

This file will instance the TPM variables that are not stack allocated. The descriptions for these variables are in Global.h for this project.

C.9.2. Includes

```
1   #define _PLATFORM_DATA_C_
2   #include    "Platform.h"
```

 Copyright © TCG 2006-2020

C.10 PPPlat.c

C.10.1. Description

This module simulates the physical presence interface pins on the TPM.

C.10.2. Includes

```
1    #include "Platform.h"
```

C.10.3. Functions

C.10.3.1. _plat__PhysicalPresenceAsserted()

Check if physical presence is signaled

Return Value	Meaning
TRUE(1)	if physical presence is signaled
FALSE(0)	if physical presence is not signaled

```
2    LIB_EXPORT int
3    _plat__PhysicalPresenceAsserted(
4        void
5        )
6    {
7        // Do not know how to check physical presence without real hardware.
8        // so always return TRUE;
9        return s_physicalPresence;
10   }
```

C.10.3.2. _plat__Signal_PhysicalPresenceOn()

Signal physical presence on

```
11   LIB_EXPORT void
12   _plat__Signal_PhysicalPresenceOn(
13       void
14       )
15   {
16       s_physicalPresence = TRUE;
17       return;
18   }
```

C.10.3.3. _plat__Signal_PhysicalPresenceOff()

Signal physical presence off

```
19   LIB_EXPORT void
20   _plat__Signal_PhysicalPresenceOff(
21       void
22       )
23   {
24       s_physicalPresence = FALSE;
25       return;
26   }
```

C.11 RunCommand.c

C.11.1. Introduction

This module provides the platform specific entry and fail processing. The _plat__RunCommand() function is used to call to ExecuteCommand() in the TPM code. This function does whatever processing is necessary to set up the platform in anticipation of the call to the TPM including setup for error processing.

The _plat__Fail() function is called when there is a failure in the TPM. The TPM code will have set the flag to indicate that the TPM is in failure mode. This call will then recursively call ExecuteCommand() in order to build the failure mode response. When ExecuteCommand() returns to _plat__Fail(), the platform will do some platform specif operation to return to the environment in which the TPM is executing. For a simulator, setjmp/longjmp is used. For an OS, a system exit to the OS would be appropriate.

C.11.2. Includes and locals

```
1   #include "Platform.h"
2   #include <setjmp.h>
3   #include "ExecCommand_fp.h"
4   jmp_buf             s_jumpBuffer;
5
6   //** Functions
7
8   //***_plat__RunCommand()
9   // This version of RunCommand will set up a jum_buf and call ExecuteCommand(). If
10  // the command executes without failing, it will return and RunCommand will return.
11  // If there is a failure in the command, then _plat__Fail() is called and it will
12  // longjump back to RunCommand which will call ExecuteCommand again. However, this
13  // time, the TPM will be in failure mode so ExecuteCommand will simply build
14  // a failure response and return.
15  LIB_EXPORT void
16  _plat__RunCommand(
17      uint32_t         requestSize,    // IN: command buffer size
18      unsigned char   *request,        // IN: command buffer
19      uint32_t        *responseSize,   // IN/OUT: response buffer size
20      unsigned char   **response       // IN/OUT: response buffer
21      )
22  {
23      setjmp(s_jumpBuffer);
24      ExecuteCommand(requestSize, request, responseSize, response);
25  }
```

C.11.2.1. _plat__Fail()

This is the platform depended failure exit for the TPM.

```
26  LIB_EXPORT NORETURN void
27  _plat__Fail(
28      void
29      )
30  {
31      longjmp(&s_jumpBuffer[0], 1);
32  }
```

C.12 Unique.c

C.12.1. Introduction

In some implementations of the TPM, the hardware can provide a secret value to the TPM. This secret value is statistically unique to the instance of the TPM. Typical uses of this value are to provide personalization to the random number generation and as a shared secret between the TPM and the manufacturer.

C.12.2. Includes

```
1  #include "Platform.h"
2  const char notReallyUnique[] =
3  "This is not really a unique value. A real unique value should"
4  " be generated by the platform.";
5
6  //** _plat__GetUnique()
7  // This function is used to access the platform-specific unique value.
8  // This function places the unique value in the provided buffer ('b')
9  // and returns the number of bytes transferred. The function will not
10 // copy more data than 'bSize'.
11 // NOTE: If a platform unique value has unequal distribution of uniqueness
12 // and 'bSize' is smaller than the size of the unique value, the 'bSize'
13 // portion with the most uniqueness should be returned.
14 LIB_EXPORT uint32_t
15 _plat__GetUnique(
16     uint32_t             which,          // authorities (0) or details
17     uint32_t             bSize,          // size of the buffer
18     unsigned char       *b               // output buffer
19     )
20 {
21     const char          *from = notReallyUnique;
22     uint32_t             retVal = 0;
23
24     if(which == 0) // the authorities value
25     {
26         for(retVal = 0;
27         *from != 0 && retVal < bSize;
28             retVal++)
29         {
30             *b++ = *from++;
31         }
32     }
33     else
34     {
35 #define uSize  sizeof(notReallyUnique)
36         b = &b[((bSize < uSize) ? bSize : uSize) - 1];
37         for(retVal = 0;
38         *from != 0 && retVal < bSize;
39             retVal++)
40         {
41             *b-- = *from++;
42         }
43     }
44     return retVal;
45 }
```

C.13 DebugHelpers.c

C.13.1. Description

This file contains the NV read and write access methods. This implementation uses RAM/file and does not manage the RAM/file as NV blocks. The implementation may become more sophisticated over time.

C.13.2. Includes and Local

```
1   #include <stdio.h>
2   #include <time.h>
3   #include "Platform.h"
4   #if CERTIFYX509_DEBUG
5   FILE                *fDebug = NULL;
6   const char          *debugFileName = "DebugFile.txt";
7
8   static FILE *
9   fileOpen(
10      const char        *fn,
11      const char        *mode
12  )
13  {
14      FILE         *f;
15  #   if defined _MSC_VER
16      if(fopen_s(&f, fn, mode) != 0)
17          f = NULL;
18  #   else
19      f = fopen(fn, "w");
20  #   endif
21      return f;
22  }
```

C.13.2.1. DebugFileOpen()

This function opens the file used to hold the debug data.

Return Value	Meaning
0	success
0	error

```
23  int
24  DebugFileOpen(
25      void
26  )
27  {
28      time_t                  t = time(NULL);
29  //
30      // Get current date and time.
31  #   if defined _MSC_VER
32      char                 timeString[100];
33      ctime_s(timeString, (size_t)sizeof(timeString), &t);
34  #   else
35      char                *timeString;
36      timeString = ctime(&t);
37  #   endif
38      // Try to open the debug file
39      fDebug = fileOpen(debugFileName, "w");
40      if(fDebug)
41      {
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
42          fprintf(fDebug, "%s\n", timeString);
43          fclose(fDebug);
44          return 0;
45      }
46      return -1;
47  }
```

C.13.2.2. DebugFileClose()

```
48  void
49  DebugFileClose(
50      void
51  )
52  {
53      if(fDebug)
54          fclose(fDebug);
55  }
```

C.13.2.3. DebugDumpBuffer()

```
56  void
57  DebugDumpBuffer(
58      int                size,
59      unsigned char      *buf,
60      const char         *identifier
61  )
62  {
63      int                i;
64  //
65      FILE *f = fileOpen(debugFileName, "a");
66      if(!f)
67          return;
68      if(identifier)
69          fprintf(fDebug, "%s\n", identifier);
70      if(buf)
71      {
72          for(i = 0; i < size; i++)
73          {
74              if(((i % 16) == 0) && (i))
75                  fprintf(fDebug, "\n");
76              fprintf(fDebug, " %02X", buf[i]);
77          }
78          if((size % 16) != 0)
79              fprintf(fDebug, "\n");
80      }
81      fclose(f);
82  }
83  #endif // CERTIFYX509_DEBUG
```

 Copyright © TCG 2006-2020

C.14 Platform.h

```
1   #ifndef    _PLATFORM_H_
2   #define    _PLATFORM_H_
3   #include "TpmBuildSwitches.h"
4   #include "BaseTypes.h"
5   #include "TPMB.h"
6   #include "MinMax.h"
7   #include "TpmProfile.h"
8   #include "PlatformACT.h"
9   #include "PlatformClock.h"
10  #include "PlatformData.h"
11  #include "Platform_fp.h"
12  #endif  // _PLATFORM_H_
```

C.15 PlatformACT.h

This file contains the definitions for the ACT macros and data types used in the ACT implementation.

```
1   #ifndef _PLATFORM_ACT_H_
2   #define _PLATFORM_ACT_H_
3   typedef struct ACT_DATA
4   {
5       uint32_t            remaining;
6       uint32_t            newValue;
7       uint8_t             signaled;
8       uint8_t             pending;
9       uint8_t             number;
10  } ACT_DATA, *P_ACT_DATA;
11  #if !(defined RH_ACT_0) || (RH_ACT_0 != YES)
12  #   undef   RH_ACT_0
13  #   define  RH_ACT_0 NO
14  #   define IF_ACT_0_IMPLEMENTED(op)
15  #else
16  #   define IF_ACT_0_IMPLEMENTED(op) op(0)
17  #endif
18  #if !(defined RH_ACT_1) || (RH_ACT_1 != YES)
19  #   undef   RH_ACT_1
20  #   define  RH_ACT_1 NO
21  #   define IF_ACT_1_IMPLEMENTED(op)
22  #else
23  #   define IF_ACT_1_IMPLEMENTED(op) op(1)
24  #endif
25  #if !(defined RH_ACT_2) || (RH_ACT_2 != YES)
26  #   undef   RH_ACT_2
27  #   define  RH_ACT_2 NO
28  #   define IF_ACT_2_IMPLEMENTED(op)
29  #else
30  #   define IF_ACT_2_IMPLEMENTED(op) op(2)
31  #endif
32  #if !(defined RH_ACT_3) || (RH_ACT_3 != YES)
33  #   undef   RH_ACT_3
34  #   define  RH_ACT_3 NO
35  #   define IF_ACT_3_IMPLEMENTED(op)
36  #else
37  #   define IF_ACT_3_IMPLEMENTED(op) op(3)
38  #endif
39  #if !(defined RH_ACT_4) || (RH_ACT_4 != YES)
40  #   undef   RH_ACT_4
41  #   define  RH_ACT_4 NO
42  #   define IF_ACT_4_IMPLEMENTED(op)
43  #else
44  #   define IF_ACT_4_IMPLEMENTED(op) op(4)
45  #endif
46  #if !(defined RH_ACT_5) || (RH_ACT_5 != YES)
47  #   undef   RH_ACT_5
48  #   define  RH_ACT_5 NO
49  #   define IF_ACT_5_IMPLEMENTED(op)
50  #else
51  #   define IF_ACT_5_IMPLEMENTED(op) op(5)
52  #endif
53  #if !(defined RH_ACT_6) || (RH_ACT_6 != YES)
54  #   undef   RH_ACT_6
55  #   define  RH_ACT_6 NO
56  #   define IF_ACT_6_IMPLEMENTED(op)
57  #else
58  #   define IF_ACT_6_IMPLEMENTED(op) op(6)
59  #endif
60  #if !(defined RH_ACT_7) || (RH_ACT_7 != YES)
61  #   undef   RH_ACT_7
```

 Copyright © TCG 2006-2020

```
62 #   define  RH_ACT_7 NO
63 #   define IF_ACT_7_IMPLEMENTED(op)
64 #else
65 #   define IF_ACT_7_IMPLEMENTED(op) op(7)
66 #endif
67 #if !(defined RH_ACT_8) || (RH_ACT_8 != YES)
68 #   undef   RH_ACT_8
69 #   define  RH_ACT_8 NO
70 #   define IF_ACT_8_IMPLEMENTED(op)
71 #else
72 #   define IF_ACT_8_IMPLEMENTED(op) op(8)
73 #endif
74 #if !(defined RH_ACT_9) || (RH_ACT_9 != YES)
75 #   undef   RH_ACT_9
76 #   define  RH_ACT_9 NO
77 #   define IF_ACT_9_IMPLEMENTED(op)
78 #else
79 #   define IF_ACT_9_IMPLEMENTED(op) op(9)
80 #endif
81 #if !(defined RH_ACT_A) || (RH_ACT_A != YES)
82 #   undef   RH_ACT_A
83 #   define  RH_ACT_A NO
84 #   define IF_ACT_A_IMPLEMENTED(op)
85 #else
86 #   define IF_ACT_A_IMPLEMENTED(op) op(A)
87 #endif
88 #if !(defined RH_ACT_B) || (RH_ACT_B != YES)
89 #   undef   RH_ACT_B
90 #   define  RH_ACT_B NO
91 #   define IF_ACT_B_IMPLEMENTED(op)
92 #else
93 #   define IF_ACT_B_IMPLEMENTED(op) op(B)
94 #endif
95 #if !(defined RH_ACT_C) || (RH_ACT_C != YES)
96 #   undef   RH_ACT_C
97 #   define  RH_ACT_C NO
98 #   define IF_ACT_C_IMPLEMENTED(op)
99 #else
100 #   define IF_ACT_C_IMPLEMENTED(op) op(C)
101 #endif
102 #if !(defined RH_ACT_D) || (RH_ACT_D != YES)
103 #   undef   RH_ACT_D
104 #   define  RH_ACT_D NO
105 #   define IF_ACT_D_IMPLEMENTED(op)
106 #else
107 #   define IF_ACT_D_IMPLEMENTED(op) op(D)
108 #endif
109 #if !(defined RH_ACT_E) || (RH_ACT_E != YES)
110 #   undef   RH_ACT_E
111 #   define  RH_ACT_E NO
112 #   define IF_ACT_E_IMPLEMENTED(op)
113 #else
114 #   define IF_ACT_E_IMPLEMENTED(op) op(E)
115 #endif
116 #if !(defined RH_ACT_F) || (RH_ACT_F != YES)
117 #   undef   RH_ACT_F
118 #   define  RH_ACT_F NO
119 #   define IF_ACT_F_IMPLEMENTED(op)
120 #else
121 #   define IF_ACT_F_IMPLEMENTED(op) op(F)
122 #endif
123 #define FOR_EACH_ACT(op)                                                          \
124     IF_ACT_0_IMPLEMENTED(op)                                                      \
125     IF_ACT_1_IMPLEMENTED(op)                                                      \
126     IF_ACT_2_IMPLEMENTED(op)                                                      \
127     IF_ACT_3_IMPLEMENTED(op)                                                      \
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
128         IF_ACT_4_IMPLEMENTED(op)                                                        \
129         IF_ACT_5_IMPLEMENTED(op)                                                        \
130         IF_ACT_6_IMPLEMENTED(op)                                                        \
131         IF_ACT_7_IMPLEMENTED(op)                                                        \
132         IF_ACT_8_IMPLEMENTED(op)                                                        \
133         IF_ACT_9_IMPLEMENTED(op)                                                        \
134         IF_ACT_A_IMPLEMENTED(op)                                                        \
135         IF_ACT_B_IMPLEMENTED(op)                                                        \
136         IF_ACT_C_IMPLEMENTED(op)                                                        \
137         IF_ACT_D_IMPLEMENTED(op)                                                        \
138         IF_ACT_E_IMPLEMENTED(op)                                                        \
139         IF_ACT_F_IMPLEMENTED(op)
140     #endif // _PLATFORM_ACT_H_
```


