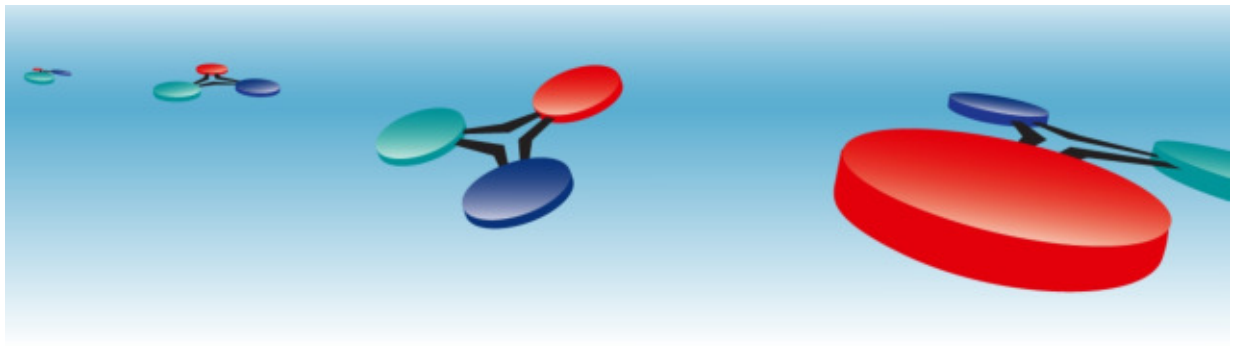

BACstac/Win v7.6

User's Guide

Document Version: 1.0

Cimetrics, Inc.

BOSTON, MASSACHUSETTS
USA
TELEPHONE: +1 (617) 350-7550
FAX: +1 (617) 350-7552
E-MAIL: products@cimetrics.com (sales), support@cimetrics.com (tech. support)
WEB: http://www.cimetrics.com

cimetrics™

BACstac/Win v7.6User's Guide

by Cimetrics, Inc.

Document Version: 1.0 Edition

Table of Contents

Chapter 1. Introduction

Thank you for your interest in Cimetrics' BACstac software. The BACstac Software provides you with the ability to create client and server application programs which use the protocol for communication. Certain versions of the BACstac Software also allow you to create gateway application programs that make a collection of non-BACnet devices appear as a virtual BACnet network containing multiple BACnet devices.

This document describes how to construct applications using the Cimetrics' BACstac software. It also gives a description of BACstac data types and the basics of using them in applications.

For a description of how to install the BACstac software, please refer to the BACstac *Installation Guide*. For a more detailed description of the data types and API routines available in the BACstac software, please refer to the BACstac *Programmers' Reference*.

About BACnet

BACnet is an ISO, ANSI, and ASHRAE standard network protocol for building automation. BACnet was developed by the American Society of Heating, Refrigerating, and Air-Conditioning Engineers (ASHRAE), and it was originally approved as an ASHRAE standard in 1995. The BACnet standard is maintained by ASHRAE committee SSPC 135.

BACnet was originally developed in order to improve interoperability between building automation equipment, but its data definitions and flexible architecture make it usable in a wide range of distributed control applications.

BACnet is an object-oriented protocol. BACnet *Objects* are used to represent network-accessible control equipment. BACnet defines several standard object types, including the following: Accumulator, Analog Input/Output/Value, Averaging, Binary Input/Output/Value, Calendar, Command, Device, File, Life Safety Point, Life Safety Zone, Loop, Multi-state Input/Output/Value, Notification Class, Event Notification, Program, Pulse Converter, Schedule and Trend-Log. Each Object contains a list of Properties, whose values indicate the current state of the control equipment. The BACnet standard specifies mandatory and optional Properties for the standard Objects, as well as mandatory behavior associated with some Properties.

A BACnet *Server* maintains a collection of Objects, including one Device Object which describes the BACnet capabilities of the Server. A Server will respond to BACnet service requests to read and manipulate its Objects. A BACnet *Client* generates service requests to monitor and control BACnet Servers. Some service requests that Servers will commonly respond to are Who-Is, Read-Property and Write-Property. Servers can also generate service requests, such as I-Am and the service requests used to report events and alarms. Many BACnet controllers and workstations have both Client and Server functionality.

For transmission of packets, BACnet makes use of three popular LAN technologies: Ethernet™, ARCNET™, and LONtalk™. BACnet devices may also be directly connected to IP networks. BACnet includes an RS-485 master/slave data link layer and a point-to-point data link layer that were specially developed for BACnet. A BACnet internetwork can be formed from several BACnet LANs employing similar or different LAN technologies using BACnet-compliant routers.

Any user of the BACstac software will need to learn more about BACnet. The main references for BACnet are:

- *ANSI/ASHRAE Standard 135-2016* or later "BACnet" (ISSN 1041-2336). The standard is available in book form or as an electronic document from ASHRAE (http://www.ashrae.org/).
- *Direct Digital Control of Building Systems*, by H. Michael Newman (ISBN 0-471-51696-1). This book, which was published in 1994, contains one chapter on data communications and one chapter describing BACnet prior to its approval as a standard.

- *BACnet Gebäude-Automation 1.4*, by Hans R. Kranz (ISBN 3-922420-02-8). This German-language book was published in 2005.
- Various articles about BACnet have been published in the ASHRAE Journal and elsewhere. The BACnet bibliography (http://www.bacnet.org/Bibliography/) contains references for many of these articles.

Technical Support

You may contact us by any of the following means (e-mail is preferred):

Telephone: +1 617-350-7550 (9 a.m. to 5 p.m. EST)

Fax: +1 617-350-7552

E-mail: support@cimetrics.com (mailto:support@cimetrics.com)

Chapter 2. BACstac Data Types

The BACnet protocol represents control equipment as a collection of Devices containing Objects. The Objects contain Properties that describe the current state of the control equipment. The BACnet protocol defines various services that the Devices can be asked to perform, including inspecting and changing the values of these Properties.

The binary encoding for BACnet service requests is described using Abstract Syntax Notation One (ASN.1) in Clause 21 of the BACnet standard. The BACstac software hides many of the details of this binary encoding, and allows applications to manipulate BACnet service parameters and Properties by defining corresponding C data types. The BACstac data types correspond closely to the BACnet application and base data types defined in Clause 21.

The BACstac data types are defined in the header file `BACTYPES.H`. This file is included automatically by each BACstac application header file (`BACCLI.H`, `BACSRV.H` or `BACGTW.H`). Each of the BACstac data types is described in the BACstac *Programmers' Reference*. The BACstac data types are available in BACstac Client, Server and Gateway applications. This chapter describes some of the most common data types.

Access Routines

The BACstac software provides access routines for manipulating the BACstac data types. These access routines are intended to improve the readability, and maintainability of BACstac applications by hiding the implementation details of the BACstac data types. Using the access routines is recommended to protect your BACstac applications from future changes to the implementation of the data types. To use the access routines, your application must also link to BACACC.LIB.

The BACstac can store the images of all the devices and objects being in work, making manipulating with them easy but determined enough and somehow limited. In this case some of the BACstac data types are "hidden" inside the BACstac API. For these data types, (such as Devices, and Objects) the actual data is maintained by the BACstac Object Database. Then the BACstac applications use handles to refer to the instances of these data types. Access routines must be used to manipulate data types represented by handles. Later in this document the alternative way of programming will be introduced (see the Section called *Low-Level API Programming Tools* in Chapter 8 of this Guide). Thus far while we describe "handle-based" methods we mean using the BACstac Object Database.

The names of access routines have the `BACstac` prefix, and are lower case with the first letter of words capitalized, like the BACstac API routines that generate service requests. The names of access routines are constructed using the patterns:

```
BACstacVerbName
BACstacVerbNameAttribute
```

The BACstac Access Routines created for all purposes are described in detail in the BACstac *Programmers' Reference*.

Access routines typically take a pointer to an instance of the data type as their first argument. For a data type containing multiple attributes, the access routines include methods for reading and writing each of the attributes. If the attribute is in turn represented by a simple data type, the attribute can be passed directly as an argument (of a BACstacSet-NameAttr() routine) or as a return value (of a BACstacGetNameAttr() routine). If the attribute is a complex data type, a pointer to the attribute is passed (to a BACstacCopyNameAttr() routine) or returned (by a BACstacGetNameAttrPtr() routine).

For example, the following code fragment manipulates a Real value that is contained in a Priority Array Item data type that is contained in a Property Contents data type. The resulting Priority Array Item is then copied into another Property Contents data type. The Real value is manipulated directly with Get/Set access routines, while the more complex Priority Array Item is manipulated with GetPtr/Copy access routines.

```
BACSTAC_REAL X;
BACSTAC_PRI_ARRAY_ITEM *pPriItem;
BACSTAC_PROPERTY_CONTENTS ContentsWithOldItem; /* initialized elsewhere */
BACSTAC_PROPERTY_CONTENTS ContentsWithNewItem; /* initialized elsewhere,*/
                                               /* but we will overwrite it*/
pPriItem = BACstacGetContentsPriItemPtr(&ContentsWithOldItem);
BACstacCopyContentsPriItem(&ContentsWithNewItem,pPriItem);
X = BACstacGetPriArrayItemReal(pPriItem);
BACstacSetPriArrayItemReal(pPriItem,X*0.95);
```

The BACstac software also provides access routines to initialize instances of a data type (typically named BACstacInitName()). An initialization routine must be provided with values for some of attributes of the data type. Other attributes will be set to default values by the initialization routines. For example, an Object Identifier variable can be initialized by providing an Object type and Object instance number:

```
BACSTAC_INST_NUMBER InstNum = 1967;
BACSTAC_OBJECT_TYPE ObjType = OBJ_ANALOG_INPUT;
BACSTAC_OBJECT_ID ObjID;
BACstacInitObjectID(&ObjID,ObjType,InstNum);
```

An initialization routine usually provides the same result as using a series of Set/Copy access routines on an instance of a data type. However, by using an initialization routine, an application programmer is relieved of the burden of remembering which attributes are required, which are optional and which can take default values.

Address

A BACnet Address is a destination for a BACnet service request or reply. It describes the network address of a BACnet Device or Client, or it describes a BACnet network.

A BACnet Address can contain a MAC address (1 byte for ARCNET, 6 bytes for Ethernet) for a particular Device or it can indicate that a broadcast is desired. A BACnet Address can be defined on the local BACnet network or a remote BACnet network, in which case it includes the BACnet network number of the remote BACnet network. A BACnet Address can also indicate that a global broadcast on the entire BACnet inter-network is desired.

The BACnet protocol requires that only one BACnet Device appear at an address (known as a *MAC address*) on a local area network (LAN). BACnet routers can be used to connect multiple BACnet LANs, in which case each LAN must be assigned a unique BACnet *network number*. This means that each Device on a BACnet inter-network can be identified by a BACnet network number and a MAC address. In case of a local Device the network number is set to 0.

Some BACnet messages can be sent to every computer on a BACnet network (known as a local or remote broadcast) or broadcast over the entire BACnet inter-network (known as a global broadcast). Broadcasts are used by BACnet systems to discover or publicize the existence of BACnet Devices using the Who-Is, I-Am, Who-Has, and I-Have services.

Client applications can have a BACnet Address without acting like a BACnet Device and maintaining Objects. That is, they can receive replies and broadcasts but they do not respond to most BACnet requests. Such a Client will listen for all broadcasts on the local BACnet network. BACnet Devices will be able to direct a reply to such a Client, because the Client's Address is included in any requests that it sends to the Devices.

For example, a Client can send a Who-Is query to all the Devices on the entire BACnet inter-network by setting up an instance of the BACstac Address data type to indicate a global broadcast as the destination for the service request.

```
BACSTAC_ADDRESS Address;
```

```
/* Set up the Address to indicate all Devices
   on the BACnet inter-network */
BACstacInitAddrGlobalBroadcast(&Address);
/* Use the Address to send the Who-Is query */
BACstacWhoIs(&Address,BACSTAC_NO_FILTER,BACSTAC_NO_FILTER);
...
```

A BACstac Client sends some service requests to Devices by using handles to refer to the Devices. Other service requests are sent directly to Objects within Devices by using handles to refer to the Objects. The BACstac software keeps track of the appropriate Address to use when sending service requests. When the Client obtains Object and Device handles, the Device Address can be supplied explicitly to the BACstac by the Client application or the Device Address can be discovered by using a Who-Is or Who-Has query.

A Client can use an Address data type to obtain a handle for the Device. The following Client example uses access routines to construct an Address for a known Device, which can then be used to obtain the desired Device and Object handles.

```
BACSTAC_ADDRESS Address;
BACSTAC_BYTE *pMAC;
/* Set up the Address to indicate a particular Device
   on the local ARCNET network */
pMAC[0] = 0xAA;  /* set the MAC bytes - Ethernet would have 6 */
BACstacInitAddrLocal(&Address,pMAC,BACSTAC_MAC_ARCNET_LENGTH);
/* Now we can obtain the handles
   see the examples in the Object section */
```

The BACstac software allows Server and Gateway applications to override the default service handling behavior by using application-supplied subroutines called hooks (discussed in Chapter 7 of this manual). A hook that is handling a service request will be given the Address of the source of the service request as one of its parameters. The Server application could discover information about the source of the request from this Address. In the following example the Server extracts the BACnet network number of the request source using a Get access routine:

```
MyHook(...,BACSTAC_ADDRESS *pSourceAddress, ...)
{
    BACSTAC_UI16 NetNumber;
    ...
    if(BACstacIsAddrRemote(pSourceAddress))
    {
    NetNumber = BACstacGetAddrNetworkNumber(pSourceAddress);
    }
    else
    {
    /* The Network number for local devices must be 0 */
    NetNumber = 0;
    }
    ...
}
```

Device

A BACnet *Device* is control equipment connected to a BACnet network; the Device is capable of responding to BACnet service requests. A Device represents the control equipment by maintaining a collection of BACnet Objects.

The Objects contain Properties that can be manipulated using the BACnet Read-Property and Write-Property service requests. The Objects within any Device are uniquely identified by an Object *type* and by an instance number (together these data make up an Object Identifier).

Each Device must contain one *Device Object* to represent the BACnet capabilities of the Device. Each Device on a BACnet inter-network must have a unique Device Object instance number and, consequently, a unique Device Object identifier. Devices can be contacted directly if their BACnet Address is known, or they can be discovered using the Who-Is and Who-Has broadcasts.

A Device is represented in BACstac applications based on BACstac Object Database by a Device handle. The BACstac maintains internally the Device information such as the BACnet Address of the Device, or the list of Device's Objects. The BACstac uses this information when it generates service requests or responds to.

To extend the example in the previous section, consider a Client that can retrieve a list of Device handles for Devices that responded to a Who-Is request. These Device handles could be used to discover the Addresses of the Devices or to access the Objects in the Device. The following example discovers the Device Object instance numbers of all the responding Devices on the BACnet inter-network:

```
BACSTAC_HDEVICE hDevices[NDEVMAX];
BACSTAC_DEVICE_COUNT nDevCount,nDev;
BACSTAC_INST_NUMBER InstNum;
BACSTAC_ADDRESS Address;
/* Set up the Address to indicate all Devices
on the BACnet inter-network */
BACstacInitAddrGlobalBroadcast(&Address);
/* Use the Address to send the Who-Is query */
BACstacWhoIs(&Address,BACSTAC_NO_FILTER,BACSTAC_NO_FILTER);
OSLSleep(2*1000); /* delay for 2 seconds to allow devices to respond */
BACstacGetDeviceList(hDevices,NDEVMAX,&nDevCount);
for(nDev=0;nDev<nDevCount;nDev++)
{
    BACstacGetDeviceInstNum(hDevices[nDev],&InstNum);
    ...

}
```

When a Device's Address and Device Object instance number are known, a BACstac Client can generate a new Device handle using BACstacCreateDeviceImage():

```
BACSTAC_HDEVICE hDev;
BACSTAC_ADDRESS Address;
/* set up Address, as in the previous "Address" section */
/* Now create a handle for the Device */
/* its Device Object instance number is known to be 1967 */
BACstacCreateDeviceImage(&Address,1967,&hDev);
```

A BACstac Client application can destroy a Device handle by using the BACstacDeleteDeviceImage() routine. A BACstac Client application can look up an existing Device handle using its unique Device Object instance number or Address:

```
hDev = BACstacDeviceAddressToHandle(&Address);
hDev = BACstacDeviceInstNumToHandle(InstNum);
```

BACstac Server and Gateway applications can obtain the Device handles of their local Devices. A Server would need to use its local Device handle to obtain Object handles for the local Objects. The Device handle of a local Device can

also be used to obtain other information about the Device, such as its Address. For example a Server could discover its MAC address by using the local Device handle:

```
BACSTAC_HDEVICE hDev;
BACSTAC_ADDRESS Address;
BACSTAC_BYTE *pMAC;
hDev = BACstacGetLocalDeviceHandle();
BACstacGetDeviceAddress(hDev,&Address);
pMAC = BACstacGetAddrMACPtr(&Address);
...
```

A BACstac Server maintains only a single Device, which is created when the application invokes the BACstacServerInit() routine. The local Device in a Server cannot be deleted. A Gateway maintains a list of local Devices. The Gateway acts like a BACnet router and makes each local Device appear to exist at a unique MAC Address on a virtual BACnet network. The following example illustrates how a Gateway could review these Addresses:

```
BACSTAC_HDEVICE hDevices[NDEVMAX];
BACSTAC_DEVICE_COUNT nDevCount, nDev;
BACSTAC_ADDRESS Address;
BACSTAC_BYTE *pMAC;
BACstacGetDeviceList(hDevices,NDEVMAX,&nDevCount);
for(nDev=0;nDev<nDevCount;nDev++)
{
    BACstacGetDeviceAddress(hDevices[nDev],&Address);
    pMAC = BACstacGetAddrMACPtr(&Address);
    ...
}
```

BACstac Gateways can create and delete local Devices by using BACstacCreateDeviceFromTemplate() and BACstacDeleteDevice(). These routines are described in Chapter 5.

Object

A BACnet Object makes visible one specific control function of a Device to other computers on a BACnet network. An Object contains a list of Properties that describe its functionality. The BACnet standard defines mandatory Properties and behaviors for standard types of Objects such as Analog Input, Analog Output, Schedule etc. Clause 12 of BACnet standard, “MODELING CONTROL DEVICES AS A COLLECTION OF OBJECTS”, contains descriptions of the Properties required to be present in standard Objects.

An Object in a BACnet Device is uniquely identified within the current Device by an Object Identifier, which consist of an Object *type* and instance number. A Device must contain one Device Object, which has an instance number that is unique within the BACnet inter-network. All other objects in the Device have not to be unique within the network.

Objects are represented in BACstac applications by an Object handle. The BACstac maintains information internally about an Object, such as the list of Properties that it contains. An Object’s handle can be obtained by searching through the list of Objects contained in a Device. An Object’s handle can also be obtained if its Object Identifier (type and instance number) is known.

A BACstac application can obtain a list of Object Identifiers for all the Objects in a Device by using the BACstacReadObjectList() routine (in case BACstac Object Database is used). Direct reading of the Object List property of the Device through the use of BACstacReadProperty2() is also possible. In a Client application this routine sends a Read-Property request to the Device Object to retrieve its object-list Property. This information is saved in the local

Device image for use in subsequent calls. Servers and Gateways can retrieve the list of Objects in a local Device without generating a service request.

To obtain the number of Objects in the Device first, you may call BACstacReadObjectList() twice:

```
BACSTAC_HDEVICE hDev; /* initialized elsewhere */
BACSTAC_HOBJECT hObjs[MAX_OBJ_NUM];
BACSTAC_OBJECT_ID ObjectIDs[MAX_OBJ_NUM];
BACSTAC_OBJECT_COUNT nObjs;
BACSTAC_ERROR err;
/* supply zero arguments to obtain the number of Objects */
BACstacReadObjectList(hDev, NULL, 0, &nObjs, &err);
/* Read the list of nObjs elements */
BACstacReadObjectList(hDev, &objectIDs, nObjs, &nObjs, &err);
```

After having obtained the Object Identifier, a BACstac application can further obtain all existing Object handles by using the BACstacFindObjectByID() routine in a loop:

```
int i;
/* Obtain handles for all Objects in the list */
for(i = 0; i < nObjs; i++)
    hObjs[i] = BACstacFindObjectByID(hDev,ObjectIDs[i]);
```

A BACstac application can also obtain the Object handle for the Device Object directly, since each Device must have one and only one Device Object (A Device handle is not the same as the Object handle for the Device Object. BACstac API routines that take Object handles for arguments will not accept a Device handle).

```
BACSTAC_HDEVICE hDev; /* initialized elsewhere */
BACSTAC_HOBJECT hDevObj;
hDevObj = BACstacGetDeviceObjectHandle(hDev);
```

A BACstac Client can obtain a new handle for an Object with BACstacCreateObjectImage(). In a BACstac Server or Gateway application, the list of local Objects in a Device can be changed using BACstacCreateDBObject() and BACstacDestroyObject(). These routines are described in Chapter 4 of this Guide.

Property

A Property represents one attribute of a BACnet Object. A Property can be a number or a character string or an enumeration or a complicated data type. The values of Properties in Objects in BACnet Devices can be manipulated using the BACnet Read-Property and Write-Property service requests.

A Property is uniquely identified within the current Object by a Property Identifier. An individual Property is identified in a BACstac application by the combination of a Property Identifier and an Object handle. Clause 12 of the BACnet standard defines the required and optional Properties for standard BACnet Objects.

BACstac applications represent the value of a Property by the Property Contents data type described in the next section. A Property Contents is used when an application generates service requests and when a Server or Gateway application manipulates local Property values.

A BACstac Server or Gateway application can control the way the BACstac default actions treat a Property when processing a BACnet request. These options include whether to invoke a callback when accessing the Property and the desired read/write permission for the Property. A BACstac Server or Gateway application also has direct access to the current value of a Property in the BACstac Object Database.

The read/write permissions are manipulated using BACstacSetPropertyAccess() and GetPropertyAccess() to specify that a Property has BACSTAC_ACCESS_READ_ONLY or BACSTAC_ACCESS_READ_WRITE. The BACstac default actions use these permissions when servicing Read-Property and Write-Property requests.

The BACstac default actions use application-supplied callbacks to access physical hardware. When a callback has been registered for a service, a Server can manipulate a Boolean flag that indicates whether an individual Property is to use the callback by using the routines BACstacSetCallbackAttachment() and BACstacGetCallbackAttachment().

The value of a local Property can be accessed by using BACstacRetrievePropertyInstance2() and BACstacStorePropertyInstance(). These routines use Property Contents data structures as arguments.

A Server or Gateway application can also provide a list of Property Instances containing this information when it uses BACstacCreateDBObject().

Property Contents

Properties can contain any of the BACnet Application Types and most of the BACnet Base Types. The ASN.1 encoding of these types is described in Clause 21 of the BACnet Standard.

The BACnet Standard also defines which of the types are permitted for particular Properties.

In BACstac applications, the value of a Property is represented by the Property Contents data type, which maintains a data type tag and a buffer to store the actual data. The Property Contents data type can contain either a single item or an array of the desired data type.

The Property Contents data type is used as an argument for routines that must be able to accept or return any of the data types that can occur as a value of a Property. The tasks of providing storage for the value and providing the value are split between the *calling* routine and *called* routine.

The calling routine must always provide the storage space for the value, either a variable of the appropriate type or a buffer with enough space to contain the value. A calling routine can initialize a Property Contents by attaching it to either 1) a variable of the appropriate type, 2) an array of the appropriate type, or 3) attaching to a buffer when the type isn't known. An application can use the BACstacInitContentsToXXX() access routines to accomplish this.

The called routine should exit indicating an error if there isn't enough space in the Property Contents in which to return a value.

A calling routine passing in a value or a called routine returning a value must ensure that the data type tag matches the data. The data access routines for the Property Contents help maintain this consistency. The supported data types are included in the BACSTAC_DATA_TYPE enumeration.

A Property Contents data is used as an argument to the API routines that generate Read-Property and Write-Property service requests.

In Server and Gateway applications, a Property Contents is used by hooks and callbacks when servicing Read-Property and Write-Property requests. Server and Gateway applications also use a Property Contents data type to access the local Property values using the BACstacStorePropertyInstance() and BACstacRetrievePropertyInstance2() routines.

There are access routines for each of the data types that can be represented as Property Contents. Classes of these data types are described in the following sections. There are also in-place access routines for initializing the data types inside a Property Contents.

The following examples illustrate some of the typical uses of the Property Contents data type.

Example 1: Write a real value to a property in a remote device using the Write-Property service request.

```
BACSTAC_REAL X = 3.14159;
```

```
BACSTAC_PROPERTY_CONTENTS pc;
BACSTAC_HOBJECT hAnalogOutputObj; /* initialized elsewhere */
/* set up the Property Contents to use the variable X */
BACstacInitContentsToVar(&pc,DATA_TYPE_REAL,&X,sizeof(X));
/* write the value using an appropriate priority (8) */
BACstacWriteProperty(hAnalogOutputObj, PROP_PRESENT_VALUE,
    BACSTAC_VOID_INDEX, BACSTAC_PRI_MANUAL_OPERATOR, &pc, NULL);
```

Example 2: Retrieve a real value from a local property.

```
BACSTAC_REAL X;
BACSTAC_PROPERTY_CONTENTS pc;
BACSTAC_HOBJECT hAnalogOutputObj; /* initialized elsewhere */
/* set up the Property Contents to use the variable X */
BACstacInitContentsToVar(&pc, DATA_TYPE_REAL, &X, sizeof (X));
/* retrieve the value from a local Object */
BACstacRetrievePropertyInstance2(hAnalogOutputObj, PROP_MAX_PRES_VALUE,
    BACSTAC_VOID_INDEX, &pc, NULL);
printf("X = %e\n",X);
```

Example 3 : Return a Real in callback

```
MyReadCallback(...,BACSTAC_PROPERTY_CONTENTS *pValue)
{
    BACSTAC_REAL X;
    /* assume that we know which Object and Property is being read */
    /* get the value from the hardware */
    X = ReadMyDevice();
    /* stuff the value into the Property Contents */
    BACstacSetContentsReal(pValue,X);
    return BACSTAC_CALLBACK_OK;
}
```

Example 4: Read a property using a buffer. We can discover the value's type and actual size later.

```
BACSTAC_HOBJECT hObj; /* initialized elsewhere */
BACSTAC_PROPERTY_ID PropID; /* initialized elsewhere */
BACSTAC_DATA_TYPE Type;
BACSTAC_VALUE_SIZE Size;
BACSTAC_PROPERTY_CONTENTS Value;
BACSTAC_UI32 mybuffer[BUFSIZE/sizeof(BACSTAC_UI32)];
/* initialize the Property Contents - don't know the type yet */
BACstacInitContentsToBuf(&Value,mybuffer,sizeof(mybuffer));
/* read the property - we should also check for error codes here */
BACstacReadProperty2(hObj, PropID, BACSTAC_VOID_INDEX, &Value, NULL, NULL);
/* we can now inspect the property value at our leisure */
Type = BACstacGetContentsType(&Value);
Size = BACstacGetContentsSize(&Value);
if(Type==DATA_TYPE_REAL)
{
    printf("Prop = %e\n",BACstacGetContentsReal(&Value));
}
```

Simple Data Types

The simple BACnet data types such as Real, Integer, Unsigned, Character String map easily onto C data types. The examples in the preceding section illustrated the use of the Property Contents data type using a Real variable. The numeric simple data types are represented by the C data types double and long, and can be manipulated directly using the Get/Set access routines.