Copyright © TCG 2006-2020

C.16 PlatformACT.c

C.16.1. Includes

```
1   #include "Platform.h"
```

C.16.2. Functions

C.16.2.1. ActSignal()

Function called when there is an ACT event to signal or unsignal

```
2   static void
3   ActSignal(
4       P_ACT_DATA          actData,
5       int                 on
6   )
7   {
8       if(actData == NULL)
9           return;
10      // If this is to turn a signal on, don't do anything if it is already on. If this
11      // is to turn the signal off, do it anyway because this might be for
12      // initialization.
13      if(on && (actData->signaled == TRUE))
14          return;
15      actData->signaled = (uint8_t)on;
16
17      // If there is an action, then replace the "Do something" with the correct action.
18      // It should test 'on' to see if it is turning the signal on or off.
19      switch(actData->number)
20      {
21  #if RH_ACT_0
22          case 0: // Do something
23              return;
24  #endif
25  #if RH_ACT_1
26          case 1: // Do something
27              return;
28  #endif
29  #if RH_ACT_2
30          case 2: // Do something
31              return;
32  #endif
33  #if RH_ACT_3
34          case 3: // Do something
35              return;
36  #endif
37  #if RH_ACT_4
38          case 4: // Do something
39              return;
40  #endif
41  #if RH_ACT_5
42          case 5: // Do something
43              return;
44  #endif
45  #if RH_ACT_6
46          case 6: // Do something
47              return;
48  #endif
49  #if RH_ACT_7
50          case 7: // Do something
51              return;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
52  #endif
53  #if RH_ACT_8
54              case 8: // Do something
55                  return;
56  #endif
57  #if RH_ACT_9
58              case 9: // Do something
59                  return;
60  #endif
61  #if RH_ACT_A
62              case 0xA: // Do something
63                  return;
64  #endif
65  #if RH_ACT_B
66              case 0xB:
67                  // Do something
68                  return;
69  #endif
70  #if RH_ACT_C
71              case 0xC: // Do something
72                  return;
73  #endif
74  #if RH_ACT_D
75              case 0xD: // Do something
76                  return;
77  #endif
78  #if RH_ACT_E
79              case 0xE: // Do something
80                  return;
81  #endif
82  #if RH_ACT_F
83              case 0xF: // Do something
84                  return;
85  #endif
86              default:
87                  return;
88      }
89  }
```

C.16.2.2. ActGetDataPointer()

```
90  static P_ACT_DATA
91  ActGetDataPointer(
92      uint32_t            act
93  )
94  {
95
96  #define RETURN_ACT_POINTER(N)  if(0x##N == act) return &ACT_##N;
97
98      FOR_EACH_ACT(RETURN_ACT_POINTER)
99
100     return (P_ACT_DATA)NULL;
101 }
```

C.16.2.3. _plat__ACT_GetImplemented()

This function tests to see if an ACT is implemented. It is a belt and suspenders function because the TPM should not be calling to to manipulate an ACT that is not implemented. However, this could help the simulator code which doesn't necessarily know if an ACT is implemented or not.

```
102 LIB_EXPORT int
103 _plat__ACT_GetImplemented(
104     uint32_t            act
```

```
105     )
106     {
107         return (ActGetDataPointer(act) != NULL);
108     }
```

C.16.2.4. _plat__ACT_GetRemaining()

This function returns the remaining time. If an update is pending, *newValue* is returned. Otherwise, the current counter value is returned. Note that since the timers keep running, the returned value can get stale immediately. The actual count value will be no greater than the returned value.

```
109     LIB_EXPORT uint32_t
110     _plat__ACT_GetRemaining(
111         uint32_t             act              //IN: the ACT selector
112     )
113     {
114         P_ACT_DATA               actData = ActGetDataPointer(act);
115         uint32_t                 remain;
116     //
117         if(actData == NULL)
118             return 0;
119         remain = actData->remaining;
120         if(actData->pending)
121             remain = actData->newValue;
122         return remain;
123     }
```

C.16.2.5. _plat__ACT_GetSignaled()

```
124     LIB_EXPORT int
125     _plat__ACT_GetSignaled(
126         uint32_t             act         //IN: number of ACT to check
127     )
128     {
129         P_ACT_DATA               actData = ActGetDataPointer(act);
130     //
131         if(actData == NULL)
132             return 0;
133         return (int )actData->signaled;
134     }
```

C.16.2.6. _plat__ACT_SetSignaled()

```
135     LIB_EXPORT void
136     _plat__ACT_SetSignaled(
137         uint32_t             act,
138         int                  on
139     )
140     {
141         ActSignal(ActGetDataPointer(act), on);
142     }
```

C.16.2.7. _plat__ACT_GetPending()

```
143     LIB_EXPORT int
144     _plat__ACT_GetPending(
145         uint32_t             act         //IN: number of ACT to check
146     )
147     {
148         P_ACT_DATA               actData = ActGetDataPointer(act);
```

 Copyright © TCG 2006-2020

```
149 //
150     if(actData == NULL)
151         return 0;
152     return (int )actData->pending;
153 }
```

C.16.2.8. _plat__ACT_UpdateCounter()

This function is used to write the *newValue* for the counter. If an update is pending, then no update occurs and the function returns FALSE. If *setSignaled* is TRUE, then the ACT signaled state is SET and if *newValue* is 0, nothing is posted.

```
154 LIB_EXPORT int
155 _plat__ACT_UpdateCounter(
156     uint32_t            act,        // IN: ACT to update
157     uint32_t            newValue   // IN: the value to post
158 )
159 {
160     P_ACT_DATA          actData = ActGetDataPointer(act);
161  //
162     if(actData == NULL)
163         // actData doesn't exist but pretend update is pending rather than indicate
164         // that a retry is necessary.
165         return TRUE;
166     // if an update is pending then return FALSE so that there will be a retry
167     if(actData->pending != 0)
168         return FALSE;
169     actData->newValue = newValue;
170     actData->pending = TRUE;
171
172     return TRUE;
173 }
```

C.16.2.9. _plat__ACT_EnableTicks()

This enables and disables the processing of the once-per-second ticks. This should be turned off (*enable* = FALSE) by _TPM_Init() and turned on (*enable* = TRUE) by TPM2_Startup() after all the initializations have completed.

```
174 LIB_EXPORT void
175 _plat__ACT_EnableTicks(
176     int             enable
177 )
178 {
179     actTicksAllowed = enable;
180 }
```

C.16.2.10. ActDecrement()

If *newValue* is non-zero it is copied to *remaining* and then *newValue* is set to zero. Then *remaining* is decremented by one if it is not already zero. If the value is decremented to zero, then the associated event is signaled. If setting *remaining* causes it to be greater than 1, then the signal associated with the ACT is turned off.

```
181 static void
182 ActDecrement(
183     P_ACT_DATA              actData
184 )
185 {
186     // Check to see if there is an update pending
```

```
187         if(actData->pending)
188         {
189             // If this update will cause the count to go from non-zero to zero, set
190             // the newValue to 1 so that it will timeout when decremented below.
191             if((actData->newValue == 0) && (actData->remaining != 0))
192                 actData->newValue = 1;
193             actData->remaining = actData->newValue;
194
195             // Update processed
196             actData->pending = 0;
197         }
198         // no update so countdown if the count is non-zero but not max
199         if((actData->remaining != 0) && (actData->remaining != UINT32_MAX))
200         {
201             // If this countdown causes the count to go to zero, then turn the signal for
202             // the ACT on.
203             if((actData->remaining -= 1) == 0)
204                 ActSignal(actData, TRUE);
205         }
206         // If the current value of the counter is non-zero, then the signal should be
207         // off.
208         if(actData->signaled && (actData->remaining > 0))
209                 ActSignal(actData, FALSE);
210     }
```

C.16.2.11._plat__ACT_Tick()

This processes the once-per-second clock tick from the hardware. This is set up for the simulator to use the control interface to send ticks to the TPM. These ticks do not have to be on a per second basis. They can be as slow or as fast as desired so that the simulation can be tested.

```
211     LIB_EXPORT void
212     _plat__ACT_Tick(
213         void
214     )
215     {
216         // Ticks processing is turned off at certain times just to make sure that nothing
217         // strange is happening before pointers and things are
218         if(actTicksAllowed)
219         {
220             // Handle the update for each counter.
221     #define DECREMENT_COUNT(N)    ActDecrement(&ACT_##N);
222
223             FOR_EACH_ACT(DECREMENT_COUNT)
224         }
225     }
```

C.16.2.12.ActZero()

This function initializes a single ACT

```
226     static void
227     ActZero(
228         uint32_t        act,
229         P_ACT_DATA      actData
230     )
231     {
232         actData->remaining = 0;
233         actData->newValue = 0;
234         actData->pending = 0;
235         actData->number = (uint8_t)act;
236         ActSignal(actData, FALSE);
```

```
237 }
```

C.16.2.13._plat__ACT_Initialize()

This function initializes the ACT hardware and data structures

```
238 LIB_EXPORT int
239 _plat__ACT_Initialize(
240     void
241 )
242 {
243     actTicksAllowed = 0;
244 #define ZERO_ACT(N)  ActZero(0x##N, &ACT_##N);
245     FOR_EACH_ACT(ZERO_ACT)
246
247     return TRUE;
248 }
```

C.17 PlatformClock.h

This file contains the instance data for the Platform module. It is collected in this file so that the state of the module is easier to manage.

```
1   #ifndef _PLATFORM_CLOCK_H_
2   #define _PLATFORM_CLOCK_H_
3   #ifdef _MSC_VER
4   #include <sys/types.h>
5   #include <sys/timeb.h>
6   #else
7   #include <sys/time.h>
8   #include <time.h>
9   #endif
```

CLOCK_NOMINAL is the number of hardware ticks per *mS*. A value of 300000 means that the nominal clock rate used to drive the hardware clock is 30 MHz. The adjustment rates are used to determine the conversion of the hardware ticks to internal hardware clock value. In practice, we would expect that there woudl be a hardware register will accumulated *mS*. It would be incremented by the output of a pre-scaler. The pre-scaler would divide the ticks from the clock by some value that would compensate for the difference between clock time and real time. The code in Clock does the emulation of this function.

```
10  #define     CLOCK_NOMINAL           30000
```

A 1% change in rate is 300 counts

```
11  #define     CLOCK_ADJUST_COARSE     300
```

A 0.1% change in rate is 30 counts

```
12  #define     CLOCK_ADJUST_MEDIUM     30
```

A minimum change in rate is 1 count

```
13  #define     CLOCK_ADJUST_FINE       1
```

The clock tolerance is +/-15% (4500 counts) Allow some guard band (16.7%)

```
14  #define     CLOCK_ADJUST_LIMIT      5000
15  #endif // _PLATFORM_CLOCK_H_
```

Annex D
(informative)
Remote Procedure Interface

D.1 Introduction

These files provide an RPC interface for a TPM simulation.

The simulation uses two ports: a command port and a hardware simulation port. Only TPM commands defined in TPM 2.0 Part 3 are sent to the TPM on the command port. The hardware simulation port is used to simulate hardware events such as power on/off and locality; and indications such as _TPM_HashStart.

 Copyright © TCG 2006-2020

D.2 TpmTcpProtocol.h

D.2.1. Introduction

TPM commands are communicated as BYTE streams on a TCP connection. The TPM command protocol is enveloped with the interface protocol described in this file. The command is indicated by a UINT32 with one of the values below. Most commands take no parameters return no TPM errors. In these cases the TPM interface protocol acknowledges that command processing is completed by returning a UINT32=0. The command TPM_SIGNAL_HASH_DATA takes a UINT32-prepended variable length BYTE array and the interface protocol acknowledges command completion with a UINT32=0. Most TPM commands are enveloped using the TPM_SEND_COMMAND interface command. The parameters are as indicated below. The interface layer also appends a UIN32=0 to the TPM response for regularity.

D.2.2. Typedefs and Defines

```
1   #ifndef      TCP_TPM_PROTOCOL_H
2   #define      TCP_TPM_PROTOCOL_H
```

D.2.3. TPM Commands

All commands acknowledge processing by returning a UINT32 == 0 except where noted

```
3   #define TPM_SIGNAL_POWER_ON         1
4   #define TPM_SIGNAL_POWER_OFF        2
5   #define TPM_SIGNAL_PHYS_PRES_ON     3
6   #define TPM_SIGNAL_PHYS_PRES_OFF    4
7   #define TPM_SIGNAL_HASH_START       5
8   #define TPM_SIGNAL_HASH_DATA        6
9           // {UINT32 BufferSize, BYTE[BufferSize] Buffer}
10  #define TPM_SIGNAL_HASH_END         7
11  #define TPM_SEND_COMMAND            8
12          // {BYTE Locality, UINT32 InBufferSize, BYTE[InBufferSize] InBuffer} ->
13          //     {UINT32 OutBufferSize, BYTE[OutBufferSize] OutBuffer}
14  #define TPM_SIGNAL_CANCEL_ON        9
15  #define TPM_SIGNAL_CANCEL_OFF       10
16  #define TPM_SIGNAL_NV_ON            11
17  #define TPM_SIGNAL_NV_OFF           12
18  #define TPM_SIGNAL_KEY_CACHE_ON     13
19  #define TPM_SIGNAL_KEY_CACHE_OFF    14
20  #define TPM_REMOTE_HANDSHAKE        15
21  #define TPM_SET_ALTERNATIVE_RESULT  16
22  #define TPM_SIGNAL_RESET            17
23  #define TPM_SIGNAL_RESTART          18
24  #define TPM_SESSION_END             20
25  #define TPM_STOP                    21
26  #define TPM_GET_COMMAND_RESPONSE_SIZES  25
27  #define TPM_ACT_GET_SIGNALED        26
28  #define TPM_TEST_FAILURE_MODE       30
```

D.2.4. Enumerations and Structures

```
29  enum TpmEndPointInfo
30  {
31      tpmPlatformAvailable = 0x01,
32      tpmUsesTbs = 0x02,
33      tpmInRawMode = 0x04,
34      tpmSupportsPP = 0x08
35  };
36  #ifdef _MSC_VER
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
37  #   pragma warning(push, 3)
38  #endif
```

Existing RPC interface type definitions retained so that the implementation can be re-used

```
39  typedef struct in_buffer
40  {
41      unsigned long BufferSize;
42      unsigned char *Buffer;
43  } _IN_BUFFER;
44  typedef unsigned char *_OUTPUT_BUFFER;
45  typedef struct out_buffer
46  {
47      uint32_t         BufferSize;
48      _OUTPUT_BUFFER   Buffer;
49  } _OUT_BUFFER;
50  #ifdef _MSC_VER
51  #   pragma warning(pop)
52  #endif
53  #ifndef WIN32
54  typedef unsigned long        DWORD;
55  typedef void                *LPVOID;
56  #endif
57  #endif
```

 Copyright © TCG 2006-2020

D.3 TcpServer.c

D.3.1. Description

This file contains the socket interface to a TPM simulator.