A BACnet Character String (using X3.4 encoding) is represented as a C string (a '\0' terminated array of char). Since the size of a string can vary, calling routines reading string values must provide an adequate buffer. Pointers are used to manipulate strings, using the GetPtr/Copy access routines. Since the standard C library provides routines for manipulating character strings, the BACstac software only provides a handful of access routines for using strings that occur inside a Property Contents. More complicated data types have access routines for manipulating them when they occur outside of a Property Contents ("bare" instances of the data types).

For example, a called routine could copy a string value into a Property Contents to provide a return value. A called routine could also discover the pointer to a string inside a Property Contents to read it or to manipulate it in place.

```
void ReadStringValue(BACSTAC_PROPERTY_CONTENTS *pValue)
{
    BACSTAC_CHAR *pStr;
    pStr = ReadFromDeviceString();
    BACstacCopyContentsString(pValue,pStr);
    return;
}
void WriteStringValue(BACSTAC_PROPERTY_CONTENTS *pValue)
{
    BACSTAC_CHAR *pStr;
    if(DATA_TYPE_UTF_8_STRING==BACstacGetContentsType(pValue))
    {
        pStr = BACstacGetContentsStringPtr(pValue);
        WriteToDeviceString(pStr);
    }
    return;
}
```

Enumerations

Many BACnet Properties are enumerations, i.e. data types that can take on a small number of known symbolic values. These are each represented in BACstac applications by a C `enum`. A Property Contents containing enumeration has type equal to DATA_TYPE_ENUM.

In general, an application needs to know an applicable Property Identifier and Object type to know the meaning of an enumerated value. This information is not contained in a Property Contents data type, but is normally passed as other arguments to routines that manipulate Property Contents. The `BNETDEF.H` header file defines the symbolic values for BACnet enumerations.

For example, the present-value Property of a Binary Input Object is the data type BACSTAC_BINARY_PV, which can take the values BACSTAC_BINARY_INACTIVE or BACSTAC_BINARY_ACTIVE. A BACstac Client workstation could read the value and update its display:

```
BACSTAC_HOBJECT hBinInput; /* initialized elsewhere */
BACSTAC_PROPERTY_CONTENTS Value;
BACSTAC_BINARY_PV MyBinInput;
```

```
/* attach the variable to the Property Contents */
BACstacInitContentsToVar(
    &Value,
    DATA_TYPE_ENUM,
    &MyBinInput,
    sizeof(MyBinInput));
/* read the present value property of a Binary Input Object*/
BACstacReadProperty2(
    hBinInput,
    PROP_PRESENT_VALUE,
    BACSTAC_VOID_INDEX,
    &Value,
    NULL,
    NULL);
/* update the console display */
if(MyBinInput==BACSTAC_BINARY_ACTIVE)
{
    SetLightOnScreen(GREEN);
}
else
{
    SetLightOnScreen(RED);
}
```

Bit Strings

Bit strings are used in BACnet Objects for Properties that answer collections of Yes/No questions. Most common Properties represented by bit strings are status-flags in Input/Output Objects and protocol-object-types-supported and protocol-services-supported in Device Objects.

Bit strings are represented in BACstac applications as packed byte arrays. The BACSTAC_BIT_STRING data type uses a fixed size buffer into which an application can place a variable length bit string. The maximum length bit string that can be represented in a BACstac application is (8 bits)*BACSTAC_MAX_BIT_STRING_BUF = 80 bits. Access routines are provided for initializing a bit string data type (with zeroed bits, 0==FALSE) and for setting/reading a particular bit.

The `BNETDEF.H` file defines symbolic values for the bits in the standard BACnet bit strings. These values are indices, not masks, to be used with the bit string access routines.

For example, during startup a Server could construct a bit string to dynamically initialize the protocol-services-supported Property of its Device Object:

```
#define N_SVC_BITS 35
BACSTAC_BIT_STRING MyServices;
BACSTAC_HOBJECT hDevObj; /* initialized elsewhere */
BACSTAC_PROPERTY_CONTENTS Value;
/* Describe some of the Server's capabilities */
BACstacInitBitString(&MyServices,N_SVC_BITS); /* zeros all bits */
BACstacSetBitStringBit(&MyServices,SVC_READ_PROP, BACSTAC_TRUE);
BACstacSetBitStringBit(&MyServices,SVC_WRITE_PROP, BACSTAC_TRUE);
BACstacSetBitStringBit(&MyServices,SVC_I_AM, BACSTAC_TRUE);
BACstacSetBitStringBit(&MyServices,SVC_WHO_IS, BACSTAC_TRUE);
LookUpOtherCapabilities(&MyServices);
```

```
if(BACstacTestBitStringBit(&MyServices,SVC_ADD_LIST_ELEMENT)){
GetReadyForLists(); /* have to do some more work */
    }
/* now set the value of the protocol-services-supported Property */
BACstacInitContentsToVar(
&Value,
DATA_TYPE_BIT_STRING,
&MyServices,
sizeof(MyServices));
BACstacStorePropertyInstance(
hDevObj,
PROP_PROT_SERVICES_SUPPORTED,
BACSTAC_VOID_INDEX,
&Value);
```

Sequences

Many of the more complicated BACnet data types are sequences of simpler data types (indicated in the ASN.1 description by the keyword SEQUENCE). Such data types are represented in BACstac applications by C data structures containing multiple fields. Access routines are provided to Set/Get or Copy/GetPtr the contents of each field. Initialization access routines are provided to set initial values for the fields.

The fields of a Date data type are the year, month, day-of-month and day-of-the-week. Set and Get routines can be used to access the fields. Data types (such as BACSTAC_MONTH) are defined for many fields to hide implementation details.

```
BACSTAC_DATE Date; /* initialized elsewhere */
BACSTAC_MONTH WorkMonth;
WorkMonth = BACstacGetDateMonth(&Date);
if(WorkMonth==MONTH_AUGUST)
{
    /* everyone is on vacation in August */
    WorkMonth = MONTH_SEPTEMBER;
    BACstacSetDateMonth(&Date,WorkMonth);
}
```

All of the fields of the Date data type can be passed as parameters to the initialization routines, although the BACnet standard allows the fields to take on special values such as YEAR_UNSPECIFIED. For example, a caller routine could initialize a Date data type directly and attach it to Property Contents:

```
BACSTAC_DATE Date;
BACSTAC_PROPERTY_CONTENTS Value;
BACstacInitDate(&Date,1966,MONTH_OCTOBER,4);
BACstacInitContentsToVar(
    &Value,
    DATA_TYPE_DATE,
    &Date,
    sizeof(Date));
```

A called routine would use an in-buffer initialization routine to set up the Date, since it must ensure that the provided buffer has enough space and that the data type tag gets set to match the data:

```
BACSTAC_BOOLEAN GetOldDate(BACSTAC_PROPERTY_CONTENTS *pValue)
{
    BACSTAC_BOOLEAN OK;
    OK = BACstacInitInContentsDate(
        pValue,
        1966,
        MONTH_OCTOBER,
        4);
    if(!OK)
    {
        /* buffer could have been too small */
        /* perform some cleanup actions? */
    }
return OK;
}
```

Choices

Some BACnet data types (and some fields of BACnet SEQUENCE data types) are choices of simpler data types (indicated in the ASN.1 description by the keyword CHOICE). Such data types are represented in BACstac applications by C unions plus a tag field. The tag indicates which of the choices is represented by the current contents of the union. Access routines are provided for the tag field and for the sub-fields in each choice in the union. Initialization access routines are provided for each choice to set the initial values for the choice.

For example the Priority Array Item data type is used to manipulate the Priority-Array Property of commandable Objects. An application can initialize the variable to contain Null (a common value indicating that no value is present at a given priority level in the priority array of a commandable object), or set the variable to a Real or Unsigned or Enumerated or Boolean or Integer or Double or Time or Character String or Octet String or Bit String or Date or Object Identifier or Date Time value.

```
BACSTAC_PRI_ARRAY_ITEM PriItem;
/* The Priority Array Item will be set to Null */
BACstacInitPriArrayItem(&PriItem);
...  /* do stuff */
/* now we may want to change it to a Real value */
if(IsPriArrayItemNull(&PriItem))
{
    BACstacSetPriArrayItemReal(&PriItem,3.14159);
}
```

Arrays

Some BACnet Properties are defined as arrays of simpler data types (indicated by the ASN.1 keyword ARRAY). Arrays are represented in BACstac applications by fixed size elements, stored sequentially in a C array. BACstac applications can access an entire array or one element of an array by supplying an index to the BACstac API routines (Read/Write, Store/Retrieve). The index for BACnet arrays starts counting at 1. Supplying an index of 0 (corresponding to the special value BACSTAC_ARRAY_COUNT) to an API routine will return the number of elements in the

array. Supplying the special value BACSTAC_ENTIRE_ARRAY as the index will return the entire array. An application uses the special value BACSTAC_VOID_INDEX as an API routine argument to access Properties that aren't arrays.

When an entire array is being manipulated in a BACstac application, a Property Contents data type keeps a C array in its buffer, and keeps track of the size of the array. The number of elements in the array can be queried with the BACstacGetContentsElementsNum() access routine. A different initialization routine, BACstacInitContentsToArray(), must be used to attach an array variable to a Property Contents data type. Pointers to elements of the C array can be passed to access routines for the element data type. The index for C arrays starts counting at 0.

In the following example, a Priority Array is initialized by setting all the elements to Null; then a real value is written to one of priorities and the array is attached to a Property Contents data type.

```
BACSTAC_REAL ControlValue; /* initialized elsewhere */
BACSTAC_PRIORITY_ARRAY PriArray;
BACSTAC_PROPERTY_CONTENTS Value;
/* set all 16 elements of the array to NULL */
BACstacInitPriArray(&PriArray);
/* insert a control value at BACnet Priority 15 */
BACstacSetPriArrayItemReal(&(PriArray[14]),ControlValue);
/* attach the array to a Property Contents */
BACstacInitContentsToArray(
    &Value,
    DATA_TYPE_PRI_ARRAY_ITEM,
    &PriArray,
    sizeof(PriArray),
    BACSTAC_MAX_PRI_ARRAY);
/* now we can write the array to the priority-array Property */
```

Chapter 3. Creating a Client Application

You can create a BACnet Client application by using BACstac Client API routines to assemble information about remote Devices and Objects. In response to control logic or user input, your client application can read and write the Properties of remote Objects. The routines used in the examples below are documented in the *Client API* section of the BACstac *Programmers' Reference*. BACnet Clients do not maintain local Devices, or respond to most BACnet service requests. Workstations, test equipment and controllers could be implemented as BACstac Clients.

You can link the BACstac routines to your application by using the BACCLIx.LIB libraries. You must include the BACstac Client application header file in your source code files:

```
#include <
baccli.h>
```

Device and Object Images

A BACstac Client application keeps local images of the remote Devices and Objects that it is manipulating. Your client application uses handles to refer to these local images. When Device images are created, the Device Object is always created automatically.

Initialization of the BACstac Client

You must initialize the BACstac client before your application can send any BACnet network requests. You do this by calling the API routine BACstacClientInit(). Its arguments include size limits for the data structures describing the local images of Objects in the application. If you do not supply the maximum number of Devices and Objects for your application, default values will be set (32 for both in the current version). The maximum number of Device (and Object) Images that you can specify is 64k. Device Objects are created automatically by the BACstac, so you may provide a list of descriptions of properties to include in these images. If no description list is provided, the application will initialize using a default list of properties.

An example of use of BACstacClientInit() in an application is:

```
#include <
baccli.h>    /* define the client API */
#define N_DEV_PROP_MAX 20
#define N_DEVICES 10
#define N_OBJECTS 10
main()
{
    BACSTAC_CLI_INIT init_data;
    BACSTAC_PROPERTY_COUNT nDevPropDescBuf;
    BACstacInitCliInit(&init_data,7,6);       /* for BACstac 7.6 */
    BACstacSetCliInitMaxDevices(&init_data,N_DEVICES);
    BACstacSetCliInitMaxObjects(&init_data,N_OBJECTS);
    BACstacClientInit(&init_data,NULL);
    ...
}
```

The Version numbers (Major and Minor) supplied by your application are checked by the BACstac initialization routine to ensure that your application is compatible with the version of the BACstac library with which your application was built.

Finding Remote BACnet Devices

Your client application can discover remote Devices by broadcasting a BACnet Who-Is request. The BACstac client will accumulate replies until the application asks for the list of Devices that have responded. This list is returned to the application by the BACstacGetDeviceList() routine. A client application should wait for the I-Am replies for an interval that is appropriate to the local BACnet network architecture. An example of discovering remote devices with this technique is:

```
BACSTAC_ADDRESS Address;
BACSTAC_HDEVICE hDevBuf[N_DEVICES]
BACSTAC_DEVICE_COUNT nHDev;
BACstacInitAddrGlobalBroadcast(&Address);
BACstacWhoIs(&Address,BACSTAC_NO_FILTER,BACSTAC_NO_FILTER);
OSLSleep(2*1000); /* delay for 2 seconds to allow devices to respond */
BACstacGetDeviceList(hDevBuf,N_DEVICES,&nHDev);
```

The number of Device Handles returned by BACstacGetDeviceList() is limited by the size of the buffer that you provide and by the limit set at initialization. The filter arguments can be used to narrow the range of Devices which will respond to the Who-Is request. If you know the Address and Device Object Instance Number of a remote Device, you can also create a Handle for that Device using BACstacCreateDeviceImage(). BACstacGetDeviceList() may also be used to discover devices that have broadcast I-Am messages to the BACnet network. The I-Am messages can be broadcast automatically by Devices, or can be responses to a Who-Is request by your Client application.

Finding Objects

Your application can read the list of Objects in a Device using BACstacReadObjectList(). This is done by sending a Read-Property request for the object-list property of the Device Object. After the first use of the routine for a Device, the information is cached in the local Device image. An example of discovering the objects in a device is:

```
BACSTAC_HDEVICE hDevice; /* initialized elsewhere */
BACSTAC_OBJECT_ID ObjBuf[N_OBJECTS];
BACSTAC_OBJECT_COUNT nObjAvail;
/* read the list of Object IDs */
BACstacReadObjectList(hDevice,ObjBuf,N_OBJECTS,&nObjAvail,NULL);
```

Your client application can also use the BACstacReadProperty2() routine with array indices to inspect the Object-List contents one element at a time. This Property is contained in the Device Object, so you need to first obtain the Handle for the Device Object:

```
BACSTAC_HDEVICE hDevice;
BACSTAC_HOBJECT hDevObject;
hDevObject = BACstacGetDeviceObjectHandle(hDevice);
```

Creating Object Images

A client application to obtain a handle to a remote device should create its image in the local database. This image contains properties of a remote object. There is no need to store in this cache any properties except that are intended to be used in the read-once. Therefore, the default property list, which includes only property-identifier, is normally what you want.

The client application provides the Device Handle and Object ID and the routine returns a handle to the remote Object.

```
BACSTAC_HDEVICE hDevice; /* initialized elsewhere */
BACSTAC_OBJECT_ID ObjID;
BACSTAC_HOBJECT hAnalogIn1;
/* set up the Object ID */
BACstacInitObjectID(&ObjID,OBJ_ANALOG_INPUT,1);
/* create the local object image using the default set of Properties */
BACstacCreateObjectImage(hDevice,&ObjID,&hAnalogIn1);
```

If you want to re-create an Object Image completely, you should first destroy the old Object Image:

```
BACstacDeleteObjectImage(hAnalogIn1);
```

Reading Properties

The values of Properties are manipulated using a Property Contents data type. See the *Property Contents* section of the BACstac *Programmers' Reference* for a description of the access routines for manipulating Property Contents. The data type provides a mechanism for manipulating properties of varying types. The contents of a property value are stored in a variable or buffer that you provide. Your application must also provide information about the available buffer size.

Querying the value of a Property has two steps: setting up the Property Contents data type, and then reading the value into the space provided by the data type. An example of reading a property value is:

```
BACSTAC_HOBJECT hAnalogInput1; /* initialized elsewhere */
BACSTAC_PROPERTY_CONTENTS Value;
BACSTAC_REAL Temperature;
/* attach the Temperature variable to the Property Contents */
BACstacInitContentsToVar(
    &Value,
    DATA_TYPE_REAL,
    &Temperature,
    sizeof(Temperature));
/* Read the value into Temperature */
BACstacReadProperty2(
    hAnalogInput1,
    PROP_PRESENT_VALUE,
    BACSTAC_VOID_INDEX,
    &Value,
    NULL,
    NULL);
```

If a Property is an array, then you can specify an array index when making a Write-Property request to access one element of the array. The index value BACSTAC_ENTIRE_ARRAY can be used to access the entire array with one request. The index value BACSTAC_VOID_INDEX should be used for non-array Properties.

BACstacReadProperty2() will change the type and number of elements of the Property Contents argument to match the received data. Thus when the expected data type is not known, before calling BACstacReadProperty2(), you may use the BACstacInitContentsToBuf(). In the above example, instead of BACstacInitContentsToVar(), you might call

```
BACstacInitContentsToBuf(
    &Value,
    &Temperature,
    sizeof(Temperature));
```

Reading Multiple Properties

When you have a need to read more than one Property of the same Object at a time, or to read many properties of different Objects, it is convenient to use the BACnet ReadPropertyMultiple service.

Suppose your application needs to know Notification Class values used by all the intrinsic alarming objects of a specified Device. Suppose further you have already obtained the Object handles of all the IO Objects in that Device (see 2.4) and these handles are stored in hObjs buffer, their number being stored in nObjs. In the following example, through the reading of the Notification Class property, an application discovers which IO Objects support intrinsic alarming:

```
#define MAX_PROP_NUM 1
BACSTAC_HDEVICE hDev;                       /* initialized elsewhere */
BACSTAC_HOBJECT hObjs[MAX_OBJ_NUM];         /* initialized elsewhere */
BACSTAC_OBJECT_COUNT nObjs;                 /* initialized elsewhere */
BACSTAC_READ_LIST readList[MAX_OBJ_NUM];
BACSTAC_READ_RESULT_LIST resList[MAX_OBJ_NUM];
BACSTAC_READ_RESULT_ITEM resItems[MAX_OBJ_NUM][MAX_PROP_NUM];
BACSTAC_OBJECT_TYPE objType[MAX_OBJ_NUM];
BACSTAC_UNSIGNED notClass[MAX_OBJ_NUM];
int i;
/* Specify properties to read */
BACSTAC_PROP_REF propRef[MAX_PROP_NUM]=
{
    {PROP_NOTIFICATION_CLASS,  BACSTAC_VOID_INDEX}
};
/* For every Object in the list of Object handles */
for(i = 0; i < nObjs; i++)
{
    /* Initialize readList with an Object and its Properties */
    BACstacInitRPMAccessList(
        readList,
        hNCObjs[i],
        propRef,
        MAX_PROP_NUM);

    /* Initialize resItems with buffers for specified Properties */
    notClass[i] = 0;
    BACstacInitRPMResultItem (
```

```
        &resItems[i][0],
        &notClass[i],
        sizeof(BACSTAC_UNSIGNED));
    /* Initialize resList element with a buffer for Result Items */
    BACstacInitRPMResultList (&resList[i], resItems[i], PROP_NUM);
}
/* Read the both specified Properties of all Objects in the list */
BACstacReadPropertyMultiple2 (hDev, readList, &resList, nObjs, NULL, NULL);
```

The notification class values will be placed in the notClass array. The result list indicates whether each property was successfully read — so the result list can also be used to flag those objects supporting intrinsic alarming.

```
BACSTAC_BOOLEAN isIntrinsicAlarmingObject[MAX_OBJ_NUM];
/* For every Object in the list of Object handles */
for(i = 0; i < nObjs; i++)
{
    isIntrinsicAlarmingObject[i] =
        BACstacGetRPMResultItemStatus(&resItems[i][NC]);
}
```

Writing to Properties

Setting the value of a Property has two steps. As with querying a value, your application needs to set up a Property Value data type and then write the value to the remote Object. When writing the Present-Value property, a BACnet Write-Property request will be sent to the remote Device. An example of writing a value to a remote Property is:

```
BACSTAC_HOBJECT hBinaryValue2; /* initialized elsewhere */
BACSTAC_PROPERTY_CONTENTS Value;
BACSTAC_BINARY_PV PowerStatus;
/* attach PowerStatus to the Property Contents data type */
PowerStatus = BACSTAC_BINARY_ACTIVE;
BACstacInitContentsToVar(
    &Value,
    DATA_TYPE_ENUM,
    &PowerStatus,
    sizeof(PowerStatus));
BACstacWriteProperty(
    hBinaryValue2,
    PROP_PRESENT_VALUE,
    BACSTAC_VOID_INDEX,
    BACSTAC_PRI_MANUAL_OPERATOR,
    &Value,
    NULL);
```

If a Property is an array, then you can specify an array index when making a Write-Property request to access one element of the array. The index value BACSTAC_ENTIRE_ARRAY can be used to access the entire array with one request. The index value BACSTAC_VOID_INDEX should be used for non-array Properties.

If a Property is commandable (see Clause 19 of the BACnet Standard for a description of this concept), you can specify a priority when making a Write-Property request. The priority BACSTAC_VOID_PRI should be used for non-commandable Properties.

Chapter 4. Creating a Server Application

You can create a BACstac Server application by defining the Objects to be maintained by the BACstac Server and by providing *hooks* and *callbacks* to interact with your physical device (see Chapter 7 of this Guide for the detailed info on hooks). The Objects will be manipulated by BACnet requests and your application can also manipulate the Objects directly. The routines used in the examples below are documented in the *Server API* section of the BACstac *Programmers' Reference*. You can link the BACstac routines to your application by using the BACSRVx.LIB libraries. You must include the BACstac Server application header file in your source code files:

```
#include <
bacsrv.h>
```

Defining Objects in a Device Template

You define the Objects maintained by your Server application by creating a Device template with a .TPI file. A .TPI file contains a text BACnet Protocol Implementation Conformance Statement (Text PICS) description of the Objects. The easiest way to create a .TPI file for an application is to modify an existing .TPI file with a text editor. See *Appendix B - .TPI File Format* in the *Programmers' Reference* for a description of the syntax of the .TPI file. A .TPI file must contain a Device Object. Other Objects may be created in the Device dynamically.

A compiler (CBNOBJ.EXE) is used to convert this description into a C source file which defines the static data structures representing the Objects. A pointer to the top-level data type is then passed into the BACstac server initialization routine. The name of the top-level data type is set by a command line argument for the Object Compiler.

The Cimetrics Compiler of BACnet Objects (CBNOBJ) is a tool which can be run separately from the Server application. It takes three arguments: the source file name, the output file name, and the name of the top-level data type. See *Appendix A - CNBOBJ.EXE* in the *Programmers' Reference* for a description of usage of this utility, including the error messages it generates. For example, to compile the Objects for the Temperature Sensor example application, you would execute:

```
cbnobj tempobj.tpi tempobj.c tempDevice
```

You would then link the resulting file to your application, as demonstrated in the next section.

Initialization of the BACstac Server

Your application must initialize the BACstac server before it will respond to BACnet network requests. You do this by calling the API routine BACstacServerInit(). Its arguments include a pointer to the data structures describing the local Objects in the application and the BACnet Device Object Instance Number that the application will use to uniquely identify itself on the BACnet inter-network. An example of use of BACstacServerInit() in an application is:

```
#include <
bacsrv.h>     /* define the server API */
/* top level device data structure in tempobj.c */
extern const struct DEVICE_TEMPLATE tempDevice;
main()
{
    BACSTAC_SRV_INIT init_data;
    /* for compiling with BACstac version 7.6, Instance Number 1966 */
    BACstacInitSrvInit(&init_data,7,6,&tempDevice,1966);
```

```
    BACstacSetSrvInitMaxObjects(&init_data,10);
    BACstacServerInit(&init_data,NULL);
    ...
    BACstacUnlockData();
    ...
}
```

The maximum number of Objects set by BACstacSetSrvInitMaxObjects() must not exceed 64k.

If you do not call BACstacSetSrvInitMaxObjects(), this number will be set to a default value (32 for the current version).

The Version numbers (Major and Minor) supplied by your application are checked by the BACstac initialization routine to ensure that your application is compatible with the version of the BACstac library with which your application was built.

When the BACstac Server is initialized, the object database is locked, allowing you to modify their values and preventing other threads as well as the default BACstac action from reading or modifying their values while data may be still an inconsistent state. When the initialization is complete you must call to the BACstacUnlockData() routine to unlock the database from exactly the same thread that called BACstacServerInit(). Failing to do so will leave the stack in an unworkable state.

Changing property attributes and values

The BACstac represents Objects and Devices by using *handles*. In a server application there is a unique handle for the local Device. Your application obtains handles for objects in the local device by specifying its BACnet Object ID (type and instance number). Your application can then manipulate any property in the object using its handle and the Property Identifier.

Your application can change the Access attribute of properties to prevent them from being modified by a BACnet Write-Property Write-Property-Multiple request. Note: this attribute does not affect ability to change this property from your program using BACstacStorePropertyInstance, but only the default action performed for this service. The example below shows how to obtain all relevant handles and modifying the access attribute of a Property.

```
BACSTAC_HDEVICE hDevice;
BACSTAC_HOBJECT hAnalogValue1;
BACSTAC_OBJECT_ID ObjID;
/*
 * receiving the object handle; this can be done
 * once and the handle can be reused later
 */
hDevice = BACstacGetLocalDeviceHandle();
BACstacInitObjectID(&ObjID,OBJ_ANALOG_VALUE,1);
hAnalogValue1 = BACstacFindObjectByID(hDevice, ObjID);
/*
 * Change property access to read-only
 */
BACstacSetPropertyAccess(
    hAnalogValue1,
    PROP_PRESENT_VALUE,
    BACSTAC_ACCESS_READ_ONLY);
```

A BACstac Server application can register callback routines for the default BACstac actions to use to access physical hardware when servicing Read-Property and Write-Property requests. Any Property of any Object can be flagged as attached to a callback. For an attached property, the callback is called instead of reading or writing the value from the database. This is useful for properties that represents a value supplied by a physical device or any other sources that changes often.

```
BACSTAC_HOBJECT hAnalogValue1; /* initialized elsewhere */
/* mark the present-value Property of an analog-value Object */
/* to use the Read-Property callback */
BACstacSetCallbackAttachment(
    CB_READ,
    hAnalogValue1,
    PROP_PRESENT_VALUE,
    BACSTAC_TRUE);
```

A BACstac Server application can read or write the values of properties directly without generating a Read-Property or Write-Property service request. For that purpose BACstacStorePropertyInstance() and BACstacRetrievePropertyInstance2() routines, which takes as parameters the object handle, property ID, optional index (if the property is an array), and a pointer to a Property Contents data type. The direct access routines do not invoke any default actions or call hooks or callbacks. For example, a server application could update the present-value property in an analog-input object in the following way:

```
BACSTAC_HOBJECT hAnalogInput; /* initialized elsewhere */
BACSTAC_REAL X;
BACSTAC_PROPERTY_CONTENTS Value;
/* connect X with the Property Contents data type */
BACstacInitContentsToVar(
    &Value,
    DATA_TYPE_REAL,
    &X,
    sizeof(X));
/* read the current Present Value into X */
BACstacRetrievePropertyInstance2(
    hAnalogInput,
    PROP_PRESENT_VALUE,
    BACSTAC_VOID_INDEX,
    &Value,
    NULL);
X = X*0.95;  /* modify X */
/* update the Present Value */
BACstacStorePropertyInstance(
    hAnalogInput,
    PROP_PRESENT_VALUE,
    BACSTAC_VOID_INDEX,
    &Value);
```

Unfortunately, this example does not take into account one important factor – the value of property can be modified concurrently from the network using a Write-Property request. So if this property is writable and the operation needs to be atomically, the application should lock the database as it is shown in the next section.