D.3.2. Includes, Locals, Defines and Function Prototypes

```
1   #include "TpmBuildSwitches.h"
2   #include <stdio.h>
3   #ifdef _MSC_VER
4   #   pragma warning(push, 3)
5   #   include <windows.h>
6   #   include <winsock.h>
7   #   pragma warning(pop)
8   typedef int socklen_t;
9   #elif defined(__unix__)
10  #   include <string.h>
11  #   include <unistd.h>
12  #   include <errno.h>
13  #   include <stdint.h>
14  #   include <netinet/in.h>
15  #   include <sys/socket.h>
16  #   include <pthread.h>
17  #   define ZeroMemory(ptr, sz) (memset((ptr), 0, (sz)))
18  #   define closesocket(x) close(x)
19  #   define INVALID_SOCKET (-1)
20  #   define SOCKET_ERROR   (-1)
21  #   define WSAGetLastError() (errno)
22  #   define INT_PTR intptr_t
23      typedef int SOCKET;
24  #else
25  #   error "Unsupported platform."
26  #endif
27  #ifndef TRUE
28  #   define TRUE    1
29  #endif
30  #ifndef FALSE
31  #   define FALSE   0
32  #endif
33  #include <string.h>
34  #include <stdlib.h>
35  #include <stdint.h>
36  #include "TpmTcpProtocol.h"
37  #include "Manufacture_fp.h"
38  #include "TpmProfile.h"
39  #include "Simulator_fp.h"
40  #include "Platform_fp.h"
41  typedef int         BOOL;
```

To access key cache control in TPM

```
42  void RsaKeyCacheControl(int state);
43  #ifndef __IGNORE_STATE__
44  static uint32_t ServerVersion = 1;
45
46  #define MAX_BUFFER 1048576
47  char InputBuffer[MAX_BUFFER];       //The input data buffer for the simulator.
48  char OutputBuffer[MAX_BUFFER];      //The output data buffer for the simulator.
49
50  struct
51  {
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
52      uint32_t    largestCommandSize;
53      uint32_t    largestCommand;
54      uint32_t    largestResponseSize;
55      uint32_t    largestResponse;
56  } CommandResponseSizes = {0};
57
58  #endif // __IGNORE_STATE___
59
60  //** Functions
61
62  //*** CreateSocket()
63  // This function creates a socket listening on 'PortNumber'.
64  static int
65  CreateSocket(
66      int                     PortNumber,
67      SOCKET                 *listenSocket
68      )
69  {
70      struct                  sockaddr_in MyAddress;
71      int res;
72  //
73      // Initialize Winsock
74  #ifdef _MSC_VER
75      WSADATA                 wsaData;
76      res = WSAStartup(MAKEWORD(2, 2), &wsaData);
77      if(res != 0)
78      {
79          printf("WSAStartup failed with error: %d\n", res);
80          return -1;
81      }
82  #endif
83      // create listening socket
84      *listenSocket = socket(PF_INET, SOCK_STREAM, 0);
85      if(INVALID_SOCKET == *listenSocket)
86      {
87          printf("Cannot create server listen socket.  Error is 0x%x\n",
88                 WSAGetLastError());
89          return -1;
90      }
91      // bind the listening socket to the specified port
92      ZeroMemory(&MyAddress, sizeof(MyAddress));
93      MyAddress.sin_port = htons((short)PortNumber);
94      MyAddress.sin_family = AF_INET;
95
96      res = bind(*listenSocket, (struct sockaddr*) &MyAddress, sizeof(MyAddress));
97      if(res == SOCKET_ERROR)
98      {
99          printf("Bind error.  Error is 0x%x\n", WSAGetLastError());
100         return -1;
101     }
102     // listen/wait for server connections
103     res = listen(*listenSocket, 3);
104     if(res == SOCKET_ERROR)
105     {
106         printf("Listen error.  Error is 0x%x\n", WSAGetLastError());
107         return -1;
108     }
109     return 0;
110 }
```

D.3.2.1. PlatformServer()

This function processes incoming platform requests.

 Copyright © TCG 2006-2020

```
111 BOOL
112 PlatformServer(
113     SOCKET           s
114     )
115 {
116     BOOL                    OK = TRUE;
117     uint32_t                Command;
118 //
119     for(;;)
120     {
121         OK = ReadBytes(s, (char*)&Command, 4);
122         // client disconnected (or other error).  We stop processing this client
123         // and return to our caller who can stop the server or listen for another
124         // connection.
125         if(!OK)
126             return TRUE;
127         Command = ntohl(Command);
128         switch(Command)
129         {
130             case TPM_SIGNAL_POWER_ON:
131                 _rpc__Signal_PowerOn(FALSE);
132                 break;
133             case TPM_SIGNAL_POWER_OFF:
134                 _rpc__Signal_PowerOff();
135                 break;
136             case TPM_SIGNAL_RESET:
137                 _rpc__Signal_PowerOn(TRUE);
138                 break;
139             case TPM_SIGNAL_RESTART:
140                 _rpc__Signal_Restart();
141                 break;
142             case TPM_SIGNAL_PHYS_PRES_ON:
143                 _rpc__Signal_PhysicalPresenceOn();
144                 break;
145             case TPM_SIGNAL_PHYS_PRES_OFF:
146                 _rpc__Signal_PhysicalPresenceOff();
147                 break;
148             case TPM_SIGNAL_CANCEL_ON:
149                 _rpc__Signal_CancelOn();
150                 break;
151             case TPM_SIGNAL_CANCEL_OFF:
152                 _rpc__Signal_CancelOff();
153                 break;
154             case TPM_SIGNAL_NV_ON:
155                 _rpc__Signal_NvOn();
156                 break;
157             case TPM_SIGNAL_NV_OFF:
158                 _rpc__Signal_NvOff();
159                 break;
160             case TPM_SIGNAL_KEY_CACHE_ON:
161                 _rpc__RsaKeyCacheControl(TRUE);
162                 break;
163             case TPM_SIGNAL_KEY_CACHE_OFF:
164                 _rpc__RsaKeyCacheControl(FALSE);
165                 break;
166             case TPM_SESSION_END:
167                 // Client signaled end-of-session
168                 TpmEndSimulation();
169                 return TRUE;
170             case TPM_STOP:
171                 // Client requested the simulator to exit
172                 return FALSE;
173             case TPM_TEST_FAILURE_MODE:
174                 _rpc__ForceFailureMode();
175                 break;
176             case TPM_GET_COMMAND_RESPONSE_SIZES:
```

 Copyright © TCG 2006-2020

```
177                 OK = WriteVarBytes(s, (char *)&CommandResponseSizes,
178                                    sizeof(CommandResponseSizes));
179                 memset(&CommandResponseSizes, 0, sizeof(CommandResponseSizes));
180                 if(!OK)
181                     return TRUE;
182                 break;
183             case TPM_ACT_GET_SIGNALED:
184             {
185                 UINT32 actHandle;
186                 OK = ReadUINT32(s, &actHandle);
187                 WriteUINT32(s, _rpc__ACT_GetSignaled(actHandle));
188                 break;
189             }
190             default:
191                 printf("Unrecognized platform interface command %d\n",
192                        (int)Command);
193                 WriteUINT32(s, 1);
194                 return TRUE;
195         }
196         WriteUINT32(s, 0);
197     }
198 }
```

D.3.2.2. PlatformSvcRoutine()

This function is called to set up the socket interfaces to listen for commands.

```
199 DWORD WINAPI
200 PlatformSvcRoutine(
201     LPVOID           port
202     )
203 {
204     int                     PortNumber = (int)(INT_PTR)port;
205     SOCKET                  listenSocket, serverSocket;
206     struct                  sockaddr_in HerAddress;
207     int                     res;
208     socklen_t               length;
209     BOOL                    continueServing;
210 //
211     res = CreateSocket(PortNumber, &listenSocket);
212     if(res != 0)
213     {
214         printf("Create platform service socket fail\n");
215         return res;
216     }
217     // Loop accepting connections one-by-one until we are killed or asked to stop
218     // Note the platform service is single-threaded so we don't listen for a new
219     // connection until the prior connection drops.
220     do
221     {
222         printf("Platform server listening on port %d\n", PortNumber);
223
224         // blocking accept
225         length = sizeof(HerAddress);
226         serverSocket = accept(listenSocket,
227                               (struct sockaddr*) &HerAddress,
228                               &length);
229         if(serverSocket == INVALID_SOCKET)
230         {
231             printf("Accept error.  Error is 0x%x\n", WSAGetLastError());
232             return (DWORD)-1;
233         }
234         printf("Client accepted\n");
235
```

```
236             // normal behavior on client disconnection is to wait for a new client
237             // to connect
238             continueServing = PlatformServer(serverSocket);
239             closesocket(serverSocket);
240         } while(continueServing);
241
242         return 0;
243     }
```

D.3.2.3. PlatformSignalService()

This function starts a new thread waiting for platform signals. Platform signals are processed one at a time in the order in which they are received.

```
244     int
245     PlatformSignalService(
246         int                 PortNumber
247         )
248     {
249     #if defined(_MSC_VER)
250         HANDLE                  hPlatformSvc;
251         int                     ThreadId;
252         int                     port = PortNumber;
253     //
254         // Create service thread for platform signals
255         hPlatformSvc = CreateThread(NULL, 0,
256                                     (LPTHREAD_START_ROUTINE)PlatformSvcRoutine,
257                                     (LPVOID)(INT_PTR)port, 0, (LPDWORD)&ThreadId);
258         if(hPlatformSvc == NULL)
259         {
260             printf("Thread Creation failed\n");
261             return -1;
262         }
263         return 0;
264     #else
265         pthread_t               thread_id;
266         int                     ret;
267         int                     port = PortNumber;
268
269         ret = pthread_create(&thread_id, NULL, (void*)PlatformSvcRoutine,
270                              (LPVOID)(INT_PTR)port);
271         if (ret == -1)
272         {
273             printf("pthread_create failed: %s", strerror(ret));
274         }
275         return ret;
276     #endif // _MSC_VER
277     }
```

D.3.2.4. RegularCommandService()

This function services regular commands.

```
278     int
279     RegularCommandService(
280         int                 PortNumber
281         )
282     {
283         SOCKET                  listenSocket;
284         SOCKET                  serverSocket;
285         struct                  sockaddr_in HerAddress;
286         int                     res;
287         socklen_t               length;
```


 Copyright © TCG 2006-2020

```
288     BOOL                    continueServing;
289 //
290     res = CreateSocket(PortNumber, &listenSocket);
291     if(res != 0)
292     {
293         printf("Create platform service socket fail\n");
294         return res;
295     }
296     // Loop accepting connections one-by-one until we are killed or asked to stop
297     // Note the TPM command service is single-threaded so we don't listen for
298     // a new connection until the prior connection drops.
299     do
300     {
301         printf("TPM command server listening on port %d\n", PortNumber);
302
303         // blocking accept
304         length = sizeof(HerAddress);
305         serverSocket = accept(listenSocket,
306                               (struct sockaddr*) &HerAddress,
307                               &length);
308         if(serverSocket == INVALID_SOCKET)
309         {
310             printf("Accept error.  Error is 0x%x\n", WSAGetLastError());
311             return -1;
312         }
313         printf("Client accepted\n");
314
315         // normal behavior on client disconnection is to wait for a new client
316         // to connect
317         continueServing = TpmServer(serverSocket);
318         closesocket(serverSocket);
319     } while(continueServing);
320     return 0;
321 }
322 #if RH_ACT_0
```

D.3.2.5. SimulatorTimeServiceRoutine()

This function is called to service the time *ticks*.

```
323 static DWORD WINAPI
324 SimulatorTimeServiceRoutine(
325     LPVOID          notUsed
326 )
327 {
328     // All time is in ms
329     const INT64   tick = 1000;
330     UINT64  prevTime = _plat__RealTime();
331     INT64   timeout = tick;
332
333     (void)notUsed;
334
335     while (TRUE)
336     {
337         UINT64  curTime;
338
339 #if defined(_MSC_VER)
340         Sleep((DWORD)timeout);
341 #else
342         struct timespec req = {timeout / 1000, (timeout % 1000) * 1000}
343                         rem;
344         nanosleep(&req, &rem);
345 #endif // _MSC_VER
346         curTime = _plat__RealTime();
```

```
347
348             // May need to issue several ticks if the Sleep() took longer than asked,
349             // or no ticks at all, it Sleep() was interrupted prematurely.
350             while (prevTime < curTime - tick / 2)
351             {
352                 //printf("%05lld | %05lld\n",
353                 //       prevTime % 100000, (curTime - tick / 2) % 100000);
354                 _plat__ACT_Tick();
355                 prevTime += (UINT64)tick;
356             }
357             // Adjust the next timeout to keep the average interval of one second
358             timeout = tick + (prevTime - curTime);
359             //prevTime = curTime;
360             //printf("%04lld | c:%05lld | p:%05llu\n",
361             //          timeout, curTime % 100000, prevTime);
362         }
363         return 0;
364     }
```

D.3.2.6. ActTimeService()

This function starts a new thread waiting to wait for time ticks.

Return Value	Meaning
==0	success
!=0	failure

```
365     static int
366     ActTimeService(
367         void
368     )
369     {
370         static BOOL          running = FALSE;
371         int                  ret = 0;
372         if(!running)
373         {
374     #if defined(_MSC_VER)
375             HANDLE               hThr;
376             int                  ThreadId;
377         //
378             printf("Starting ACT thread...\n");
379             //  Don't allow ticks to be processed before TPM is manufactured.
380             _plat__ACT_EnableTicks(FALSE);
381
382             // Create service thread for ACT internal timer
383             hThr = CreateThread(NULL, 0,
384                 (LPTHREAD_START_ROUTINE)SimulatorTimeServiceRoutine,
385                                 (LPVOID)(INT_PTR)NULL, 0, (LPDWORD)&ThreadId);
386             if(hThr != NULL)
387                 CloseHandle(hThr);
388             else
389                 ret = -1;
390     #else
391             pthread_t            thread_id;
392     //
393             ret = pthread_create(&thread_id, NULL, (void*)PlatformSvcRoutine,
394                 (LPVOID)(INT_PTR)NULL);
395     #endif // _MSC_VER
396
397             if(ret != 0)
398                 printf("ACT thread Creation failed\n");
399             else
```

```
400                 running = TRUE;
401         }
402         return ret;
403     }
404     #endif // RH_ACT_0
```

D.3.2.7. StartTcpServer()

This is the main entry-point to the TCP server. The server listens on port specified.

Note that there is no way to specify the network interface in this implementation.

```
405     int
406     StartTcpServer(
407         int                 PortNumber
408         )
409     {
410         int                     res;
411     //
412     #if RH_ACT_0 || 1
413         // Start the Time Service routine
414         res = ActTimeService();
415         if(res != 0)
416         {
417             printf("TimeService failed\n");
418             return res;
419         }
420     #endif
421
422         // Start Platform Signal Processing Service
423         res = PlatformSignalService(PortNumber + 1);
424         if(res != 0)
425         {
426             printf("PlatformSignalService failed\n");
427             return res;
428         }
429         // Start Regular/DRTM TPM command service
430         res = RegularCommandService(PortNumber);
431         if(res != 0)
432         {
433             printf("RegularCommandService failed\n");
434             return res;
435         }
436         return 0;
437     }
```

D.3.2.8. ReadBytes()

This function reads the indicated number of bytes (*NumBytes*) into buffer from the indicated socket.

```
438     BOOL
439     ReadBytes(
440         SOCKET           s,
441         char            *buffer,
442         int              NumBytes
443         )
444     {
445         int                     res;
446         int                     numGot = 0;
447     //
448         while(numGot < NumBytes)
449         {
450             res = recv(s, buffer + numGot, NumBytes - numGot, 0);
```

```
451         if(res == -1)
452         {
453             printf("Receive error.  Error is 0x%x\n", WSAGetLastError());
454             return FALSE;
455         }
456         if(res == 0)
457         {
458             return FALSE;
459         }
460         numGot += res;
461     }
462     return TRUE;
463 }
```

D.3.2.9. WriteBytes()

This function will send the indicated number of bytes (*NumBytes*) to the indicated socket

```
464 BOOL
465 WriteBytes(
466     SOCKET           s,
467     char            *buffer,
468     int              NumBytes
469     )
470 {
471     int                 res;
472     int                 numSent = 0;
473 //
474     while(numSent < NumBytes)
475     {
476         res = send(s, buffer + numSent, NumBytes - numSent, 0);
477         if(res == -1)
478         {
479             if(WSAGetLastError() == 0x2745)
480             {
481                 printf("Client disconnected\n");
482             }
483             else
484             {
485                 printf("Send error.  Error is 0x%x\n", WSAGetLastError());
486             }
487             return FALSE;
488         }
489         numSent += res;
490     }
491     return TRUE;
492 }
```

D.3.2.10. WriteUINT32()

Send 4 byte integer

```
493 BOOL
494 WriteUINT32(
495     SOCKET           s,
496     uint32_t         val
497     )
498 {
499     UINT32 netVal = htonl(val);
500 //
501     return WriteBytes(s, (char*)&netVal, 4);
502 }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

D.3.2.11. ReadUINT32()

Function to read 4 byte integer from socket.