Synchronization of data access

Your application can use multiple threads to read and write properties from database's objects as it was shown in the previous section. Each of reading or writing operation is atomic, and there is no need to use any additional synchronization for that. However, it may be necessary to perform a sequence of operation atomically, so there was no interference from other threads including the BACstac default action, which operates on an internal BACstac thread. For that purpose BACstacLockData() and BACstacUnlockData() routines are used.

The BACstacLockData() routine locks the database from any modification from other threads. If the database is already locked then this routine will wait till the thread that locked database unlocks it. Therefore it is very important to call BACstacUnlockData() for every call to BACstacLockData() from the same thread, otherwise the database will remain locked forever and all other BACstac routines that have access to the database will not work properly including BACstacClose().

As an example, let's consider reducing the value of the present-value of an analog-input object as in the previous section, but this time we will make this change atomically, so if another device trying to write to it concurrently, the result will be still correct.

```
BACSTAC_HOBJECT hAnalogInput; /* initialized elsewhere */
BACSTAC_REAL X;
BACSTAC_PROPERTY_CONTENTS Value;
/* connect X with the Property Contents data type */
BACstacInitContentsToVar(
    &Value,
    DATA_TYPE_REAL,
    &X,
    sizeof(X));
/* now, it is time to lock the database */
BACstacLockData();
/* read the current Present Value into X */
BACstacRetrievePropertyInstance2(
    hAnalogInput,
    PROP_PRESENT_VALUE,
    BACSTAC_VOID_INDEX,
    &Value,
    NULL);
X = X*0.95;  /* modify X */
/* update the Present Value */
BACstacStorePropertyInstance(
    hAnalogInput,
    PROP_PRESENT_VALUE,
    BACSTAC_VOID_INDEX,
    &Value);
/* Don't forget to unlock! */
BACstacUnlockData();
```

Creating a Local Object

A BACstac Server application can create new local Objects, within the maximum Object number limits set at initialization. The application must provide a description of all the properties to appear in the Object. For each property, the Property Instance description contains the Property ID, wh ether the Property access permission is read-only or

read-write , whether read and write callbacks are attached, and a Property Contents data type containing an initial value for the Property.

```
BACSTAC_HDEVICE hDev; /* initialized elsewhere */
BACSTAC_HOBJECT hNewAnalogInput;
BACSTAC_OBJECT_ID objID;
BACSTAC_DB_PROPERTY propList[N_MAX_PROPS];
BACSTAC_DB_OBJECT dbObject;
BACSTAC_PROPERTY_COUNT nPropCount;
BACSTAC_PROPERTY_COUNT firstFailed;
/* the new Object will be an analog-input with instance number 7 */
BACstacInitObjectID(&objID,OBJ_ANALOG_INPUT,7);
/* an application routine to set up the propList */
nPropCount = BuildAnalogInputPropList(&objID, propList, N_MAX_PROPS);
BACstacInitDbObject(&dbObject, propList, nPropCount);
/* create the new Object */
BACstacCreateDBObject(hDev, &dbObject, &hNewAnalogInput, &firstFailed);
```

Deleting a Local Object

A BACstac Server can delete a local Object (created by BACstacCreateDBObject() or created at initialization from a .TPI file template) by using the BACstacDestroyObject() routine:

```
BACSTAC_HOBJECT hOldObj; /* initialized elsewhere */
BACstacDestroyObject(hOldObj);
```

Generating an I-Am Broadcast

Your server application can broadcast a I-Am message to inform client applications of its presence on the BACnet. It is suggested that your application broadcast an I-Am message at least once after initialization. The BACstac server will also respond automatically to Who-Is requests with the I-Am message. An example of broadcasting an I-Am message is:

```
BACSTAC_HDEVICE hDevice;  /* Initialized previously */
BACSTAC_ADDRESS Address;
BACstacInitAddrGlobalBroadcast(&Address);
BACstacIAm(hDevice,&Address);
```

Generating an I-Have Broadcast

Your server application can broadcast an I-Have message to inform client applications about the existence of the given object. The BACstac server will also automatically respond to Who-Has requests with an I-Have message. An example of broadcasting an I-Have message is:

```
BACSTAC_HOBJECT hObject;
BACSTAC_ADDRESS Address;
/* hDevice and hObject are initialized previously */
```

```
BACstacInitAddrGlobalBroadcast(&Address);
BACstacIHave(NULL, hObject, &Address);
```

This message is also automatically sent by the BACnet server in response to a BACnet Who-Has request if the Who-Has service application hook is not registered.

Generating a Local Read-Property Request

Your application can read a property value as described in the Section called *Changing property attributes and values*. However, if the application needs to emulate receiving a Read-Property request from the network, it can send a local Read Property request to itself. This request will indistinguishable from one received from another device in all respect except having exactly the same source and destination address.

Generating a local Read-Property request has two steps: setting up the Property Value data type, and then reading the value into the space provided by the data type. When reading a Property, a hook or callback may be invoked to read from the physical device. An example of reading a local Property value is:

```
BACSTAC_HOBJECT hAnalogInput1; /* initialized previously */
BACSTAC_PROPERTY_CONTENTS Value;
BACSTAC_REAL Temperature;
BACstacInitContentsToVar(
    &Value,
    DATA_TYPE_REAL,
    &Temperature,
    sizeof(Temperature));
BACstacReadProperty2(
    hAnalogInput1,
    PROP_PRESENT_VALUE,
    BACSTAC_VOID_INDEX,
    &Value,
    NULL,
    NULL);
/* Now the value has been copied into Temperature */
```

Generating a Local Write-Property Request

Your application can write a property value as described in the Section called *Changing property attributes and values*. However, if the application needs to emulate receiving a Write-Property request from the network, it can send a local Write-Property request to itself. This request will indistinguishable from one received from another device in all respect except having exactly the same source and destination address.

Your application also uses two steps to generate a local Write-Property request. As with querying a value, you need to set up a Property Contents and then write the value into the BACstac Server. When writing to a Property, a hook or callback may be invoked to write the value to the physical device. An example of writing a value to a local Property is:

```
BACSTAC_HOBJECT hBinaryValue2; /* initialized previously */
BACSTAC_PROPERTY_CONTENTS Value;
BACSTAC_ENUM PowerStatus;
PowerStatus = BACSTAC_BINARY_ACTIVE;
```

```
BACstacInitContentsToVar(
    &Value,
    DATA_TYPE_ENUM,
    &PowerStatus,
    sizeof(PowerStatus));
BACstacWriteProperty(
    hBinaryValue2,
    PROP_PRESENT_VALUE,
    BACSTAC_VOID_INDEX,
    &Value,
    NULL);
```

Generating a Remote Read-Property Request

Generating a remote Read-Property request has three steps: setting up the property contents for the given type, initialization read property information, and reading the value into the space provided by the data type. The following example demonstrates how to read the present value of an analog-input object of the given device.

```
BACSTAC_ADDRESS *pDestDevice; /* initialized elsewhere */
BACSTAC_PROPERTY_CONTENTS value;
BACSTAC_REAL temperature;
BACSTAC_READ_INFO readInfo;
/* attach powerStatus to the property contents  */
BACstacInitContentsToVar(
    &value,
    DATA_TYPE_REAL,
    &temperature,
    sizeof(temperature));
/* Initialize read information, which consists of
 * an object identifier, and property ID.
 */
BACstacInitObjectID(&ObjID,OBJ_ANALOG_INPUT,1);
BACstacInitReadInfo(
    &readInfo,
    &objectID,
    PROP_PRESENT_VALUE);
/* Send a Read-Property request and wait for a response */
BACstacReadPropertyEx2(
    NULL,
    pDestDevice,
    &readInfo,
    NULL, NULL, NULL,
    &value,
    NULL, NULL);
```

If a read property is an array, and the application needs to read one element of it then the BACstacSetReadInfoIndex() routine can be used to specify the desired index in the read information.

Generating a Remote Write-Property Request

Your application uses three steps to generate a remote Write-Property request. First the application should set the property contents to the value to be written. Second, to initialize the write information. Third, to send a Write-Property request and waits for a response. The following example demonstrates how to write to the present value of an binary-output object of the given device.

```
BACSTAC_ADDRESS *pDestDevice; /* initialized elsewhere */
BACSTAC_WRITE_INFO writeInfo;
BACSTAC_OBJECT_ID objectID;
BACSTAC_PROPERTY_CONTENTS value;
BACSTAC_ENUM powerStatus;
powerStatus = BACSTAC_BINARY_ACTIVE;
/* attach powerStatus to the Property Contents data type */
BACstacInitContentsToVar(
    &value,
    DATA_TYPE_ENUM,
    &powerStatus,
    sizeof(powerStatus));
/* Initialize write information, which consists of an
 * object identifier, property ID, and pointer to the
 * property contents with the value to be written.
 */
BACstacInitObjectID(&ObjID,OBJ_BINARY_OUTPUT,1);
BACstacInitWriteInfo(
    &writeInfo,
    &objectID,
    PROP_PRESENT_VALUE,
    &value);
/* Send a Write-Property request and wait for a response */
BACstacWritePropertyEx(
    NULL,
    pDestDevice,
    &writeInfo,
    NULL, NULL, NULL, NULL);
```

If a written property is an array, and the application needs to write one element of it then the BACstacSetWriteInfoIndex() routine can be used to specify the desired index in the write information.

Registering Callbacks

When a Read-Property or Write-Property request for a property is being processed, the BACstac server can invoke a callback routine that you provide to read values from or write values to the physical device. The write callback is given a pointer to a Property Contents data type, and it can use the given value to update the physical device. A Read-Property callback can store the new value into the database using BACstacStorePropertyInstance(), which will be later used by BACstac to respond the Read Property request.

```
BACSTAC_CALLBACK_STATUS BACstac_callback ReadPropertyCallback(
BACSTAC_HOBJECT hObj,
BACSTAC_PROPERTY_ID PropertyID,
BACSTAC_ARRAY_INDEX Index);
BACstacSetReadCb(ReadPropertyCallback);
```

If a callback isn't registered, or if a callback isn't attached to a particular Property, the value of the Property in the local Object will still be modified or read in response to BACnet requests.

For a commandable property, the commandable write callback will be invoked instead of the write callback if it is installed. The commandable write callback allows to validate all values written to the commandable property and not only the value having the highest priority now. This allows this callback to return an error if the given value is out of range or cannot be accepted for any other reason. If the write commandable callback is not installed or the write commandable callback returns CALLBACK_STATUS_DEFAULT then the write callback for the commandable property will be invoked, but the write callback for a commandable property should not return an error (the returned value by the write callback is ignored for commandable properties).

It is important to use the BACstac_callback modifier in callback definitions and prototypes, as in the above example.

It should be noted that the ReadPropertyMultiple and WritePropertyMultiple services are translated by the BACstac default actions into simple ReadProperty and WriteProperty services. So an application provide support for both ReadProperty and ReadPropertyMultiple with a single callback.

How to Use Callbacks

A callback is invoked when default actions to obtain the current value of this property or to pass a new value of the property to application (typically to synchronize the value stored in the BACstac database and in the physical device). However, not every invocation of a callback requires to access to physical device. In particular, when present-value properties are read or written, the default actions may not need to access the physical device depending on the current values of the out-of-service.

In the following example, a read callback provides update values for the present-value Property of an analog-input Object and a binary-input Object.

```
/* include dbacc.h from examples to simplify get and set operations
   for commonly use BACstac data types */
#include "dbacc.h"
BACSTAC_CALLBACK_STATUS BACstac_callback ReadPropertyCallback(
    BACSTAC_HOBJECT hObj,
    BACSTAC_PROPERTY_ID PropertyID,
    BACSTAC_ARRAY_INDEX Index)
{
    BACSTAC_BOOLEAN OutOfService;
    BACSTAC_OBJECT_TYPE Type;
    BACSTAC_REAL X;
    BACSTAC_BINARY_PV BinPV;
    BACSTAC_CALLBACK_STATUS Status;
    /* The Present_Value property is decoupled from the physical input
       and will not track changes to the physical input when the value
       of Out_Of_Service is TRUE. */
    GetBoolProperty (hObj, PROP_OUT_OF_SERVICE, &OutOfService);
    if (OutOfService)
        return CALLBACK_STATUS_DEFAULT;
    /* This particular callback should only be invoked for
       present-values.  */
    switch(Type)
    {
    case ANALOG_INPUT:
        /* read the value */
```

```
        X = ReadMyDevice();
        /* store the new value into the database */
        StoreRealProperty(hObj, PropertyID, X);
        Status = CALLBACK_STATUS_OK;
        break;
    case BINARY_INPUT:
        /* read the value */
        BinPV = IsMyDeviceActive();
        /* store the new value into the database */
        StoreEnumProperty(hObj, PropertyID, BinPV);
        Status = CALLBACK_STATUS_OK;
        break;
    default:
        /* shouldn't get here, but let default actions know */
        /* that we didn't read the physical device, so the */
        /* previous value of the Property should be used */
        Status = CALLBACK_STATUS_DEFAULT;
        break;
    }
    return Status;
}
```

Chapter 5. Creating a Gateway Application

BACstac Routing Edition software includes Gateway library in addition to the Client and Server libraries of the BACstac Standard Edition software.

A BACstac Server allows a control device to appear as a BACnet Device to the BACnet network. A BACstac Gateway allows one or more control devices to appear as a set of BACnet Devices on a virtual BACnet network connected to the real BACnet network by a router. The BACstac Gateway includes the capabilities of a BACnet router. The mechanism of a BACnet Gateway is described in *Annex H,* sections H.1 and H.2 of the *BACnet standard*.

A BACstac Gateway application can be used to make devices on a proprietary network act as Servers to BACnet requests. The BACstac Client library must be used to allow devices on a proprietary network to control BACnet Devices.

The routines used in the examples below are documented in the *Server API* and *Gateway API* sections of the BACstac *Programmers' Reference*. You can link the BACstac routines to your application by using the BACGTWx.LIB libraries. You must include the BACstac Gateway application header file in your source code files:

```
#include <
bacgtw.h>
```

How Is a Gateway Different from a Server?

A BACstac Gateway is different from a BACstac Server in three ways: it contains BACnet router functionality, it maintains multiple local Devices, and it allows dynamic creation and deletion of local Devices.

The BACnet Standard includes a number of network layer messages, described in Clause 6. These messages are used by the network layer of BACnet applications to discover paths to Devices on remote BACnet networks. BACnet routers must be able to respond to these network layer service requests. A Gateway application acts like a router to a virtual BACnet network, and must respond to these network layer service requests too. The BACstac Gateway performs these functions transparently.

A Gateway maintains multiple local Devices, whereas a Server has a single local Device. As a result, the mechanism for discovering local Device handles is somewhat different in a Gateway than in a Server. Once a Device handle is obtained, both Server and Gateway applications can manipulate the Device (obtain information about the device or obtain Object handles) in the same manner. If a Gateway uses hooks to override the default actions for a service, the hooks must do a bit more work than their counterparts in a Server. Hooks are given source and destination Addresses as arguments. A Server can assume that the destination Address refers to the single local Object, but a Gateway must use the destination Address to find the corresponding local Device handle.

In a Server, the single local Device is created during initialization of the BACstac using a Device template created from a .TPI file with the CBNOBJ utility program. In a Gateway, Devices can be created and deleted at any time.

BACnet Virtual Networks

A BACnet system can contain multiple BACnet local networks, connected by BACnet routers. The mechanism for achieving this is described in Clause 6 of the BACnet Standard. Each BACnet local network is identified by a BACnet network number. The BACnet routers connecting the networks maintain the BACnet network number information.

When a BACstac Gateway application is initialized, the application must provide two BACnet network numbers: the BACnet network number of the real BACnet network to which the Gateway is attached, and the BACnet network number of the virtual BACnet network maintained by the Gateway.

A BACstac Gateway maintains a router table to allow BACnet requests to reach the correct Device in the Gateway and to allow Gateway replies and broadcasts to reach the correct BACnet network. A BACstac Gateway supports the following network layer messages: Who-Is-Router-To-Network, I-Am-Router-To-Network, and Initialize-Routing Table (query only).

The handling of the router table and network layer messages occurs transparently to a Gateway application.

When a BACstac Gateway creates a Device, it must ensure that the new Device has a unique (within the Gateway) virtual MAC address and a unique (over the entire BACnet inter-network) Device Object instance number. The BACstac checks for uniqueness on the local Gateway, but the BACnet standard requires Device Object instance numbers be unique over entire BACnet inter-network.

Initialization of the BACstac Gateway

Your application must initialize the BACstac Gateway before it will respond to BACnet network requests. You do this by calling the API routine BACstacGatewayInit(). Its argument includes configuration parameters for the application and the port identifier (taken from the routing table) corresponding to the virtual network you choose.

The following example initializes a Gateway application:

```
#include <
bacgtw.h>
#define MY_PORT_IDENTIFIER 5
main()
{
    BACSTAC_GTW_INIT InitData;
    BACstacInitGtwInit(&InitData,7,6,MY_PORT_IDENTIFIER); /* for BACstac v7.6 */
    BACstacSetGtwInitMaxDevices(&InitData,30);
    BACstacGatewayInit(&InitData);
    ...
    BACstacUnlockData();
    ...
}
```

The maximum number of Devices set by BACstacSetGtwInitMaxDevices() must not exceed 64k.

If you do not call BACstacSetGtwInitMaxDevices(), this number will be set to a default value (32 for the current version).

The Version numbers (Major and Minor) supplied by your application are checked by the BACstac initialization routine to ensure that your application is compatible with the version of the BACstac library with which your application was built.

The call of BACstacUnlockData() unlocks BACstac object database after initialization, which is necessary for subsequent service processing. BACstacGatewayInit() procedure (and BACstacServerInit() — see the Section called *Initialization of the BACstac Server* in Chapter 4) locks the BACstac database to prevent the data. Therefore you should watch your code to call the unlocking procedure within every thread that calls BACstacGatewayInit() procedure.

Creating a Device Using a Device Template

A Gateway can create Devices on its virtual BACnet network at any time. The maximum number of Devices that a Gateway can maintain is set when the Gateway is initialized.

When a Device is created, a Gateway application must provide a virtual MAC address for the Device, and a unique Device Object instance number. A Device template, created from a .TPI file by the CBNOBJ utility program, is used to describe the Objects and Properties in the

new Device. (see the Section called *Defining Objects in a Device Template* in Chapter 4). Information in the template (including initial values for all the Properties) is copied into the new Device. A template can be used to create multiple Devices. Once Devices have been created, Objects can be added to or removed from them,.

For instance, suppose that a .TPI file containing just a Device Object is called `emptydev.tpi`. A .C file containing the Device template can be created with the CBNOBJ utility. The command line arguments of CBNOBJ are the source file, destination file and external name for the template:

```
CBNOBJ emptydev.tpi emptydev.c emptyDevice
```

You can compile the .C file containing the Device template separately and link the resulting object file to your Gateway application. To create a Device, a Gateway application could use the BACstacCreateDeviceFromTemplate() API routine, as in the following example:

```
#define NMAXOBJ 25
extern const struct DEVICE_TEMPLATE emptyDevice;
/* in EMPTYDEV.C created by CBNOBJ from EMPTYDEV.TPI */
/* make the virtual MAC address look like an Ethernet,
   and choose some Address selection scheme */
BACSTAC_BYTE pMAC[] = {'\0','\0','\001','\004','\009','\002'};
BACSTAC_ADDRESS Address;
BACSTAC_INST_NUMBER InstNum = 1492;
BACSTAC_HDEVICE hDev;
/* initialize the MAC address for the Device */
BACstacInitAddrLocal(&Address,pMAC,BACSTAC_MAC_ETHERNET_LENGTH);
BACstacCreateDeviceFromTemplate(
    &emptyDevice,
    &Address,
    InstNum,
    NMAXOBJ,
    &hDev);
```

A BACstac Gateway application can use multiple templates for creating Devices.

Deleting a Device

A BACstac Gateway application can delete a Device and the Objects that it contains by using the BACstacDeleteDevice() API routine:

```
BACSTAC_HDEVICE hDev; /* initialized elsewhere */
BACstacDeleteDevice(hDev);
```

Chapter 6. Resolving Device ID

BACstac includes a Device ID resolver that provides a convenient way to resolve required Device ID to address. The resolver automatically caches the most recently used Device ID in the internal cache. If the requested Device ID is unknown, it tries to resolve it using the Who-Is request and waiting for the I-Am indition. If the application has a hook for I-Am, it should invoke the default action (by return HOOK_DEFAULT), so the resolver will receive that I-Am indication.

This API is available for all types of applications (CLIENT, SERVER, GATEWAY). The size of the resolver cache specified in the corresponding section of the APIL configuration. This API is intended to resolve individual Device ID and not suitable to enumerate all devices existing on the BACnet internetwork.

When the specified Device ID is unknown, the resolver uses a global Who-Is request with the given Device ID as the filter. If the application wants to limit the search to particular BACnet networks for all or some Device ID values, it can provide its own Who-Is sender using BACstacSetResolverWhoIsSender().

A gateway application needs to know the source address that should be used to send a Who-Is request. The application can either provide a fixed value of the source address using BACstacSetResolverSrcAddress for the default Who-Is sender, or to install its own Who-Is sender as described above.

The resolver uses the default APDU Timeout and Retry Count when it searches the specified Device ID. Thus if the specified Device ID cannot be located, it takes APDU_Timeout * (Retry_Count + 1) before the final timeout expires.

The resolver API provides a few ways to resolve Device ID, which are described below. The optimal choice depends on application requirements. Simple application that deals with a small number of Device ID can use synchronous search. However, the synchronous search can be blocked for a while, so it is not appropriate to deal with a large number of devices. Asynchronous search allows to resolve multiple Device ID without blocking.

Synchronous Device ID resolution

The synchronous API is easy to use as it requires to call only one API function:

```
status = BACstacResolveDeviceID(deviceID, &address, &iAmInfo);
```

where deviceID is the device instance number that needs to be resolver. On successful return, the variable pointed by 'address' contains the BACnet address of the device and the variable pointed by 'iAmInfo' contains additional information received in I-Am. If I-Am information is not needed the last parameter may be NULL.

Asynchronous Device ID resolution

The asynchronous API allows to avoid being blocked while the resolver is trying to resolve Device ID. Therefore, this API is more suitable when the application needs to resolve a large number of devices or it is undesirable for the calling thread to be blocked for long time.

```
/* Define MY_RESOLVE_OPER, so we can pass additional parameters
 * to the asynchronous callback routine. If you do not need any
 * additional parameters then there is no need for it. */
typedef struct tagMY_RESOLVE_OPER
{
   BACSTAC_RESOLVE_OPER base;
   // ... additional application data
```

```
} MY_RESOLVE_OPER;
/*
 * IMPORTANT: The ACR is invoked asynchronously on the BACstac internal thread!
 * Make sure that access to any data uses proper synchronisation primitives!
 */
void BACstac_callback MyResolveAcr(
    BACSTAC_STATUS status,
    BACSTAC_I_AM_INFO *iAmInfo
    BACSTAC_ADDRESS *deviceAddress,
    BACSTAC_RESOLVE_OPER *operBase)
{
   /* We rely on that BACSTAC_RESOLVE_OPER is the first field */
   MY_RESOLVE_OPER *oper = (MY_RESOLVE_OPER*)operBase;
   if (status == BACSTAC_STATUS_OK)
   {
       // Use obtained device address
   }
   else
   {
       // Report the error
   }
}
MY_RESOLVE_OPER oper;
void Foo()
{
    BACstacInitResolveOper(&oper.base);
    // ... set other oper data if necessary
    /* IMPORTANT: pointer to BACSTAC_RESOLVE_OPER must remain
     * valid until the ACR callback is invoked. */
    status = BACstacStartResolveOper(deviceID, &oper.base);
    // ... verify status and report error if necessary
}
```

Ahead-of-time Device ID resolution

If an application knows ahead of time that it may need some Device ID to use later, it can ask the resolver to resolve this address. It is similar to the asynchronous resolution but the application does not need any ACR to be invoked when the device ID is resolved. So, the resolver operation parameter is NULL.

```
status = BACstacStartResolveOper(deviceID, NULL);
// ... verify status and report error if necessary
```

Later, the application can use a non-blocking call to obtain the device address from the resolver cache if it is ready:

```
status = BACstacGetDeviceAddressByID(deviceID, &address, &iAmInfo);
```

The above call will fail if the Device ID has not been resolved yet. So, this approach is not suitable everywhere, but it can be useful when a device needs to send Event Notifications. In this case, the list of recipients is rarely changed, so if ahead of time resolution started when a new device is added to the list, we can assume that there is enough time to pass before the first notification. Moreover, delaying sending event notifications till the device ID is resolved may result in sending obsolete notification, so it is not desirable anyway.

Chapter 7. BACstac Hooks

A hook is an application routine that allows your application to handle BACstac service requests. Defining hooks is one of the most important features of low-level programming style. Still, you can use hooks in application using high level API in general. In this case using hooks highly increases your application's capability. To learn about BACstac default method of handling requests see below (the Section called *Default Actions and Building Blocks*).

A hook is given all of the parameters that appear in the service request, and the hook has complete control over sending a reply. A hook can choose to pass a service request back to BACstac to be handled with default method; the hook can use helper routines to perform parts of the BACstac's default actions itself or better to control the request handling completely.

Default Actions and Building Blocks

BACstac has its own mechanism of processing service request. Thus, if you let the request processing to BACstac, it performs its *default actions*. You can use BACstac default actions to handle service requests when programming with high level API and BACstac Object Database.

If your application needs to perform some parts of these default actions such as parameters check-up, checking if a property is writable or checking the BACstac Object Database for certain Objects and Properties existence you can use the specific *building block routines*. Building blocks implement much of the logic of the default actions and help you to assemble a simple hook when you use the Database. Please refer to the *Programmers' Reference* for more information.

However, the default actions of the BACstac may appear too general to handle service requests properly in the particular application. You may also need to define your own set of actions to perform. In such cases registering a hook to handle the service request is preferable.