```
503  BOOL
504  ReadUINT32(
505      SOCKET           s,
506      UINT32          *val
507  )
508  {
509      UINT32 netVal;
510  //
511      if (!ReadBytes(s, (char*)&netVal, 4))
512          return FALSE;
513      *val = ntohl(netVal);
514      return TRUE;
515  }
```

D.3.2.12. ReadVarBytes()

Get a UINT32-length-prepended binary array. Note that the 4-byte length is in network byte order (big-endian).

```
516  BOOL
517  ReadVarBytes(
518      SOCKET           s,
519      char            *buffer,
520      uint32_t        *BytesReceived,
521      int              MaxLen
522      )
523  {
524      int                     length;
525      BOOL                    res;
526  //
527      res = ReadBytes(s, (char*)&length, 4);
528      if(!res) return res;
529      length = ntohl(length);
530      *BytesReceived = length;
531      if(length > MaxLen)
532      {
533          printf("Buffer too big.  Client says %d\n", length);
534          return FALSE;
535      }
536      if(length == 0) return TRUE;
537      res = ReadBytes(s, buffer, length);
538      if(!res) return res;
539      return TRUE;
540  }
```

D.3.2.13. WriteVarBytes()

Send a UINT32-length-prepended binary array. Note that the 4-byte length is in network byte order (big-endian).

```
541  BOOL
542  WriteVarBytes(
543      SOCKET           s,
544      char            *buffer,
545      int              BytesToSend
546      )
547  {
548      uint32_t                netLength = htonl(BytesToSend);
```

```
549     BOOL res;
550 //
551     res = WriteBytes(s, (char*)&netLength, 4);
552     if(!res)
553         return res;
554     res = WriteBytes(s, buffer, BytesToSend);
555     if(!res)
556         return res;
557     return TRUE;
558 }
```

D.3.2.14. TpmServer()

Processing incoming TPM command requests using the protocol / interface defined above.

```
559 BOOL
560 TpmServer(
561     SOCKET           s
562     )
563 {
564     uint32_t              length;
565     uint32_t              Command;
566     BYTE                  locality;
567     BOOL                  OK;
568     int                   result;
569     int                   clientVersion;
570     _IN_BUFFER            InBuffer;
571     _OUT_BUFFER           OutBuffer;
572 //
573     for(;;)
574     {
575         OK = ReadBytes(s, (char*)&Command, 4);
576         // client disconnected (or other error).  We stop processing this client
577         // and return to our caller who can stop the server or listen for another
578         // connection.
579         if(!OK)
580             return TRUE;
581         Command = ntohl(Command);
582         switch(Command)
583         {
584             case TPM_SIGNAL_HASH_START:
585                 _rpc__Signal_Hash_Start();
586                 break;
587             case TPM_SIGNAL_HASH_END:
588                 _rpc__Signal_HashEnd();
589                 break;
590             case TPM_SIGNAL_HASH_DATA:
591                 OK = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
592                 if(!OK) return TRUE;
593                 InBuffer.Buffer = (BYTE*)InputBuffer;
594                 InBuffer.BufferSize = length;
595                 _rpc__Signal_Hash_Data(InBuffer);
596                 break;
597             case TPM_SEND_COMMAND:
598                 OK = ReadBytes(s, (char*)&locality, 1);
599                 if(!OK)
600                     return TRUE;
601                 OK = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
602                 if(!OK)
603                     return TRUE;
604                 InBuffer.Buffer = (BYTE*)InputBuffer;
605                 InBuffer.BufferSize = length;
606                 OutBuffer.BufferSize = MAX_BUFFER;
607                 OutBuffer.Buffer = (_OUTPUT_BUFFER)OutputBuffer;
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

```
608                 // record the number of bytes in the command if it is the largest
609                 // we have seen so far.
610                 if(InBuffer.BufferSize > CommandResponseSizes.largestCommandSize)
611                 {
612                     CommandResponseSizes.largestCommandSize = InBuffer.BufferSize;
613                     memcpy(&CommandResponseSizes.largestCommand,
614                            &InputBuffer[6], sizeof(UINT32));
615                 }
616                 _rpc__Send_Command(locality, InBuffer, &OutBuffer);
617                 // record the number of bytes in the response if it is the largest
618                 // we have seen so far.
619                 if(OutBuffer.BufferSize > CommandResponseSizes.largestResponseSize)
620                 {
621                     CommandResponseSizes.largestResponseSize
622                         = OutBuffer.BufferSize;
623                     memcpy(&CommandResponseSizes.largestResponse,
624                            &OutputBuffer[6], sizeof(UINT32));
625                 }
626                 OK = WriteVarBytes(s,
627                                    (char*)OutBuffer.Buffer,
628                                    OutBuffer.BufferSize);
629                 if(!OK)
630                     return TRUE;
631                 break;
632             case TPM_REMOTE_HANDSHAKE:
633                 OK = ReadBytes(s, (char*)&clientVersion, 4);
634                 if(!OK)
635                     return TRUE;
636                 if(clientVersion == 0)
637                 {
638                     printf("Unsupported client version (0).\n");
639                     return TRUE;
640                 }
641                 OK &= WriteUINT32(s, ServerVersion);
642                 OK &= WriteUINT32(s, tpmInRawMode
643                                   | tpmPlatformAvailable | tpmSupportsPP);
644                 break;
645             case TPM_SET_ALTERNATIVE_RESULT:
646                 OK = ReadBytes(s, (char*)&result, 4);
647                 if(!OK)
648                     return TRUE;
649                 // Alternative result is not applicable to the simulator.
650                 break;
651             case TPM_SESSION_END:
652                 // Client signaled end-of-session
653                 return TRUE;
654             case TPM_STOP:
655                 // Client requested the simulator to exit
656                 return FALSE;
657             default:
658                 printf("Unrecognized TPM interface command %d\n", (int)Command);
659                 return TRUE;
660         }
661         OK = WriteUINT32(s, 0);
662         if(!OK)
663             return TRUE;
664     }
665 }
```

 Copyright © TCG 2006-2020

D.4 TPMCmdp.c

D.4.1. Description

This file contains the functions that process the commands received on the control port or the command port of the simulator. The control port is used to allow simulation of hardware events (such as, _TPM_Hash_Start()) to test the simulated TPM's reaction to those events. This improves code coverage of the testing.

D.4.2. Includes and Data Definitions

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <setjmp.h>
4  #include "TpmBuildSwitches.h"
5  #ifdef _MSC_VER
6  #   pragma warning(push, 3)
7  #   include <windows.h>
8  #   include <winsock.h>
9  #   pragma warning(pop)
10 #elif defined(__unix__)
11     typedef int SOCKET;
12 #else
13 #   error "Unsupported platform."
14 #endif
15 #ifndef TRUE
16 #   define TRUE    1
17 #endif
18 #ifndef FALSE
19 #   define FALSE   0
20 #endif
21 #include "Platform_fp.h"
22 #include "ExecCommand_fp.h"
23 #include "Manufacture_fp.h"
24 #include "_TPM_Init_fp.h"
25 #include "_TPM_Hash_Start_fp.h"
26 #include "_TPM_Hash_Data_fp.h"
27 #include "_TPM_Hash_End_fp.h"
28 #include "TpmFail_fp.h"
29 #include "TpmTcpProtocol.h"
30 #include "Simulator_fp.h"
31 static BOOL     s_isPowerOn = FALSE;
32
33 //** Functions
34
35 //*** Signal_PowerOn()
36 // This function processes a power-on indication. Among other things, it
37 // calls the _TPM_Init() handler.
38 void
39 _rpc__Signal_PowerOn(
40     BOOL        isReset
41     )
42 {
43     // if power is on and this is not a call to do TPM reset then return
44     if(s_isPowerOn && !isReset)
45         return;
46     // If this is a reset but power is not on, then return
47     if(isReset && !s_isPowerOn)
48         return;
49     // Unless this is just a reset, pass power on signal to platform
50     if(!isReset)
51         _plat__Signal_PowerOn();
```

 Copyright © TCG 2006-2020

```
52      // Power on and reset both lead to _TPM_Init()
53      _plat__Signal_Reset();
54
55      // Set state as power on
56      s_isPowerOn = TRUE;
57  }
```

D.4.2.1. Signal_Restart()

This function processes the clock restart indication. All it does is call the platform function.

```
58  void
59  _rpc__Signal_Restart(
60      void
61      )
62  {
63      _plat__TimerRestart();
64  }
```

D.4.2.2. Signal_PowerOff()

This function processes the power off indication. Its primary function is to set a flag indicating that the next power on indication should cause _TPM_Init() to be called.

```
65  void
66  _rpc__Signal_PowerOff(
67      void
68      )
69  {
70      if(s_isPowerOn)
71          // Pass power off signal to platform
72          _plat__Signal_PowerOff();
73      // This could be redundant, but...
74      s_isPowerOn = FALSE;
75
76      return;
77  }
```

D.4.2.3. _rpc__ForceFailureMode()

This function is used to debug the Failure Mode logic of the TPM. It will set a flag in the TPM code such that the next call to TPM2_SelfTest() will result in a failure, putting the TPM into Failure Mode.

```
78  void
79  _rpc__ForceFailureMode(
80      void
81      )
82  {
83      SetForceFailureMode();
84      return;
85  }
```

D.4.2.4. _rpc__Signal_PhysicalPresenceOn()

This function is called to simulate activation of the physical presence **pin**.

```
86  void
87  _rpc__Signal_PhysicalPresenceOn(
88      void
```

```
89      )
90  {
91      // If TPM power is on
92      if(s_isPowerOn)
93          // Pass physical presence on to platform
94          _plat__Signal_PhysicalPresenceOn();
95      return;
96  }
```

D.4.2.5. _rpc__Signal_PhysicalPresenceOff()

This function is called to simulate deactivation of the physical presence **pin**.

```
97  void
98  _rpc__Signal_PhysicalPresenceOff(
99      void
100     )
101 {
102     // If TPM is power on
103     if(s_isPowerOn)
104         // Pass physical presence off to platform
105         _plat__Signal_PhysicalPresenceOff();
106     return;
107 }
```

D.4.2.6. _rpc__Signal_Hash_Start()

This function is called to simulate a _TPM_Hash_Start() event. It will call

```
108 void
109 _rpc__Signal_Hash_Start(
110     void
111     )
112 {
113     // If TPM power is on
114     if(s_isPowerOn)
115         // Pass _TPM_Hash_Start signal to TPM
116         _TPM_Hash_Start();
117     return;
118 }
```

D.4.2.7. _rpc__Signal_Hash_Data()

This function is called to simulate a _TPM_Hash_Data() event.

```
119 void
120 _rpc__Signal_Hash_Data(
121     _IN_BUFFER       input
122     )
123 {
124     // If TPM power is on
125     if(s_isPowerOn)
126         // Pass _TPM_Hash_Data signal to TPM
127         _TPM_Hash_Data(input.BufferSize, input.Buffer);
128     return;
129 }
```

D.4.2.8. _rpc__Signal_HashEnd()

This function is called to simulate a _TPM_Hash_End() event.

```
130 void
131 _rpc__Signal_HashEnd(
132     void
133     )
134 {
135     // If TPM power is on
136     if(s_isPowerOn)
137         // Pass _TPM_HashEnd signal to TPM
138         _TPM_Hash_End();
139     return;
140 }
```

D.4.2.9. _rpc__Send_Command()

This is the interface to the TPM code.

```
141 void
142 _rpc__Send_Command(
143     unsigned char    locality,
144     _IN_BUFFER       request,
145     _OUT_BUFFER     *response
146     )
147 {
148     // If TPM is power off, reject any commands.
149     if(!s_isPowerOn)
150     {
151         response->BufferSize = 0;
152         return;
153     }
154     // Set the locality of the command so that it doesn't change during the command
155     _plat__LocalitySet(locality);
156     // Do implementation-specific command dispatch
157     _plat__RunCommand(request.BufferSize, request.Buffer,
158                       &response->BufferSize, &response->Buffer);
159     return;
160 }
```

D.4.2.10. _rpc__Signal_CancelOn()

This function is used to turn on the indication to cancel a command in process. An executing command is not interrupted. The command code may periodically check this indication to see if it should abort the current command processing and returned TPM_RC_CANCELLED.

```
161 void
162 _rpc__Signal_CancelOn(
163     void
164     )
165 {
166     // If TPM power is on
167     if(s_isPowerOn)
168         // Set the platform canceling flag.
169         _plat__SetCancel();
170     return;
171 }
```

D.4.2.11. _rpc__Signal_CancelOff()

This function is used to turn off the indication to cancel a command in process.

```
172 void
173 _rpc__Signal_CancelOff(
```

 Copyright © TCG 2006-2020

```
174         void
175         )
176     {
177         // If TPM power is on
178         if(s_isPowerOn)
179             // Set the platform canceling flag.
180             _plat__ClearCancel();
181         return;
182     }
```

D.4.2.12. _rpc__Signal_NvOn()

In a system where the NV memory used by the TPM is not within the TPM, the NV may not always be available. This function turns on the indicator that indicates that NV is available.

```
183     void
184     _rpc__Signal_NvOn(
185         void
186         )
187     {
188         // If TPM power is on
189         if(s_isPowerOn)
190             // Make the NV available
191             _plat__SetNvAvail();
192         return;
193     }
```

D.4.2.13. _rpc__Signal_NvOff()

This function is used to set the indication that NV memory is no longer available.

```
194     void
195     _rpc__Signal_NvOff(
196         void
197         )
198     {
199         // If TPM power is on
200         if(s_isPowerOn)
201             // Make NV not available
202             _plat__ClearNvAvail();
203         return;
204     }
205     void RsaKeyCacheControl(int state);
```

D.4.2.14. _rpc__RsaKeyCacheControl()

This function is used to enable/disable the use of the RSA key cache during simulation.

```
206     void
207     _rpc__RsaKeyCacheControl(
208         int              state
209         )
210     {
211     #if USE_RSA_KEY_CACHE
212         RsaKeyCacheControl(state);
213     #else
214         NOT_REFERENCED(state);
215     #endif
216         return;
217     }
218     #define TPM_RH_ACT_0        0x40000110
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

D.4.2.15. _rpc__ACT_GetSignaled()

This function is used to count the ACT second tick.

```
219 BOOL
220 _rpc__ACT_GetSignaled(
221     UINT32 actHandle
222 )
223 {
224     // If TPM power is on
225     if (s_isPowerOn)
226         // Query the platform
227         return _plat__ACT_GetSignaled(actHandle - TPM_RH_ACT_0);
228     return FALSE;
229 }
```

D.5 TPMCmds.c

D.5.1. Description

This file contains the entry point for the simulator.