Actually, default actions for many services return a reject response with REJECT_UNRECOGNIZED_SERVICE reason. Here are the services that *can* be effectively processed through default actions:

```
Who-Is, Who-Has, I-Am, I-Have, AddListElement, RemoveListElement, ReadProperty, WriteProperty, ReadP
```

If you need to enable one of the other services (see *Programmers' Reference* for the complete list of supported hooks) in your application you should define hook for it (see below).

```
/* NOTE: This listing uses BACstacVerifyTSMCompletion() from examples */
BACSTAC_HOOK_STATUS BACstac_hook ReadHook(
    BACSTAC_HTSM hTSM,
    BACSTAC_ADDRESS *pSourceAddress,
    BACSTAC_ADDRESS *pDestAddress,
    const BACSTAC_READ_INFO *pServiceInfo)
{
    BACSTAC_STATUS statX;
    const BACSTAC_OBJECT_ID *pObjId;
    BACSTAC_INST_NUMBER deviceID, instNum;
    BACSTAC_READ_INFO riBuf;
    BACSTAC_HDEVICE hDev;
    /* If it is a gateway then check that the destination address
     * is for an existing device */
#ifdef BACSTAC_GATEWAY
    hDev = BACstacDeviceAddressToHandle(pDestAddr);
    if (hDev == BACSTAC_INVALID_HANDLE)
```

```
    {
        BACstacDestroyTSM(hTSM);
        return HOOK_STATUS_OK;
    }
#else
    hDev = BACstacGetLocalDeviceHandle();
#endif
    /* Check whether we are interested in the specified object type and
     * property-identifier. If not, then return HOOK_STATUS_DEFAULT */
    if (BACstacGetReadInfoPropID (pServiceInfo) != PROP_ACTIVE_COV_SUBSCRIPTIONS)
        return HOOK_STATUS_DEFAULT;
    pObjId = BACstacGetReadInfoObjectIDPtr (pServiceInfo);
    if (BACstacGetObjectIDType (pObjId) != OBJ_DEVICE)
        return HOOK_STATUS_DEFAULT;
    /* Check that the specified object instance exists. If the object does not
     * exist then send (OBJECT, UNKNOWN_OBJECT) error in response.
     * For the device object, instance number 4194303 means any instance number,
     * and we should send our actual instance number in response.
     */
    /* Here were are dealing with the device object... */
    BACstacGetDeviceInstNum (hDev, &deviceID); /* our actual instance number */
    instNum = BACstacGetObjectIDInstNum (pObjId); /* requested instance number */
    if (instNum == 4194303)
    {
        riBuf = *pServiceInfo;
        BACstacSetObjectIDInstNum(BACstacGetReadInfoObjectIDPtr(&riBuf), deviceID);
        pServiceInfo = &riBuf;
    }
    else if (instNum != deviceID)
    {
        statX = BACstacSrvcError (hTSM, ERR_CLASS_OBJECT, ERR_CODE_UNKNOWN_OBJECT);
        /* Any response can fail, and we need to ensure that either some
         * response is sent or, if it is not possible, the transaction is
         * destroyed, so resources will not leak... */
        BACstacVerifyTSMCompletion ("BACstacSrvcError()", statX, hTSM);
        return HOOK_STATUS_OK;
    }
    /* Do our work here... */
    ....
    /* Send the response */
    statX = BACstacReadPropResponse (hTSM, pServiceInfo, &propContents);
    /* ensure that the TSM will not leak.. */
    BACstacVerifyTSMCompletion ("BACstacReadPropResponse()", statX, hTSM);
    return HOOK_STATUS_OK;
}
```

Registering and Using Hooks

A hook is an application routine that can handle a BACnet service request. A hook can decline to handle a service request by returning HOOK_STATUS_DEFAULT, in which case the BACstac default action will service the request. If a hook is to handle the service request completely, the value HOOK_STATUS_OK should be returned as a result of the hook.

All hook types supported by BACstac are listed below. These are included in the enumerated type BACSTAC_-HOOK_TYPE.

HOOK_WHO_IS,	HOOK_READ_PROP,
HOOK_READ_PROP_MULTIPLE,	HOOK_WHO_HAS,
HOOK_WRITE_PROP,	HOOK_WRITE_PROP_MULTIPLE,
HOOK_I_AM,	HOOK_SUBSCRIBE_COV,
HOOK_ACK_ALARM,	HOOK_I_HAVE,
HOOK_CONF_COV_NOTIFICATION,	HOOK_GET_ALARM_SUMMARY,
HOOK_TIME_SYNC,	HOOK_UNCONF_COV_NOTIFICATION,
HOOK_GET_ENROLLMENT_SUMMARY,	HOOK_UTC_TIME_SYNC,
HOOK_CONF_EVENT_NOTIF,	HOOK_DCC,
HOOK_ADD_ELEMENT,	HOOK_UNCONF_EVENT_NOTIF,
HOOK_CREATE_OBJECT,	HOOK_REMOVE_ELEMENT,
HOOK_CONF_PRIVATE_TRANSFER,	HOOK_DELETE_OBJECT,
HOOK_READ_RANGE,	HOOK_UNCONF_PRIVATE_TRANSFER,
HOOK_REINIT_DEV,	HOOK_READ_FILE,
HOOK_CONF_TEXT_MESSAGE,	HOOK_READ_PROP_CONDITIONAL
HOOK_WRITE_FILE,	HOOK_UNCONF_TEXT_MESSAGE,
HOOK_GET_EVENT_INFO	HOOK_SUBSCRIBE_COV_PROP
HOOK_LIFE_SAFETY_OPERATION	

For the following hook type defined in a header file:

```
typedef BACSTAC_HOOK_STATUS (BACstac_hook *BACSTAC_READ_FILE_HOOK_PROC) (
    BACSTAC_HTSM hTransaction,
    BACSTAC_ADDRESS *sourceAddress,
    BACSTAC_ADDRESS *destinationAddress,
    const BACSTAC_READ_FILE_INFO *pServiceInfo
);
```

A hook instance declaration looks as follows:

```
BACSTAC_HOOK_STATUS BACstac_hook MyReadFileHook(
     BACSTAC_HTSM hTransaction,
     BACSTAC_ADDRESS *sourceAddress,
     BACSTAC_ADDRESS *destinationAddress,
     const BACSTAC_READ_FILE_INFO *pServiceInfo);
```

All hooks have the same set of arguments, though the structure of the pServiceInfo argument depends on the service type. It is important to use the BACstac_hook modifier in hook definitions and prototypes, as in the above example.

The TSM handle (hTransaction in the above example) serves as a unique identifier of the transaction in case of confirmed service request hook. If your application defines an unconfirmed request hook (BACstac I-Am hook for example) you will get NULL for this parameter. In this case your application would not deal with TSMs.

sourceAddress is a pointer to BACSTAC_ADDRESS variable containing the address of device which has sent the request.

destinationAddress is a pointer to BACSTAC_ADDRESS variable containing the destination address as it was specified in the request.

The pointers passed into the hook can be used until corresponding transaction is completed so that you can use these pointers even outside the hook until TSM completion. This is important when defining *asynchronous* hooks (more about asynchronous hooks see in the Section called *Synchronous/asynchronous Request Handling*).

One of important hook's tasks is checking the service request arguments. For example, a Read-Property hook is given a pointer to a BACSTAC_READ_INFO data type variable that contains the fields ObjectID, ePropertyID, and nIndex. It is a good idea to check (for example) if such a property exists in this object.

In case a hook has returned HOOK_STATUS_OK the application should care for *TSM* completion (for details on TSM handling refer to the Section called *Transaction Life Cycle* in Chapter 8 of this Guide). The TSM should be completed returning either positive or negative response. If any problem with the service executing or the transaction completion appears (see 7.5) the TSM is completed via specialized routines. The choice of the concrete routine depends on the service type (please refer to the *Programmers' Reference* for details). Your application may complete the corresponding TSM even outside the hook but the period of time to pass should not be too long to avoid the transaction timeout.

A hook is registered using the BACstacSetHook() routine:

```
BACstacSetHook(HOOK_READ_PROP, (BACSTAC_HOOK_PROC) MyReadHook);
```

After this call all service requests of READ_PROPERTY type are executed using MyReadHook.

When a hook is registered by a Server application, the hook type argument may take *any* of the BACSTAC_HOOK_-TYPE values.

On the contrary, Client application can register only several hooks to handle notifications and broadcasts. The hook types that can be used are:

```
HOOK_CONF_COV_NOTIFICATION
HOOK_CONF_EVENT_NOTIF
HOOK_I_AM
HOOK_I_HAVE
HOOK_UNCONF_COV_NOTIFICATION
HOOK_UNCONF_EVENT_NOTIF
```

Hooks and Transaction Completion Routines

Completion routines are invoked only for confirmed requests. They are used to send a reply and free all internal resources associated with the transaction. The transaction handle, which is passed in as the first argument of a hook, is used as the first argument of any completion routine to identify the request to which the response is sent.

The other arguments of a service completion routine are the BACnet source and destination addresses and the service specific information. Here is an example of a hook that uses a positive acknowledgment routine to send the response. If the response cannot be sent, it destroys the transaction to free all internal resources associated with the transaction.

```
BACSTAC_HOOK_STATUS BACstac_hook ReadHook(
    BACSTAC_HTSM hTransaction,
    BACSTAC_ADDRESS *pSourceAddress,
    BACSTAC_ADDRESS *pDestAddress,
```

```
    const BACSTAC_READ_INFO *pServiceInfo)
{
    BACSTAC_PROPERTY_CONTENTS value;
    BACSTAC_STATUS status;
    BACSTAC_HDEVICE hDevice;
    const BACSTAC_OBJECT_ID *objectID;
    BACSTAC_HOBJECT hObject;
    BACSTAC_PROPERTY_ID propID;
    /* If it is a gateway then check that the destination address
     * is for an existing device */
#ifdef BACSTAC_GATEWAY
    hDevice = BACstacDeviceAddressToHandle(pDestAddr);
    if (hDevice == BACSTAC_INVALID_HANDLE)
    {
        BACstacDestroyTSM(hTransaction);
        return HOOK_STATUS_OK;
    }
#else
    hDevice = BACstacGetLocalDeviceHandle();
#endif
    /* Obtain the object handle and check its existence */
    objectID = BACstacGetReadInfoObjectIDPtr(pServiceInfo);
    hObject = BACstacFindObjectByID(hDevice, *objectID);
    if (hObject == BACSTAC_INVALID_HANDLE)
    {
        status = BACstacSrvcError(hTransaction,
                ERR_CLASS_OBJECT, ERR_CODE_UNKNOWN_OBJECT);
    }
    else
    {
        /* Check property existence */
        propID = BACstacGetReadInfoPropID(pServiceInfo)
        if (!BACstacDoesPropertyExist(propID))
        {
            status = BACstacSrvcError(hTransaction,
                    ERR_CLASS_PROPERTY, ERR_CODE_UNKNOWN_PROPERTY);
        }
        else
        {
            /* Read from hardware if it is Present_Value */
            if (propID == PROP_PRESENT_VALUE)
            {
                value = GetPropertyValueFromHardware(/*...*/);
                status = BACstacReadPropResponse(hTransaction, pServiceInfo, &value);
            }
            else /* Invoke the default action for all other properties */
               return HOOK_STATUS_DEFAULT;
        }
    }
    /* Check status & destroy the transaction if the response can't be sent */
    if (status != BACSTAC_STATUS_OK)
    {
        status = BACstacDestroyTSM(hTransaction);
        return HOOK_STATUS_OK;
    }
```

```
}
```

The BACstacDestroyTSM() routine is used in the example to destroy a transaction. More detailed information about destroying TSMs see in the Section called *Transaction Life Cycle* in Chapter 8 of this Guide.

The hook can use one of the four existing completion routines in case of negative reaction. Three of them correspond to the three types of BACnet messages: Error, Abort and Reject. These messages are sent on the corresponding situations discussed in AHSRAE standard. The fourth routine allows a Server application to ignore a service request by not sending any response. This might be necessary if a Server pretends to be off-line. If one of negative acknowledgment routines is used, the hook still returns HOOK_STATUS_OK since the service request is handled. For example, a ReadProperty hook can check some of the service request arguments using a BACstac building block routine and use an error completion routine if any arguments are incorrect.

Some hooks may also use completion routines that cannot serve all hooks but are applied in particular cases. For example the hooks for AddListElement and RemoveListElement services should use BACstacChangeListError() negative acknowledgment completion routine if an element could not be added to/removed from the list:

```
BACSTAC_HOOK_STATUS BACstac_hook AddElementHook(
    BACSTAC_HTSM hTSM,
    BACSTAC_ADDRESS *pSourceAddress,
    BACSTAC_ADDRESS *pDestAddress,
    const BACSTAC_CHANGE_LIST_INFO *pServiceInfo)
{
    BACSTAC_UTIL_STATUS status;
    BACSTAC_STATUS statX;
    BACSTAC_ERROR_CLASS errClass;
    BACSTAC_ERROR_CODE errCode;
    BACSTAC_ELEMENT_COUNT firstFailed;
    BACSTAC_HDEVICE hDevice;
    BACSTAC_HOBJECT hObject;
    /* define values of routine arguments */
    hDevice = BACstacGetLocalDeviceHandle();   /* This is a Server */
    hObject = BACstacFindObjectByID(
        hDevice,
        *BACstacGetReadInfoObjectIDPtr(pServiceInfo));
    /* this application routine does all the error checking */
    /* and changes the list of elements                     */
    status = BACstacAddElement(
        hObject,
        BACstacGetReadInfoPropID(pServiceInfo),
        BACstacGetReadInfoIndex(pServiceInfo),
        BACstacGetARLEHookInfoElementsPtr(pServiceInfo),
        &errClass,
        &errCode,
        &firstFailed);
    switch(status)
    {
        case UTIL_STATUS_OK:
                                        /* positive reply */
            statX = BACstacSrvcResponse (hTSM);
            /*                          Check status... */
            ...
            return HOOK_STATUS_OK;
        case UTIL_STATUS_BACNET_ERROR:
                                         /* negative reply */
```

```
                                    /* impossible to add element */
            statX = BACstacChangeListError (
                hTSM,
                ErrClass,
                errCode,
                firstFailed);
            /*                          Check status... */
            ...
            return HOOK_STATUS_OK;
        default:
                    /* invalid parameter or another technical error. */
                    /* let the default actions generate the error */
        return HOOK_STATUS_DEFAULT;
    }
}
```

If a hook handles an unconfirmed service request, it does not use any completion routine; all the operations on the TSM are completed by BACstac. Here is an example of the hook for the UTCTimeSynchronization service:

```
BACSTAC_HOOK_STATUS BACstac_hook MyTimeHook(
    BACSTAC_HTSM hTSM,
    BACSTAC_ADDRESS *pSourceAddress,
    BACSTAC_ADDRESS *pDestAddress,
    const BACSTAC_UTC_TIME_SYNC_INFO *pServiceInfo)
{
    BACSTAC_HDEVICE hDev;
    /*  Define Device handle */
    hDev = BACstacGetLocalDeviceHandle();
    /* This routine can be only used in Server applications */
    /* because they have (the only) local device */
    /*  This routine synchronizes date and time */
    SyncCurrentDateTime(hDev,pServiceInfo);
    /* no completion routine necessary for unconfirmed services */
    return HOOK_STATUS_OK;
}
```

The Who-Has and Who-Is services are unconfirmed, but they are expected to generate an I-Have or I-Am unconfirmed service as a response. For example:

```
BACSTAC_HOOK_STATUS BACstac_hook WhoHasHook(
    BACSTAC_HTSM hTSM,
    BACSTAC_ADDRESS *pSourceAddress,
    BACSTAC_ADDRESS *pDestinationAddress,
    const BACSTAC_WHO_HAS_HOOK_INFO  *pServiceInfo)
{
    BACSTAC_HOBJECT hObject;
    BACSTAC_ADDRESS address;
    BACSTAC_STATUS status;

    status = IsRequestValid(pServiceInfo); /* error checking */
    if (status == BACSTAC_STATUS_OK)
    {
        /* this routine fetches a local device and object handles */
        GetObjHandle(pServiceInfo, &hObject);
        BACstacInitAddrGlobalBroadcast(&address);
```

```
        BACstacIHave(NULL, hObject, &address);
    }
    status = BACstacSrvcError(hTSM,errClass,errCode);
    /* Check the status */ ...
    return HOOK_STATUS_OK;
}
```

Synchronous/asynchronous Request Handling

There are two ways of handling service requests in a hook. They are *synchronous* and asynchronous ones. When processing a request synchronously you perform all the needed actions inside the hook. In this case the other service requests are queued until the current request process is completed. In case of synchronously processed request the TSM completes inside the hook. In the other case all (or almost all) the work is performed outside the hook. What is most important, the TSM completion is also performed somewhere else in the program.