D.5.2. Includes, Defines, Data Definitions, and Function Prototypes

```
1   #include "TpmBuildSwitches.h"
2   #include <stdlib.h>
3   #include <stdio.h>
4   #include <stdint.h>
5   #include <ctype.h>
6   #include <string.h>
7   #ifdef _MSC_VER
8   #   pragma warning(push, 3)
9   #   include <windows.h>
10  #   include <winsock.h>
11  #   pragma warning(pop)
12  #elif defined(__unix__)
13  #   define _strcmpi strcasecmp
14      typedef int SOCKET;
15  #else
16  #   error "Unsupported platform."
17  #endif
18  #ifndef TRUE
19  #   define TRUE    1
20  #endif
21  #ifndef FALSE
22  #   define FALSE   0
23  #endif
24  #include "TpmTcpProtocol.h"
25  #include "Manufacture_fp.h"
26  #include "Platform_fp.h"
27  #include "Simulator_fp.h"
28  #define PURPOSE \
29  "TPM 2.0 Reference Simulator.\n"  \
30  "Copyright (c) Microsoft Corporation. All rights reserved."
31  #define DEFAULT_TPM_PORT 2321
```

Information about command line arguments (does not include program name)

```
32  static uint32_t     s_ArgsMask = 0;     // Bit mask of unmatched command line args
33  static int          s_Argc = 0;
34  static const char **s_Argv = NULL;
35
36  //** Functions
37
38  #if DEBUG
39  //*** Assert()
40  // This function implements a run-time assertion.
41  // Computation of its parameters must not result in any side effects, as these
42  // computations will be stripped from the release builds.
43  static void Assert (BOOL cond, const char* msg)
44  {
45      if (cond)
46          return;
47      fputs(msg, stderr);
48      exit(2);
49  }
50  #else
51  #define Assert(cond, msg)
```

```
52  #endif
```

D.5.2.1. Usage()

This function prints the proper calling sequence for the simulator.

```
53  static void
54  Usage(
55      const char          *programName
56  )
57  {
58      fprintf(stderr, "%s\n\n", PURPOSE);
59      fprintf(stderr, "Usage:  %s [PortNum] [opts]\n\n"
60          "Starts the TPM server listening on TCP port PortNum (by default %d).\n\n"
61          "An option can be in the short form (one letter preceded with '-' or '/')\n"
62          "or in the full form (preceded with '--' or no option marker at all).\n"
63          "Possible options are:\n"
64          "   -h (--help) or ? - print this message\n"
65          "   -m (--manufacture) - forces NV state of the TPM simulator to be "
66          "(re)manufactured\n",
67          programName, DEFAULT_TPM_PORT);
68      exit(1);
69  }
```

D.5.2.2. CmdLineParser_Init()

This function initializes command line option parser.

```
70  static BOOL
71  CmdLineParser_Init(
72      int argc,
73      char *argv[],
74      int maxOpts
75      )
76  {
77      if (argc == 1)
78          return FALSE;
79
80      if (maxOpts && (argc - 1) > maxOpts)
81      {
82          fprintf(stderr, "No more than %d options can be specified\n\n", maxOpts);
83          Usage(argv[0]);
84      }
85
86      s_Argc = argc - 1;
87      s_Argv = (const char**)(argv + 1);
88      s_ArgsMask = (1 << s_Argc) - 1;
89      return TRUE;
90  }
```

D.5.2.3. CmdLineParser_More()

Returns true if there are unparsed options still.

```
91  static BOOL
92  CmdLineParser_More(
93      void
94  )
95  {
96      return s_ArgsMask != 0;
97  }
```

D.5.2.4. CmdLineParser_IsOpt()

This function determines if the given command line parameter represents a valid option.

```
98   static BOOL
99   CmdLineParser_IsOpt(
100      const char* opt,         // Command line parameter to check
101      const char* optFull,     // Expected full name
102      const char* optShort,    // Expected short (single letter) name
103      BOOL dashed              // The parameter is preceded by a single dash
104      )
105  {
106      return 0 == strcmp(opt, optFull)
107          || (optShort && opt[0] == optShort[0] && opt[1] == 0)
108          || (dashed && opt[0] == '-' && 0 == strcmp(opt + 1, optFull));
109  }
```

D.5.2.5. CmdLineParser_IsOptPresent()

This function determines if the given command line parameter represents a valid option.

```
110  static BOOL
111  CmdLineParser_IsOptPresent(
112      const char* optFull,
113      const char* optShort
114      )
115  {
116      int          i;
117      int          curArgBit;
118      Assert(s_Argv != NULL,
119          "InitCmdLineOptParser(argc, argv) has not been invoked\n");
120      Assert(optFull && optFull[0],
121          "Full form of a command line option must be present.\n"
122          "If only a short (single letter) form is supported, it must be"
123          "specified as the full one.\n");
124      Assert(!optShort || (optShort[0] && !optShort[1]),
125          "If a short form of an option is specified, it must consist "
126          "of a single letter only.\n");
127
128      if (!CmdLineParser_More())
129          return FALSE;
130
131      for (i = 0, curArgBit = 1; i < s_Argc; ++i, curArgBit <<= 1)
132      {
133          const char* opt = s_Argv[i];
134          if (    (s_ArgsMask & curArgBit) && opt
135              && (    0 == strcmp(opt, optFull)
136                  || (    (opt[0] == '/' || opt[0] == '-')
137                      && CmdLineParser_IsOpt(opt + 1, optFull, optShort,
138                                             opt[0] == '-'))))
139          {
140              s_ArgsMask ^= curArgBit;
141              return TRUE;
142          }
143      }
144      return FALSE;
145  }
```

D.5.2.6. CmdLineParser_IsOptPresent()

This function notifies the parser that no more options are needed.

```
146  static void
147  CmdLineParser_Done(
148      const char          *programName
149  )
150  {
151      char delim = ':';
152      int         i;
153      int         curArgBit;
154
155      if (!CmdLineParser_More())
156          return;
157
158      fprintf(stderr, "Command line contains unknown option%s",
159              s_ArgsMask & (s_ArgsMask - 1) ? "s" : "");
160      for (i = 0, curArgBit = 1; i < s_Argc; ++i, curArgBit <<= 1)
161      {
162          if (s_ArgsMask & curArgBit)
163          {
164              fprintf(stderr, "%c %s", delim, s_Argv[i]);
165              delim = ',';
166          }
167      }
168      fprintf(stderr, "\n\n");
169      Usage(programName);
170  }
```

D.5.2.7. main()

This is the main entry point for the simulator. It registers the interface and starts listening for clients

```
171  int
172  main(
173      int              argc,
174      char            *argv[]
175      )
176  {
177      BOOL    manufacture = FALSE;
178      int     PortNum = DEFAULT_TPM_PORT;
179
180      // Parse command line options
181
182      if (CmdLineParser_Init(argc, argv, 2))
183      {
184          if (   CmdLineParser_IsOptPresent("?", "?")
185              || CmdLineParser_IsOptPresent("help", "h"))
186          {
187              Usage(argv[0]);
188          }
189          if (CmdLineParser_IsOptPresent("manufacture", "m"))
190          {
191              manufacture = TRUE;
192          }
193          if (CmdLineParser_More())
194          {
195              int     i;
196              for (i = 0; i < s_Argc; ++i)
197              {
198                  char *nptr = NULL;
199                  int portNum = (int)strtol(s_Argv[i], &nptr, 0);
200                  if (s_Argv[i] != nptr)
201                  {
202                      // A numeric option is found
203                      if(!*nptr && portNum > 0 && portNum < 65535)
204                      {
```

```
205                         PortNum = portNum;
206                         s_ArgsMask ^= 1 << i;
207                         break;
208                     }
209                     fprintf(stderr, "Invalid numeric option %s\n\n", s_Argv[i]);
210                     Usage(argv[0]);
211                 }
212             }
213         }
214         CmdLineParser_Done(argv[0]);
215     }
216     printf("LIBRARY_COMPATIBILITY_CHECK is %s\n",
217          (LIBRARY_COMPATIBILITY_CHECK ? "ON" : "OFF"));
218     // Enable NV memory
219     _plat__NVEnable(NULL);
220
221     if (manufacture || _plat__NVNeedsManufacture())
222     {
223         printf("Manufacturing NV state...\n");
224         if(TPM_Manufacture(1) != 0)
225         {
226             // if the manufacture didn't work, then make sure that the NV file doesn't
227             // survive. This prevents manufacturing failures from being ignored the
228             // next time the code is run.
229             _plat__NVDisable(1);
230             exit(1);
231         }
232         // Coverage test - repeated manufacturing attempt
233         if(TPM_Manufacture(0) != 1)
234         {
235             exit(2);
236         }
237         // Coverage test - re-manufacturing
238         TPM_TearDown();
239         if(TPM_Manufacture(1) != 0)
240         {
241             exit(3);
242         }
243     }
244     // Disable NV memory
245     _plat__NVDisable(0);
246
247     StartTcpServer(PortNum);
248     return EXIT_SUCCESS;
249 }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

Trusted Platform Module Library
Part 4: Supporting Routines

Family "2.0"

Level 00 Revision 01.59

November 8, 2019

Published

Contact: admin@trustedcomputinggroup.org

TCG Published

Copyright © TCG 2006-2020

Licenses and Notices

Copyright Licenses:

- Trusted Computing Group (TCG) grants to the user of the source code in this specification (the “Source Code”) a worldwide, irrevocable, nonexclusive, royalty free, copyright license to reproduce, create derivative works, distribute, display and perform the Source Code and derivative works thereof, and to grant others the rights granted herein.
- The TCG grants to the user of the other parts of the specification (other than the Source Code) the rights to reproduce, distribute, display, and perform the specification solely for the purpose of developing products based on such documents.

Source Code Distribution Conditions:

- Redistributions of Source Code must retain the above copyright licenses, this list of conditions and the following disclaimers.
- Redistributions in binary form must reproduce the above copyright licenses, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.

Disclaimers:

- THE COPYRIGHT LICENSES SET FORTH ABOVE DO NOT REPRESENT ANY FORM OF LICENSE OR WAIVER, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, WITH RESPECT TO PATENT RIGHTS HELD BY TCG MEMBERS (OR OTHER THIRD PARTIES) THAT MAY BE NECESSARY TO IMPLEMENT THIS SPECIFICATION OR OTHERWISE. Contact TCG Administration (admin@trustedcomputinggroup.org) for information on specification licensing rights available through TCG membership agreements.
- THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO EXPRESS OR IMPLIED WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, ACCURACY, COMPLETENESS, OR NONINFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.
- Without limitation, TCG and its members and licensors disclaim all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification and to the implementation of this specification, and TCG disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this specification or any information herein.

Any marks and brands contained herein are the property of their respective owners.

CONTENTS

1 Scope 1
2 Terms and definitions 1
3 Symbols and abbreviated terms 1
4 Automation 1
4.1 Configuration Parser 1
4.2 Structure Parser 2
4.2.1 Introduction 2
4.2.2 Unmarshaling Code Prototype 2
4.2.2.1 Simple Types and Structures 2
4.2.2.2 Union Types 3
4.2.2.3 Null Types 3
4.2.2.4 Arrays 3
4.2.3 Marshaling Code Function Prototypes 3
4.2.3.1 Simple Types and Structures 3
4.2.3.2 Union Types 4
4.2.3.3 Arrays 4
4.2.3.4 The generated code for an array uses a `count`-limited loop within which it calls the marshaling code for `TYPE`.Table-driven Marshaling 4
4.3 Part 3 Parsing 5
4.4 Function Prototypes 5
4.5 Portability 5
5 Header Files 7
5.1 Introduction 7
5.2 BaseTypes.h 7
5.3 Capabilities.h 8
5.4 CommandAttributeData.h 9
5.5 CommandAttributes.h 10
5.6 CommandDispatchData.h 11
5.7 Commands.h 12
5.8 CompilerDependencies.h 13
5.9 Global.h 14
5.10 GpMacros.h 15
5.11 InternalRoutines.h 16
5.12 LibSupport.h 17
5.13 MinMax.h 17
5.14 NV.h 18
5.15 TPMB.h 19
5.16 Tpm.h 20
5.17 TpmBuildSwitches.h 21
5.18 TpmError.h 22
5.19 TpmTypes.h 23
5.20 VendorString.h 24
5.21 swap.h 25
5.22 ACT.h 26
6 Main 27
6.1 Introduction 27
6.2 ExecCommand.c 27
6.3 CommandDispatcher.c 28
6.3.1 Introduction 28

DRAFT

6.4 SessionProcess.c 29

7 Command Support Functions 30

7.1 Introduction 30
7.2 Attestation Command Support (Attest_spt.c) 30
7.3 Context Management Command Support (Context_spt.c) 31
7.4 Policy Command Support (Policy_spt.c) 32
7.5 NV Command Support (NV_spt.c) 33
7.6 Object Command Support (Object_spt.c) 34
7.7 Encrypt Decrypt Support (EncryptDecrypt_spt.c) 35
7.8 ACT Support (ACT_spt.c) 36

8 Subsystem 37

8.1 CommandAudit.c 37
8.2 DA.c 38
8.3 Hierarchy.c 39
8.4 NvDynamic.c 40
8.5 NvReserved.c 41
8.6 Object.c 42
8.7 PCR.c 43
8.8 PP.c 44
8.9 Session.c 45
8.10 Time.c 46

9 Support 47

9.1 AlgorithmCap.c 47
9.2 Bits.c 48
9.3 CommandCodeAttributes.c 49
9.4 Entity.c 50
9.5 Global.c 51
9.6 Handle.c 52
9.7 IoBuffers.c 53
9.8 Locality.c 54
9.9 Manufacture.c 55
9.10 Marshal.c 56

9.10.1 Introduction 56
9.10.2 Unmarshal and Marshal a Value 56
9.10.3 Unmarshal and Marshal a Union 57
9.10.4 Unmarshal and Marshal a Structure 59
9.10.5 Unmarshal and Marshal an Array 60
9.10.6 TPM2B Handling 62
9.10.7 Table Marshal Headers 62

9.10.7.1 TableMarshal.h 62
9.10.7.2 TableMarshalData.h 63
9.10.7.3 TableMarshalDefines.h 63
9.10.7.4 TableMarshalTypes.h 63

9.10.8 Table Marshal Source 63

9.10.8.1 TableDrivenMarshal.c 63
9.10.8.2 TableMarshalData.c 63

9.11 MathOnByteBuffers.c 64
9.12 Memory.c 65
9.13 Power.c 66
9.14 PropertyCap.c 67
9.15 Response.c 68
9.16 ResponseCodeProcessing.c 69
9.17 TpmFail.c 70

 Copyright © TCG 2006-2020

10 Cryptographic Functions 71
10.1 Headers 71
10.1.1 BnValues.h 71
10.1.2 CryptEcc.h 72
10.1.3 CryptHash.h 73
10.1.4 CryptRand.h 74
10.1.5 CryptRsa.h 75
10.1.6 CryptTest.h 76
10.1.7 HashTestData.h 77
10.1.8 KdfTestData.h 78
10.1.9 RsaTestData.h 79
10.1.10 SelfTest.h 79
10.1.11 SupportLibraryFunctionPrototypes_fp.h 79
10.1.12 SymmetricTestData.h 80
10.1.13 SymmetricTest.h 81
10.1.14 EccTestData.h 82
10.1.15 CryptSym.h 83
10.1.16 OIDs.h 84
10.1.17 PRNG_TestVectors.h 84
10.1.18 TpmAsn1.h 84
10.1.19 X509.h 84
10.1.20 TpmAlgorithmDefines.h 84
10.2 Source 85
10.2.1 AlgorithmTests.c 85
10.2.2 BnConvert.c 86
10.2.3 BnMath.c 87
10.2.4 BnMemory.c 88
10.2.5 CryptCmac.c 89
10.2.6 CryptUtil.c 90
10.2.7 CryptSelfTest.c 91
10.2.8 CryptEccData.c 92
10.2.9 CryptDes.c 93
10.2.10 CryptEccKeyExchange.c 94
10.2.11 CryptEccMain.c 95
10.2.12 CryptEccSignature.c 96
10.2.13 CryptHash.c 97
10.2.14 CryptPrime.c 98
10.2.15 CryptPrimeSieve.c 99
10.2.16 CryptRand.c 100
10.2.17 CryptRsa.c 101
10.2.18 CryptSmac.c 102
10.2.19 CryptSym.c 103
10.2.20 PrimeData.c 104
10.2.21 RsaKeyCache.c 105
10.2.22 Ticket.c 106
10.2.23 TpmAsn1.c 107
10.2.24 X509_ECC.c 108
10.2.25 X509_RSA.c 109
10.2.26 X509_spt.c 110
10.2.27 AC_spt.c 111
Annex A (informative) Implementation Dependent 112
A.1 Introduction 112
A.2 TpmProfile.h 112
A.3 TpmSizeChecks.c 112
Annex B (informative) Library-Specific 113

DRAFT

B.1 Introduction 113
B.2 OpenSSL-Specific Files 114
B.2.1. Introduction 114
B.2.2. Header Files 114
B.2.2.1. TpmToOsslHash.h 114
B.2.2.2. TpmToOsslMath.h 115
B.2.2.3. TpmToOsslSym.h 116
B.2.3. Source Files 117
B.2.3.1. TpmToOsslDesSupport.c 117
B.2.3.2. TpmToOsslMath.c 118
B.2.3.3. TpmToOsslSupport.c 119
Annex C (informative) Simulation Environment 120
C.1 Introduction 120
C.2 Cancel.c 120
C.3 Clock.c 121
C.4 Entropy.c 122
C.5 LocalityPlat.c 123
C.6 NVMem.c 124
C.7 PowerPlat.c 125
C.8 PlatformData.h 126
C.9 PlatformData.c 127
C.10 PPPlat.c 128
C.11 RunCommand.c 129
C.12 Unique.c 130
C.13 DebugHelpers.c 131
C.14 Platform.h 132
C.15 PlatformACT.h 133
C.16 PlatformACT.c 134
C.17 PlatformClock.h 135
Annex D (informative) Remote Procedure Interface 136
D.1 Introduction 136
D.2 TpmTcpProtocol.h 137
D.3 TcpServer.c 138
D.4 TPMCmdp.c 139
D.5 TPMCmds.c 140

Trusted Platform Module Library
Part 4: Supporting Routines

1 Scope

This part contains C code that describes the algorithms and methods used by the command code in TPM 2.0 Part 3. The code in this document augments TPM 2.0 Part 2 and TPM 2.0 Part 3 to provide a complete description of a TPM, including the supporting framework for the code that performs the command actions.

Any TPM 2.0 Part 4 code may be replaced by code that provides similar results when interfacing to the action code in TPM 2.0 Part 3. The behavior of code in this document that is not included in an annex is *normative*, as observed at the interfaces with TPM 2.0 Part 3 code. Code in an annex is provided for completeness, that is, to allow a full implementation of the specification from the provided code.

The code in parts 3 and 4 is written to define the behavior of a compliant TPM. In some cases (e.g., firmware update), it is not possible to provide a compliant implementation. In those cases, any implementation provided by the vendor that meets the general description of the function provided in TPM 2.0 Part 3 would be compliant.

The code in parts 3 and 4 is not written to meet any particular level of conformance nor does this specification require that a TPM meet any particular level of conformance.

2 Terms and definitions

For the purposes of this document, the terms and definitions given in TPM 2.0 Part 1 apply.

3 Symbols and abbreviated terms

For the purposes of this document, the symbols and abbreviated terms given in TPM 2.0 Part 1 apply.

4 Automation

TPM 2.0 Part 2 and 3 are constructed so that they can be processed by an automated parser. For example, TPM 2.0 Part 2 can be processed to generate header file contents such as structures, typedefs, and enums. TPM 2.0 Part 3 can be processed to generate command and response marshaling and unmarshaling code.

The automated processor is not provided by the TCG. It was used to generate the Microsoft Visual Studio TPM simulator files. These files are not specification reference code, but rather design examples.

The automation produces TPM_Types.h, a header representing TPM 2.0 Part 2. It also produces, for each major clause of Part 4, a header of the form _fp.h with the function prototypes.

EXAMPLE The header file for SessionProcess.c is SessionProcess_fp.h.

4.1 Configuration Parser

The TPM configuration is largely defined by TpmProfiles.h. This file may be edited in order to change the algorithms and commands supported by a TPM implementation.

A parser exists to process a Word document that defines the TPM configuration. This parser is used to create TpmProfiles.h.

 Copyright © TCG 2006-2020

4.2 Structure Parser

4.2.1 Introduction

The program that processes the tables in TPM 2.0 Part 2 is called "The TPM 2.0 Part 2 Structure Parser."

NOTE A Perl script was used to parse the tables in TPM 2.0 Part 2 to produce the header files and unmarshaling code in for the reference implementation.

The TPM 2.0 Part 2 Structure Parser takes as input the files produced by the TPM 2.0 Part 2 Configuration Parser and the same TPM 2.0 Part 2 specification that was used as input to the TPM 2.0 Part 2 Configuration Parser. The TPM 2.0 Part 2 Structure Parser will generate all of the C structure constant definitions that are required by the TPM interface. Additionally, the parser will generate unmarshaling code for all structures passed to the TPM, and marshaling code for structures passed from the TPM.

The unmarshaling code produced by the parser uses the prototypes defined below. The unmarshaling code will perform validations of the data to ensure that it is compliant with the limitations on the data imposed by the structure definition and use the response code provided in the table if not.

EXAMPLE: The definition for a TPMI_RH_PROVISION indicates that the primitive data type is a TPM_HANDLE and the only allowed values are TPM_RH_OWNER and TPM_RH_PLATFORM. The definition also indicates that the TPM shall indicate TPM_RC_HANDLE if the input value is not none of these values. The unmarshaling code will validate that the input value has one of those allowed values and return TPM_RC_HANDLE if not.

The sections below describe the function prototypes for the marshaling and unmarshaling code that is automatically generated by the TPM 2.0 Part 2 Structure Parser. These prototypes are described here as the unmarshaling and marshaling of various types occurs in places other than when the command is being parsed or the response is being built. The prototypes and the description of the interface are intended to aid in the comprehension of the code that uses these auto-generated routines.

4.2.2 Unmarshaling Code Prototype

4.2.2.1 Simple Types and Structures

The general form for the unmarshaling code for a simple type or a structure is:

```
TPM_RC TYPE_Unmarshal(TYPE *target, BYTE **buffer, INT32 *size);
```

Where:

TYPE	name of the data type or structure
***target**	location in the TPM memory into which the data from ****buffer** is placed
****buffer**	location in input buffer containing the most significant octet (MSO) of ***target**
***size**	number of octets remaining in ****buffer**

When the data is successfully unmarshaled, the called routine will return TPM_RC_SUCCESS. Otherwise, it will return a Format-One response code (see TPM 2.0 Part 2).

If the data is successfully unmarshaled, ***buffer** is advanced point to the first octet of the next parameter in the input buffer and **size** is reduced by the number of octets removed from the buffer.

When the data type is a simple type, the parser will generate code that will unmarshal the underlying type and then perform checks on the type as indicated by the type definition.

When the data type is a structure, the parser will generate code that unmarshals each of the structure elements in turn and performs any additional parameter checks as indicated by the data type.

4.2.2.2 Union Types

When a union is defined, an extra parameter is defined for the unmarshaling code. This parameter is the selector for the type. The unmarshaling code for the union will unmarshal the type indicated by the selector.

The function prototype for a union has the form:

```
TPM_RC TYPE_Unmarshal(TYPE *target, BYTE **buffer, INT32 *size, UINT32 selector);
```

where:

TYPE	name of the union type or structure
***target**	location in the TPM memory into which the data from ****buffer** is placed
****buffer**	location in input buffer containing the most significant octet (MSO) of ***target**
***size**	number of octets remaining in ****buffer**
selector	union selector that determines what will be unmarshaled into ***target**

4.2.2.3 Null Types

In some cases, the structure definition allows an optional "null" value. The "null" value allows the use of the same C type for the entity even though it does not always have the same members.

For example, the TPMI_ALG_HASH data type is used in many places. In some cases, TPM_ALG_NULL is permitted and in some cases it is not. If two different data types had to be defined, the interfaces and code would become more complex because of the number of cast operations that would be necessary. Rather than encumber the code, the "null" value is defined and the unmarshaling code is given a flag to indicate if this instance of the type accepts the "null" parameter or not. When the data type has a "null" value, the function prototype is

```
TPM_RC TYPE_Unmarshal(TYPE *target, BYTE **buffer, INT32 *size, BOOL flag);
```

The parser detects when the type allows a "null" value and will always include **flag** in any call to unmarshal that type. **flag** TRUE indicates that null is accepted.

4.2.2.4 Arrays

Any data type may be included in an array. The function prototype use to unmarshal an array for a **TYPE** is

```
TPM_RC TYPE_Array_Unmarshal(TYPE *target, BYTE **buffer, INT32 *size,INT32 count);
```

The generated code for an array uses a **count**-limited loop within which it calls the unmarshaling code for **TYPE**.

4.2.3 Marshaling Code Function Prototypes

4.2.3.1 Simple Types and Structures

The general form for the marshaling code for a simple type or a structure is:

```
UINT16 TYPE_Marshal(TYPE *source, BYTE **buffer, INT32 *size);
```

Where:

TYPE	name of the data type or structure
***source**	location in the TPM memory containing the value that is to be marshaled in to the designated buffer
****buffer**	location in the output buffer where the first octet of the **TYPE** is to be placed
***size**	number of octets remaining in ****buffer**.

If **buffer** is a NULL pointer, then no data is marshaled, but the routine will compute and return the size of the memory required to marshal the indicated type. ***size** is not changed.

If **buffer** is not a NULL pointer, data is marshaled, ***buffer** is advanced to point to the first octet of the next location in the output buffer, and the called routine will return the number of octets marshaled into ****buffer**. This occurs even if **size** is a NULL pointer. If **size** is a not NULL pointer ***size** is reduced by the number of octets placed in the buffer.

When the data type is a simple type, the parser will generate code that will marshal the underlying type. The presumption is that the TPM internal structures are consistent and correct so the marshaling code does not validate that the data placed in the buffer has a permissible value. The presumption is also that the **size** is sufficient for the source being marshaled.

When the data type is a structure, the parser will generate code that marshals each of the structure elements in turn.

4.2.3.2 Union Types

An extra parameter is defined for the marshaling function of a union. This parameter is the selector for the type. The marshaling code for the union will marshal the type indicated by the selector.

The function prototype for a union has the form:

```
UINT16 TYPE_Marshal(TYPE *source, BYTE **buffer, INT32 *size, UINT32 selector);
```

The parameters have a similar meaning as those in 4.2.2.2 but the data movement is from **source** to **buffer**.

4.2.3.3 Arrays

Any type may be included in an array. The function prototype use to unmarshal an array is:

```
UINT16 TYPE_Array_Marshal(TYPE *source, BYTE **buffer, INT32 *size, INT32 count);
```

4.2.3.4 The generated code for an array uses a count-limited loop within which it calls the marshaling code for TYPE.Table-driven Marshaling

The most recent versions of the TPM code includes the option to use table-driven marshaling rather that the procedural marshaling described in previous clauses in 4.2.2. The structure and processing of this code is complex and is provided in the code.

 Copyright © TCG 2006-2020

4.3 Part 3 Parsing

The Command / Response tables in Part 3 of this specification are processed by scripts to produce the command-specific data structures used by functions in this TPM 2.0 Part 4. They are:

- **CommandAttributeData.h** -- This file contains the command attributes reported by TPM2_GetCapability.
- **CommandAttributes.h** – This file contains the definition of command attributes that are extracted by the parsing code. The file mainly exists to ensure that the parsing code and the function code are using the same attributes.
- **CommandDispatchData.h** – This file contains the data definitions for the table driven version of the command dispatcher.

Part 3 parsing also produces special function prototype files as described in 4.4.

4.4 Function Prototypes

For functions that have entry definitions not defined by Part 3 tables. a script is used to extracts function prototypes from the code. For each .c file that is not in Part 3, a file with the same name is created with a suffix of _fp.h. For example, the function prototypes for Create.c will be placed in a file called Create_fp.h. The _fp.h is added because some files have two types of associated headers: the one containing the function prototypes for the file and another containing definitions that are specific to that file.

In some cases, a function will be replaced by a macro. The macro is defined in the .c file and extracted by the function prototype processor. A special comment tag ("//%") is used to indicate that the line is to be included in the function prototype file. If the "//%" tag occurs at the start of the line, it is deleted. If it occurs later in the line, it is preserved. Removing the "//%/ at the start of the line allows the macro to be placed in the .c file with the tag as a prefix, and then show up in the _fp.h file as the actual macro. This allows the code that includes that function prototype code to use the appropriate macro.

For files that that contain the command actions, a special _fp.h file is created from the tables in Part 3. These files contain:

- the definition of the input and output structure of the function;
- definition of command-specific return code modifiers (parameter identifiers); and
- the function prototype for the command action function.

Create_fp.h (shown below) is prototypical of the command _fp.h files.

`[[create_fp_h]]`

4.5 Portability

Where reasonable, the code is written to be portable. There are a few known cases where the code is not portable. Specifically, the handling of bit fields will not always be portable. The bit fields are marshaled and unmarshaled as a simple element of the underlying type. For example, a TPMA_SESSION is defined as a bit field in an octet (BYTE). When sent on the interface a TPMA_SESSION will occupy one octet. When unmarshaled, it is unmarshaled as a UINT8. The ramifications of this are that a TPMA_SESSION will occupy the 0th octet of the structure in which it is placed regardless of the size of the structure.

Many compilers will pad a bit field to some "natural" size for the processor, often 4 octets, meaning that `sizeof(TPMA_SESSION)` would return 4 rather than 1 (the canonical size of a TPMA_SESSION).

For a little endian machine, padding of bit fields should have little consequence since the 0th octet always contains the 0th bit of the structure no matter how large the structure. However, for a big endian machine, the 0th bit will be in the highest numbered octet. When unmarshaling a TPMA_SESSION, the current

 Copyright © TCG 2006-2020

unmarshaling code will place the input octet at the 0th octet of the TPMA_SESSION. Since the 0th octet is most significant octet, this has the effect of shifting all the session attribute bits left by 24 places.

As a consequence, someone implementing on a big endian machine should do one of two things:

a) allocate all structures as packed to a byte boundary (this may not be possible if the processor does not handle unaligned accesses); or

b) modify the code that manipulates bit fields that are not defined as being the alignment size of the system.

For many RISC processors, option #2 would be the only choice. This is may not be a terribly daunting task since only two attribute structures are not 32-bits (TPMA_SESSION and TPMA_LOCALITY).

 Copyright © TCG 2006-2020

5 Header Files

5.1 Introduction

The files in this section are used to define values that are used in multiple parts of the specification and are not confined to a single module.

5.2 BaseTypes.h

[[BaseTypes_h]]

5.3 Capabilities.h

This file contains defines for the number of capability values that will fit into the largest data buffer.

These defines are used in various function in the "support" and the "subsystem" code groups. A module that supports a type that is returned by a capability will have a function that returns the capabilities of the type.

EXAMPLE PCR.c contains PCRCapGetHandles() and PCRCapGetProperties().

[[Capabilities_h]]

 Copyright © TCG 2006-2020

5.4 CommandAttributeData.h

[[CommandAttributeData_h]]

5.5 CommandAttributes.h

[[CommandAttributes_h]]

 Copyright © TCG 2006-2020

5.6 CommandDispatchData.h

[[CommandDispatchData_h]]

Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

5.7 Commands.h

[[Commands_h]]

5.8 CompilerDependencies.h

[[CompilerDependencies_h]]

5.9 Global.h

[[Global_h]]

5.10 GpMacros.h

[[GpMacros_h]]

5.11 InternalRoutines.h

[[InternalRoutines_h]]

5.12 LibSupport.h

[[LibSupport_h]]

5.13 MinMax.h

[[MinMax_h]]

5.14 NV.h

[[NV_h]]

5.15 TPMB.h

[[TPMB_h]]

Copyright © TCG 2006-2020

5.16 Tpm.h

[[Tpm_h]]

5.17 TpmBuildSwitches.h

[[TpmBuildSwitches_h]]

 Copyright © TCG 2006-2020

5.18 TpmError.h

[[TpmError_h]]

 Copyright © TCG 2006-2020

5.19 TpmTypes.h

[[TpmTypes_h]]

5.20 VendorString.h

[[VendorString_h]]

5.21 swap.h

[[swap_h]]

 Copyright © TCG 2006-2020

5.22 ACT.h

[[ACT_h]]

6 Main

6.1 Introduction

The files in this section are the main processing blocks for the TPM. ExecuteCommand.c contains the entry point into the TPM code and the parsing of the command header. SessionProcess.c handles the parsing of the session area and the authorization checks, and CommandDispatch.c does the parameter unmarshaling and command dispatch.

6.2 ExecCommand.c

[[ExecCommand]]

 Copyright © TCG 2006-2020

6.3 CommandDispatcher.c

6.3.1 Introduction

CommandDispatcher() performs the following operations:

- unmarshals command parameters from the input buffer;

NOTE 1 Unlike other unmarshaling functions, *parmBufferStart* does not advance. *parmBufferSize* Is reduced.

- invokes the function that performs the command actions;
- marshals the returned handles, if any; and
- marshals the returned parameters, if any, into the output buffer putting in the *parameterSize* field if authorization sessions are present.

NOTE 2 The output buffer is the return from the MemoryGetResponseBuffer() function. It includes the header, handles, response parameters, and authorization area. *respParmSize* is the response parameter size, and does not include the header, handles, or authorization area.

NOTE 3 The reference implementation is permitted to do compare operations over a union as a byte array. Therefore, the command parameter *in* structure must be initialized (e.g., zeroed) before unmarshaling so that the compare operation is valid in cases where some bytes are unused.

[[CommandDispatcher]]

6.4 SessionProcess.c

[[SessionProcess]]

 Copyright © TCG 2006-2020

7 Command Support Functions

7.1 Introduction

This clause contains support routines that are called by the command action code in TPM 2.0 Part 3. The functions are grouped by the command group that is supported by the functions.

7.2 Attestation Command Support (Attest_spt.c)

[[Attest_spt]]

 Copyright © TCG 2006-2020

7.3 Context Management Command Support (Context_spt.c)

[[Context_spt]]

7.4 Policy Command Support (Policy_spt.c)

[[Policy_spt]]

 Copyright © TCG 2006-2020

7.5 NV Command Support (NV_spt.c)

[[NV_spt]]

 Copyright © TCG 2006-2020

7.6 Object Command Support (Object_spt.c)

[[Object_spt]]

7.7 Encrypt Decrypt Support (EncryptDecrypt_spt.c)

[[EncryptDecrypt_spt]]

7.8 ACT Support (ACT_spt.c)

[[ACT_spt]]

8 Subsystem

8.1 CommandAudit.c

[[CommandAudit]]

8.2 DA.c

[[DA]]

8.3 Hierarchy.c

[[Hierarchy]]

8.4 NvDynamic.c

[[NVDynamic]]

 Copyright © TCG 2006-2020

8.5 NvReserved.c

[[NVReserved]]

8.6 Object.c

[[Object]]

 Copyright © TCG 2006-2020

8.7 PCR.c

[[PCR]]

8.8 PP.c

[[PP]]

 Copyright © TCG 2006-2020

8.9 Session.c

[[Session]]

8.10 Time.c

[[Time]]

 Copyright © TCG 2006-2020

9 Support

9.1 AlgorithmCap.c

[[AlgorithmCap]]

9.2 Bits.c

[[Bits]]

9.3 CommandCodeAttributes.c

[[CommandCodeAttributes]]

 Copyright © TCG 2006-2020

9.4 Entity.c

[[Entity]]

9.5 Global.c

[[Global]]

 Copyright © TCG 2006-2020

9.6 Handle.c

[[Handle]]

9.7 IoBuffers.c

[[IoBuffers]]

9.8 Locality.c

[[Locality]]

9.9 Manufacture.c

[[Manufacture]]

Copyright © TCG 2006-2020

9.10 Marshal.c

9.10.1 Introduction

This file contains the marshaling and unmarshaling code.

The marshaling and unmarshaling code and function prototypes are not listed, as the code is repetitive, long, and not very useful to read. Examples of a few unmarshaling routines are provided. Most of the others are similar.

Depending on the table header flags, a type will have an unmarshaling routine and a marshaling routine The table header flags that control the generation of the unmarshaling and marshaling code are delimited by angle brackets ("<>") in the table header. If no brackets are present, then both unmarshaling and marshaling code is generated (i.e., generation of both marshaling and unmarshaling code is the default).

9.10.2 Unmarshal and Marshal a Value

In TPM 2.0 Part 2, a TPMI_DI_OBJECT is defined by this table:

Table xxx — Definition of (TPM_HANDLE) TPMI_DH_OBJECT Type

Values	Comments
{TRANSIENT_FIRST:TRANSIENT_LAST}	allowed range for transient objects
{PERSISTENT_FIRST:PERSISTENT_LAST}	allowed range for persistent objects
+TPM_RH_NULL	the null handle
#TPM_RC_VALUE	

This generates the following unmarshaling code:

```
1   TPM_RC
2   TPMI_DH_OBJECT_Unmarshal(TPMI_DH_OBJECT *target, BYTE **buffer, INT32 *size,
3                            BOOL flag)
4   {
5       TPM_RC    result;
6       result = TPM_HANDLE_Unmarshal((TPM_HANDLE *)target, buffer, size);
7       if(result != TPM_RC_SUCCESS)
8           return result;
9       if(*target == TPM_RH_NULL)
10      {
11          if(flag)
12              return TPM_RC_SUCCESS;
13          else
14              return TPM_RC_VALUE;
15      }
16      if(((*target < TRANSIENT_FIRST) || (*target > TRANSIENT_LAST))
17        &&((*target < PERSISTENT_FIRST) || (*target > PERSISTENT_LAST)))
18              return TPM_RC_VALUE;
19      return TPM_RC_SUCCESS;
20  }
```

and the following marshaling code:

NOTE The marshaling code does not do parameter checking, as the TPM is the source of the marshaling data.

```
1   UINT16
2   TPMI_DH_OBJECT_Marshal(TPMI_DH_OBJECT *source, BYTE **buffer, INT32 *size)
```

```
3    {
4        return UINT32_Marshal((UINT32 *)source, buffer, size);
5    }
```

An additional script is used to do the work that might be done by a linker or globally optimizing compiler. It searches for functions like TPMI_DH_OBJECT_Marshal() that do nothing but call another function and replaces the function with a #define.

```
6    #define TPMI_DH_OBJECT_Marshal(source, buffer, size)     \
7          UINT32_Marshal((UINT32 *)source, buffer, size)
```

When replacing the function with a #define, the #define is placed in marshal_fp.h and the function body is removed from marshal.c.

9.10.3 Unmarshal and Marshal a Union

In TPM 2.0 Part 2, a TPMU_PUBLIC_PARMS union is defined by:

Table xxx — Definition of TPMU_PUBLIC_PARMS Union <IN/OUT, S>

Parameter	Type	Selector	Description
keyedHash	TPMS_KEYEDHASH_PARMS	TPM_ALG_KEYEDHASH	sign | encrypt | neither
symDetail	TPMT_SYM_DEF_OBJECT	TPM_ALG_SYMCIPHER	a symmetric block cipher
rsaDetail	TPMS_RSA_PARMS	TPM_ALG_RSA	decrypt + sign
eccDetail	TPMS_ECC_PARMS	TPM_ALG_ECC	decrypt + sign
asymDetail	TPMS_ASYM_PARMS		common scheme structure for RSA and ECC keys
NOTE The Description column indicates which of TPMA_OBJECT.*decrypt* or TPMA_OBJECT.*sign* may be set. "+" indicates that both may be set but one shall be set. "|" indicates the optional settings.			

From this table, the following unmarshaling code is generated.

```
1    TPM_RC
2    TPMU_PUBLIC_PARMS_Unmarshal(TPMU_PUBLIC_PARMS *target, BYTE **buffer, INT32 *size,
3                                UINT32 selector)
4    {
5        switch(selector) {
6    #if ALG_KEYEDHASH
7            case TPM_ALG_KEYEDHASH:
8                return TPMS_KEYEDHASH_PARMS_Unmarshal(
9                           (TPMS_KEYEDHASH_PARMS *)&(target->keyedHash), buffer, size);
10   #endif
11   #if ALG_SYMCIPHER
12           case TPM_ALG_SYMCIPHER:
13               return TPMT_SYM_DEF_OBJECT_Unmarshal(
14                       (TPMT_SYM_DEF_OBJECT *)&(target->symDetail), buffer, size, FALSE);
15   #endif
16   #if ALG_RSA
17           case TPM_ALG_RSA:
18               return TPMS_RSA_PARMS_Unmarshal(
19                               (TPMS_RSA_PARMS *)&(target->rsaDetail), buffer, size);
20   #endif
21   #if ALG_ECC
22           case TPM_ALG_ECC:
23               return TPMS_ECC_PARMS_Unmarshal(
24                               (TPMS_ECC_PARMS *)&(target->eccDetail), buffer, size);
25   #endif
26       }
```

```
27      return TPM_RC_SELECTOR;
28  }
```

NOTE The `#if/#endif` directives are added whenever a value is dependent on an algorithm ID so that removing the algorithm definition will remove the related code.

The marshaling code for the union is:

```
1   UINT16
2   TPMU_PUBLIC_PARMS_Marshal(TPMU_PUBLIC_PARMS *source, BYTE **buffer, INT32 *size,
3                             UINT32 selector)
4   {
5       switch(selector) {
6   #if ALG_KEYEDHASH
7           case TPM_ALG_KEYEDHASH:
8               return TPMS_KEYEDHASH_PARMS_Marshal(
9                           (TPMS_KEYEDHASH_PARMS *)&(source->keyedHash), buffer, size);
10  #endif
11  #if ALG_SYMCIPHER
12          case TPM_ALG_SYMCIPHER:
13              return TPMT_SYM_DEF_OBJECT_Marshal(
14                          (TPMT_SYM_DEF_OBJECT *)&(source->symDetail), buffer, size);
15  #endif
16  #if ALG_RSA
17          case TPM_ALG_RSA:
18              return TPMS_RSA_PARMS_Marshal(
19                          (TPMS_RSA_PARMS *)&(source->rsaDetail), buffer, size);
20  #endif
21  #if ALG_ECC
22          case TPM_ALG_ECC:
23              return TPMS_ECC_PARMS_Marshal(
24                          (TPMS_ECC_PARMS *)&(source->eccDetail), buffer, size);
25  #endif
26      }
27      assert(1);
28      return 0;
29  }
```

For the marshaling and unmarshaling code, a value in the structure containing the union provides the value used for *selector*. The example in the next section illustrates this.

 Copyright © TCG 2006-2020

9.10.4 Unmarshal and Marshal a Structure

In TPM 2.0 Part 2, the TPMT_PUBLIC structure is defined by:

Table xxx — Definition of TPMT_PUBLIC Structure

Parameter	Type	Description
type	TPMI_ALG_PUBLIC	“algorithm” associated with this object
nameAlg	+TPMI_ALG_HASH	algorithm used for computing the Name of the object NOTE The "+" indicates that the instance of a TPMT_PUBLIC may have a "+" to indicate that the nameAlg may be TPM_ALG_NULL.
objectAttributes	TPMA_OBJECT	attributes that, along with *type*, determine the manipulations of this object
authPolicy	TPM2B_DIGEST	optional policy for using this key The policy is computed using the *nameAlg* of the object. NOTE shall be the Empty Buffer if no authorization policy is present
[type]parameters	TPMU_PUBLIC_PARMS	the algorithm or structure details
[type]unique	TPMU_PUBLIC_ID	the unique identifier of the structure For an asymmetric key, this would be the public key.

This structure is tagged (the first value indicates the structure type), and that tag is used to determine how the parameters and unique fields are unmarshaled and marshaled. The use of the type for specifying the union selector is emphasized below.

The unmarshaling code for the structure in the table above is:

```
1   TPM_RC
2   TPMT_PUBLIC_Unmarshal(TPMT_PUBLIC *target, BYTE **buffer, INT32 *size, BOOL flag)
3   {
4       TPM_RC     result;
5       result = TPMI_ALG_PUBLIC_Unmarshal((TPMI_ALG_PUBLIC *)&(target->type),
6                                          buffer, size);
7       if(result != TPM_RC_SUCCESS)
8           return result;
9       result = TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH *)&(target->nameAlg),
10                                       buffer, size, flag);
11      if(result != TPM_RC_SUCCESS)
12          return result;
13      result = TPMA_OBJECT_Unmarshal((TPMA_OBJECT *)&(target->objectAttributes),
14                                     buffer, size);
15      if(result != TPM_RC_SUCCESS)
16          return result;
17      result = TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST *)&(target->authPolicy),
18                                      buffer, size);
19      if(result != TPM_RC_SUCCESS)
20          return result;
21
22      result = TPMU_PUBLIC_PARMS_Unmarshal((TPMU_PUBLIC_PARMS *)&(target->parameters),
23                                           buffer, size, (UINT32)target->type);
24      if(result != TPM_RC_SUCCESS)
25          return result;
26
27      result = TPMU_PUBLIC_ID_Unmarshal((TPMU_PUBLIC_ID *)&(target->unique),
28                                        buffer, size, (UINT32)target->type);
29      if(result != TPM_RC_SUCCESS)
30          return result;
31
32      return TPM_RC_SUCCESS;
33  }
```


Level 00 Revision 01.59 Copyright © TCG 2006-2020 November 8, 2019

The marshaling code for the TPMT_PUBLIC structure is:

```
1  UINT16
2  TPMT_PUBLIC_Marshal(TPMT_PUBLIC *source, BYTE **buffer, INT32 *size)
3  {
4      UINT16    result = 0;
5      result = (UINT16)(result + TPMI_ALG_PUBLIC_Marshal(
6                                  (TPMI_ALG_PUBLIC *)&(source->type), buffer, size));
7      result = (UINT16)(result + TPMI_ALG_HASH_Marshal(
8                                  (TPMI_ALG_HASH *)&(source->nameAlg), buffer, size))
9  ;
10     result = (UINT16)(result + TPMA_OBJECT_Marshal(
11                           (TPMA_OBJECT *)&(source->objectAttributes), buffer, size));
12
13     result = (UINT16)(result + TPM2B_DIGEST_Marshal(
14                               (TPM2B_DIGEST *)&(source->authPolicy), buffer, size));
15
16     result = (UINT16)(result + TPMU_PUBLIC_PARMS_Marshal(
17                           (TPMU_PUBLIC_PARMS *)&(source->parameters), buffer, size,
18                                                            (UINT32)source->type));
19
20     result = (UINT16)(result + TPMU_PUBLIC_ID_Marshal(
21                                 (TPMU_PUBLIC_ID *)&(source->unique), buffer, size,
22                                                            (UINT32)source->type));
23
24     return result;
25 }
```

9.10.5 Unmarshal and Marshal an Array

In TPM 2.0 Part 2, the TPML_DIGEST is defined by:

Table xxx — Definition of TPML_DIGEST Structure

Parameter	Type	Description
count {2:}	UINT32	number of digests in the list, minimum is two
digests[count]{:8}	TPM2B_DIGEST	a list of digests For TPM2_PolicyOR(), all digests will have been computed using the digest of the policy session. For TPM2_PCR_Read(), each digest will be the size of the digest for the bank containing the PCR.
#TPM_RC_SIZE		response code when count is not at least two or is greater than 8

The *digests* parameter is an array of up to *count* structures (TPM2B_DIGESTS). The auto-generated code to Unmarshal this structure is:

```
1  TPM_RC
2  TPML_DIGEST_Unmarshal(TPML_DIGEST *target, BYTE **buffer, INT32 *size)
3  {
4      TPM_RC    result;
5      result = UINT32_Unmarshal((UINT32 *)&(target->count), buffer, size);
6      if(result != TPM_RC_SUCCESS)
7          return result;
8
9      if( (target->count < 2))     // This check is triggered by the {2:} notation
10                                  // on 'count'
11         return TPM_RC_SIZE;
12
13     if((target->count) > 8)      // This check is triggered by the {:8} notation
```

 Copyright © TCG 2006-2020

```
14                                   // on 'digests'.
15             return TPM_RC_SIZE;
16
17         result = TPM2B_DIGEST_Array_Unmarshal((TPM2B_DIGEST *)(target->digests),
18                                               buffer, size, (INT32)(target->count));
19         if(result != TPM_RC_SUCCESS)
20             return result;
21
22         return TPM_RC_SUCCESS;
23     }
```

The routine unmarshals a *count* value and passes that value to a routine that unmarshals an array of TPM2B_DIGEST values. The unmarshaling code for the array is:

```
1   TPM_RC
2   TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
3                                INT32 count)
4   {
5       TPM_RC    result;
6       INT32 i;
7       for(i = 0; i < count; i++) {
8           result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
9           if(result != TPM_RC_SUCCESS)
10              return result;
11      }
12      return TPM_RC_SUCCESS;
13  }
14
```

Marshaling of the TPML_DIGEST uses a similar scheme with a structure specifying the number of elements in an array and a subsequent call to a routine to marshal an array of that type.

```
1   UINT16
2   TPML_DIGEST_Marshal(TPML_DIGEST *source, BYTE **buffer, INT32 *size)
3   {
4       UINT16    result = 0;
5       result = (UINT16)(result + UINT32_Marshal((UINT32 *)&(source->count), buffer,
6                                                                                size));
7       result = (UINT16)(result + TPM2B_DIGEST_Array_Marshal(
8                                  (TPM2B_DIGEST *)(source->digests), buffer, size,
9                                  (INT32)(source->count)));
10
11      return result;
12  }
```

The marshaling code for the array is:

```
1   TPM_RC
2   TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
3                               INT32 count)
4   {
5       TPM_RC    result;
6       INT32 i;
7       for(i = 0; i < count; i++) {
8           result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
9           if(result != TPM_RC_SUCCESS)
10              return result;
11      }
12      return TPM_RC_SUCCESS;
13  }
```

 Copyright © TCG 2006-2020

9.10.6 TPM2B Handling

A TPM2B structure is handled as a special case. The unmarshaling code is similar to what is shown in 9.10.5 but the unmarshaling/marshaling is to a union element. Each TPM2B is a union of two sized buffers, one of which is type specific (the 't' element) and the other is a generic value (the 'b' element). This allows each of the TPM2B structures to have some inheritance property with all other TPM2B. The purpose is to allow functions that have parameters that can be any TPM2B structure while allowing other functions to be specific about the type of the TPM2B that is used. When the generic structure is allowed, the input parameter would use the 'b' element and when the type-specific structure is required, the 't' element is used.

When marshaling a TPM2B where the second member is a BYTE array, the size parameter indicates the size of the array. The second member can also be a structure. In this case, the caller does not prefill the size member. The marshaling code must marshal the structure and then back fill the calculated size.

Table xxx — Definition of TPM2B_EVENT Structure

Parameter	Type	Description
size	UINT16	Size of the operand
buffer [size] {:1024}	BYTE	The operand

```
1   TPM_RC
2   TPM2B_EVENT_Unmarshal(TPM2B_EVENT *target, BYTE **buffer, INT32 *size)
3   {
4       TPM_RC    result;
5       result = UINT16_Unmarshal((UINT16 *)&(target->t.size), buffer, size);
6       if(result != TPM_RC_SUCCESS)
7           return result;
8       // if size equal to 0, the rest of the structure is a zero buffer
9       //  so stop processing
10      if(target->t.size == 0)
11          return TPM_RC_SUCCESS;
12      if((target->t.size) > 1024)     // This check is triggered by the {:1024}
13                                      // notation on 'buffer'
14          return TPM_RC_SIZE;
15      result = BYTE_Array_Unmarshal((BYTE *)(target->t.buffer), buffer, size,
16                                    (INT32)(target->t.size));
17      if(result != TPM_RC_SUCCESS)
18          return result;
19      return TPM_RC_SUCCESS;
20  }
```

using these structure definitions:

```
1   typedef union {
2       struct {
3           UINT16        size;
4           BYTE          buffer[1024];
5       }             t;
6       TPM2B         b;
7   } TPM2B_EVENT;
```

9.10.7 Table Marshal Headers

9.10.7.1 TableMarshal.h

[[TableMarshal_h]]

9.10.7.2 TableMarshalData.h

[[TableMarshalData_h]]

9.10.7.3 TableMarshalDefines.h

[[TableMarshalDefines_h]]

9.10.7.4 TableMarshalTypes.h

[[TableMarshalTypes_h]]

9.10.8 Table Marshal Source

9.10.8.1 TableDrivenMarshal.c

[[TableDrivenMarshal]]

9.10.8.2 TableMarshalData.c

[[TableMarshalData]]

DRAFT

9.11 MathOnByteBuffers.c

[[MathOnByteBuffers]]

9.12 Memory.c

[[Memory]]

9.13 Power.c

[[Power]]

9.14 PropertyCap.c

[[PropertyCap]]

9.15 Response.c

[[Response]]

9.16 ResponseCodeProcessing.c

[[ResponseCodeProcessing]]

9.17 TpmFail.c

[[TpmFail]]

 Copyright © TCG 2006-2020

10 Cryptographic Functions

10.1 Headers

10.1.1 BnValues.h

[[BnValues_h]]

10.1.2 CryptEcc.h

[[CryptEcc_h]]

 Copyright © TCG 2006-2020

10.1.3 CryptHash.h

[[CryptHash_h]]

 Copyright © TCG 2006-2020

10.1.4 CryptRand.h

[[CryptRand_h]]

10.1.5 CryptRsa.h

[[CryptRsa_h]]

10.1.6 CryptTest.h

[[CryptTest_h]]

10.1.7 HashTestData.h

[[HashTestData_h]]

10.1.8 KdfTestData.h

[[KdfTestData_h]]

10.1.9 RsaTestData.h

[[RsaTestData_h]]

10.1.10 SelfTest.h

[[SelfTest_h]]

10.1.11 SupportLibraryFunctionPrototypes_fp.h

[[SupportLibraryFunctionPrototypes_fp_h]]

10.1.12 SymmetricTestData.h

[[SymmetricTestData_h]]

10.1.13 SymmetricTest.h

[[SymmetricTest_h]]

10.1.14 EccTestData.h

[[EccTestData_h]]

10.1.15 CryptSym.h

[[CryptSym_h]]

10.1.16 OIDs.h

[[OIDs_h]]

10.1.17 PRNG_TestVectors.h

[[PRNG_TestVectors_h]]

10.1.18 TpmAsn1.h

[[TpmAsn1_h]]

10.1.19 X509.h

[[X509_h]]

10.1.20 TpmAlgorithmDefines.h

This file contains the algorithm values from the TCG Algorithm Registry.

[[TpmAlgorithmDefines_h]]

10.2 Source

10.2.1 AlgorithmTests.c

[[AlgorithmTests]]

10.2.2 BnConvert.c

[[BnConvert]]

10.2.3 BnMath.c

[[BnMath]]

10.2.4 BnMemory.c

[[BnMemory]]

10.2.5 CryptCmac.c

[[CryptCmac]]

8

10.2.6 CryptUtil.c

[[CryptUtil]]

10.2.7 CryptSelfTest.c

[[CryptSelfTest]]

 Copyright © TCG 2006-2020

10.2.8 CryptEccData.c

[[CryptEccData]]

10.2.9 CryptDes.c

[[CryptDes]]

10.2.10 CryptEccKeyExchange.c

[[CryptEccKeyExchange]]

 Copyright © TCG 2006-2020

10.2.11 CryptEccMain.c

[[CryptEccMain]]

10.2.12 CryptEccSignature.c

[[CryptEccSignature]]

10.2.13 CryptHash.c

[[CryptHash]]

 Copyright © TCG 2006-2020

10.2.14 CryptPrime.c

[[CryptPrime]]

 Copyright © TCG 2006-2020

10.2.15 CryptPrimeSieve.c

[[CryptPrimeSieve]]

10.2.16 CryptRand.c

[[CryptRand]]

10.2.17 CryptRsa.c

[[CryptRsa]]

10.2.18 CryptSmac.c

[[CryptSmac]]

 Copyright © TCG 2006-2020

10.2.19 CryptSym.c

[[CryptSym]]

10.2.20 PrimeData.c

[[PrimeData]]

10.2.21 RsaKeyCache.c

[[RsaKeyCache]]

10.2.22 Ticket.c

[[Ticket]]

10.2.23 TpmAsn1.c

[[TpmAsn1]]

10.2.24 X509_ECC.c

[[X509_ECC]]

10.2.25 X509_RSA.c

[[X509_RSA]]

10.2.26 X509_spt.c

[[X509_spt]]

10.2.27 AC_spt.c

[[AC_spt]]

Annex A
(informative)
Implementation Dependent

A.1 Introduction

This header file contains definitions that are used to define a TPM profile. The values are chosen by the manufacturer. The values here are chosen to represent a full featured TPM so that all of the TPM's capabilities can be simulated and tested. This file would change based on the implementation.

The file listed below was generated by an automated tool using three documents as inputs. They are:

1) The TCG_Algorithm Registery,

2) Part 2 of this specification, and

3) A purpose-built document that contains vendor-specific information in tables.

All of the values in this file have #ifdef 'guards' so that they may be defined in a command line.Additionally, TpmBuildSwitches.h allows an additional file to be specified in the compiler command line and preset any of these values.

A.2 TpmProfile.h

[[TpmProfile_h]]

A.3 TpmSizeChecks.c

[[TpmSizeChecks]]

 Copyright © TCG 2006-2020

Annex B
(informative)
Library-Specific

B.1 Introduction

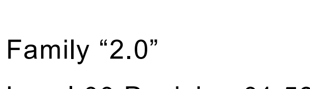
This clause contains the files that are specific to a cryptographic library used by the TPM code.

Three categories are defined for cryptographic functions:

1) big number math (asymmetric cryptography),
2) symmetric ciphers, and
3) hash functions.

The code is structured to make it possible to use different libraries for different categories. For example, one might choose to use OpenSSL for its math library, but use a different library for hashing and symmetric cryptography. Since OpenSSL supports all three categories, it might be more typical to combine libraries of specific functions; that is, one library might only contain block ciphers while another supports big number math.

DRAFT



 Copyright © TCG 2006-2020

B.2 OpenSSL-Specific Files

B.2.1. Introduction

The following files are specific to a port that uses the OpenSSL library for cryptographic functions.

B.2.2. Header Files

B.2.2.1. TpmToOsslHash.h

[[TpmToOsslHash_h]]

 Copyright © TCG 2006-2020

B.2.2.2. TpmToOsslMath.h

[[TpmToOsslMath_h]]

Copyright © TCG 2006-2020

B.2.2.3. TpmToOsslSym.h

[[TpmToOsslSym_h]]

B.2.3. Source Files

B.2.3.1. TpmToOsslDesSupport.c

[[TpmToOsslDesSupport]]

B.2.3.2. TpmToOsslMath.c

[[TpmToOsslMath]]

B.2.3.3. TpmToOsslSupport.c

[[TpmToOsslSupport]]

 Copyright © TCG 2006-2020

Annex C
(informative)
Simulation Environment

C.1 Introduction

These files are used to simulate some of the implementation-dependent hardware of a TPM. These files are provided to allow creation of a simulation environment for the TPM. These files are not expected to be part of a hardware TPM implementation.

C.2 Cancel.c

[[Cancel]]

Copyright © TCG 2006-2020

C.3 Clock.c

[[Clock]]

C.4 Entropy.c

[[Entropy]]

C.5 LocalityPlat.c

[[LocalityPlat]]

C.6 NVMem.c

[[NVMem]]

C.7 PowerPlat.c

[[PowerPlat]]

C.8 PlatformData.h

[[PlatformData_h]]

 Copyright © TCG 2006-2020

C.9 PlatformData.c

[[PlatformData]]

 Copyright © TCG 2006-2020

C.10 PPPlat.c

[[PPPlat]]

C.11 RunCommand.c

[[RunCommand]]

C.12 Unique.c

[[Unique]]

C.13 DebugHelpers.c

[[DebugHelpers]]

 Copyright © TCG 2006-2020

C.14 Platform.h

[[Platform_h]]

C.15 PlatformACT.h

[[PlatformACT_h]]

 Copyright © TCG 2006-2020

C.16 PlatformACT.c

[[PlatformACT]]

C.17 PlatformClock.h

[[PlatformClock_h]]

Annex D
(informative)
Remote Procedure Interface

D.1 Introduction

These files provide an RPC interface for a TPM simulation.

The simulation uses two ports: a command port and a hardware simulation port. Only TPM commands defined in TPM 2.0 Part 3 are sent to the TPM on the command port. The hardware simulation port is used to simulate hardware events such as power on/off and locality; and indications such as _TPM_HashStart.

 Copyright © TCG 2006-2020

D.2 TpmTcpProtocol.h

[[TpmTcpProtocol_h]]

D.3 TcpServer.c

[[TcpServer]]

 Copyright © TCG 2006-2020

D.4 TPMCmdp.c

[[TPMCmdp]]

D.5 TPMCmds.c

[[TPMCmds]]