The hook itself stores the needed information about the request, returns HOOK_STATUS_OK as soon as possible and returns the control to the BACstac. Thus, the associated TSM and all the parameters that have been passed to the hook remain intact. So, you can use the pointers you have passed there in other parts of your application code. Nevertheless, you should care of completing the TSM in a short period of time because BACstac has a timer for each TSM. On the expiry of this timeout the TSM will be aborted by BACstac (see the Section called *Transaction Life Cycle* in Chapter 8 for details).

It is important not to make the synchronously handled hook too long either, all the more that the synchronous hook makes BACstac wait for the hook completion impeding other requests processing.

Consider an example of ReadFile request handling: a file which is sent as a result of this request can be large enough to timeout all the requests in the queue while receiving this file. Therefore it is a good idea to handle the ReadFile service request asynchronously:

```
BACSTAC_HOOK_STATUS AsyncReadFileHook (
    BACSTAC_HTSM hTSM,
    BACSTAC_ADDRESS *pSourceAddress,
    BACSTAC_ADDRESS *pDestAddress,
    const BACSTAC_READ_FILE_INFO *pServiceInfo)
{
/* Store the parameters of the request for further processing */
    EnterCriticalSection(&critSection);
/* Due to multithread work method all operations
   are performed within the critical section */
    MyAllocAndStoreRequestParameters(pServiceInfo);
    LeaveCriticalSection(&critSection);
//...
    return HOOK_STATUS_OK;
}
```

The file is sent during performing another part of the application program:

```
BACSTAC_BOOLEAN MyReadFileProcess (/*...*/)
{
    EnterCriticalSection(&critSection);
    LookUpRequestParameters(pServiceInfo);
    LeaveCriticalSection(&critSection);
    BACSTAC_READ_FILE_RESULT fileResult;
```

```
    BACSTAC_STATUS status;
    MyResultConstruct(pServiceInfo,&fileResult);
    /* Transmit the file */
    status = BACstacReadFileResponse(hTSM,&fileResult);
// Check status...
    return BACSTAC_TRUE;
}
```

The storing of request parameters (in a hook) and the look-up (in the application program) are performed concurrently from different threads. Therefore the synchronizing is needed. In the above example the Critical Section mechanism is used for synchronizing means.

Hooks in Gateway Application

A hook in a Gateway application must do a bit more work than the corresponding hook in a Server application, since a Gateway hook will receive service requests directed to the Gateway's virtual network. When a Gateway hook is invoked, it must use the destination address argument to look up the correct Device. A Server hook can only reference to the Server's single local Device. A Gateway hook must also behave correctly (which means ignoring the request in the corresponding case) if a service request is sent to a nonexistent virtual network MAC address. The following example illustrates how a Gateway hook could perform these tasks:

```
BACSTAC_HOOK_STATUS BACstac_hook MyReadHook(
    BACSTAC_HTSM hTransaction,
    BACSTAC_ADDRESS *pSourceAddr,
    BACSTAC_ADDRESS *pDestAddr,
    BACSTAC_READ_INFO *pServiceInfo)
{
    BACSTAC_HDEVICE hDev;
    /* which Device is the request for? */
    hDev = BACstacDeviceAddressToHandle(pDestAddr);
    /* non-existant Devices don't reply, eh? */
    if(hDev == BACSTAC_INVALID_HANDLE)
    {
        BACstacDestroyTSM(hTransaction);
        return HOOK_STATUS_OK;
    }
    ... /* rest of the hook is the same as in a Server */
}
```

Chapter 8. Bypassing BACstac Object Database

In this chapter we are going to describe a BACstac applications programming style different from the one reported in other parts of this Guide. Mostly in the Guide we described the BACstac set of functions and techniques named high level API (or API simply). Using this style of programming you make your applications work with BACstac Object Database, its device/object images, handles and default actions.

However, BACstac supports an alternative way of programming with its own set of routines named low-level ("raw") API or BACstac APIx. It is developed for the cases you cannot or don't want to use the database. Actually, every call to the APIx functions is simply an appeal to the real BACnet service request instance.

When programming to BACstac low-level API you do not turn to device handles. The application uses device address to communicate with a device. To handle one of a device's objects APIx routines use the device address and object identifier. You can bypass the Object Database completely using user-defined hooks, low-level routines and Asynchronous Callback Routines (ACR) included in low-level API. Besides, "raw" API widens the capabilities of your BACstac applications because there are several examples of BACnet services which don't have a high level implementation. Thus, the corresponding services are inaccessible from the high level API (consider the ReadPropertyConditional service for the instance — see *Programmers' Reference* for details). In case you want to take advantage of using such services you have to use APIx functions.

Low-Level API Programming Tools

Programming your applications without BACstac Object Database you use three function types. They are hooks, APIx routines and ACRs. For the description of hooks see Chapter 7 of this Guide.

All the routines of BACstac APIx have "Ex" suffix in their names. Thus, instead of BACstacReadProperty2() routine, you use BACstacReadPropertyEx2(), instead of BACstacSubscribeCOV(), use BACstacSubscribeCOVEx() etc (see *Programmers' Reference* for the detailed descriptions of concrete routines).

Using the "raw" API you can well enlarge your possibilities. When you work through BACstac database routines you should follow limitations concerning the functionality of your devices. The possibilities of three applications types — Servers, Clients and Gateways — are different. Using low-level API you can avoid a considerable part of these limitations. APIx allows you to create servers with some client features and vice versa. There are still several BACnet services which are accessible in server but always unavailable in client applications: ConfTextMessage, UnconfTextMessage, ConfCOVNotification, UnconfCOVNotification, ConfEventNotification, UnconfEventNotification, IAm, IHave. On the other side, the "raw" client routines are used in servers and gateways which want to exhibit its capability as a client. For example, using low-level API an area controller will monitor smaller controls as a client and perform calculations on their values and present the results as a server.

Besides, the BACstac Object Database has a number of its own limitations you should hold to. These limitations apply to the number of devices (for Clients and Gateway), number of objects in devices, size of property values. It is also important that the BACstac Object Database was not optimized for extremely large number of devices and objects (tens of thousands). In this case the database interchange is too slow. Programming with low-level API allows you to come out of this frame.

The low-level API allows your application sophisticated handling of BACnet transactions (see 7.5) and the possibility to process server requests asynchronously (see the Section called *Creating Client Applications Using "Raw" API* of this chapter).

Consider the BACstacReadPropertyEx2() routine as a raw API function example.

```
BACSTAC_STATUS BACstacReadPropertyEx2(
     const BACSTAC_ADDRESS *pSourceAddress,
     const BACSTAC_ADDRESS *pDestinationAddress,
     BACSTAC_READ_INFO *pReadInfo,
     BACSTAC_READ_PROP_ACR_PROC pfACR,
     const BACSTAC_APDU_PROPERTIES *pAPDUParams,
     BACSTAC_HTSM *phTransaction,
     BACSTAC_PROPERTY_CONTENTS *pContents,
     BACSTAC_ERROR *pError);
```

In this case the value of pDestinationAddress is used to specify destination device instead of Object handle which encapsulates the device address and object identifier. This is a regular BACSTAC_ADDRESS type input parameter you should always feed to the routine.

The address in pSourceAddress is a conditional parameter of the same type you can use to indicate the device that initiates the request. This parameter is used when a gateway application starts a request to specify a particular device on its network. Therefore, the SourceAddress is mandatory when your application is of Gateway type. Otherwise (in Server or Client device) the value of SourceAddress is not used. You can pass the real address to the routine or replace it with NULL constant.

The value of read property falls in pContents. This parameter is mandatory too. The value of pReadInfo parameter of BACSTAC_READ_INFO type contains the ReadProperty parameters (object identifier, property identifier and property array index).

Other parameters are optional and you can replace them by NULL constant in your source code. One of them is BACSTAC_ERROR *pError: you can pass the address where BACstac will put information of the error returned if the request caused a negative acknowledgment (in this case status returned by the routine is BACSTAC_STATUS_-BACNET_ERROR, BACSTAC_STATUS_BACNET_REJECT or BACSTAC_STATUS_BACNET_ABORT). Actually, this parameter serves here the same purpose it serves in other (high level API) functions.

The pfACR, APDU parameters and transaction handles will be considered in the corresponding sections below.

Creating Client Applications Using “Raw” API

Using low-level API you widen the possibilities of your applications. With “raw” API client applications get raw power. To enable this power you should use low-level API routines and other “raw” style features described in chapters 7.3-7.5. A client application created with “raw” API keeps all the possibilities of gathering and storing information about remote Devices and Objects that can be provided by the Object Database.

When you skip BACstac Object Database you have to invent your own method of storing the information about remote devices and objects. This method can base on the BACstac example code or be created anew.

Consider a client application designed to collect I-Am responses and to print out the values of device object properties. When using BACstac Object Database the client application receives the information about remote device and BACstac stores it in the Database and accesses it through device or object handles.

If your client application bypasses the Object Database you should create a *hook* for at least I-Am service (hooks are described in Chapter 7 of this Guide). This hook processes service information and stores device parameters in a user-defined form instead of BACstac Object Database. Your application might need I-Have hook as well.

Consider an example of a structure designed to keep the following information about remote device: its BACnet address and device instance number.

```
typedef struct tagBACSTAC_DEVICE_TAG{
    BACSTAC_ADDRESS deviceAddr;
    BACSTAC_INSTANCE_NUMBER deviceNum;
} BACSTAC_DEVICE_TAG;
```

Every time I-Am is received in your client application's hook you store the information about sending device. Each new device will be remembered with an instance of such a structure, which will serve as a "raw" equivalent of device handle used by BACstac Object Database. You can order them in any way suitable to store; for example, you can store them as a queue, a list or a stack. You can use an existing mechanism of storing a queue of arbitrary elements developed in the BACstac examples set.

Consider an example of the client application that uses BACSTAC_DEVICE_TAG structure to store images of remote devices. This client sends Who-Is request and receives a number of I-Am responses creating device tags using the structure given above for each device, which has sent an I-Am. These device tags are stored for the further processing. The client application prints out the properties of every tracked device and keeps the corresponding tag for the future work.

```
//
..
CriticalSection    myCriticalSection;
BACSTAC_HOOK_STATUS MyIAMHook (
    BACSTAC_HTSM hTSM,
    BACSTAC_ADDRESS *pSourceAddress,
    BACSTAC_ADDRESS *pDestAddress,
    const BACSTAC_I_AM_INFO *pServiceInfo)
{
    BACSTAC_INST_NUMBER instNum;
    BACSTAC_DEVICE_TAG *curDevice;
    instNum = BACstacGetIAmInfoDeviceNumber(pServiceInfo);
    curDevice = AllocAndInitDeviceTag(pSourceAddress, instNum);
    EnterCriticalSection(&myCriticalSection);
    MyStoreDeviceTag(curDevice);
    LeaveCriticalSection(&myCriticalSection);
    return HOOK_STATUS_OK;
}
//...
void MyPrintDevice (BACSTAC_INSTANCE_NUMBER myInstNum)
{
    BACSTAC_DEVICE_TAG    *deviceToPrint;
//...
//... Look up the needed device tag
    EnterCriticalSection (&myCriticalSection);
    deviceToPrint = LookUpDeviceDetails(myInstNum);
    LeaveCriticalSection (&myCriticalSection);
//...
    PrintDeviceProperties(deviceToPrint);
//...
    }
// ...
}
```

Since both the hook and MyPrintDevice() procedure access the application's set of device tags (which is to replace BACstac object database) and their calls are executed concurrently from multiple threads the application needs a synchronizing mechanism. Therefore the operations with the device tag set are performed within the critical section.

The same mechanism is discussed in the chapter describing ACR calls (see detailed description in the Section called *Asynchronous Callback Routines*.

Asynchronous Callback Routines

In the example above the parameter pfACR of type BACSTAC_READ_PROP_ACR_PROC is used to specify a pointer to the Asynchronous Callback Routine associated to the request. Asynchronous Callback Routine is an application-supplied function called asynchronously by BACstac to notify application about the request completion. This parameter is optional.

An ACR prototype for our ReadProperty example will look like the following:

```
typedef void (*BACSTAC_READ_PROP_ACR_PROC) (
    BACSTAC_HTSM hTransaction,
    const BACSTAC_ADDRESS *pSourceAddress,
    const BACSTAC_ADDRESS *pDestinationAddress,
    BACSTAC_STATUS status,
    BACSTAC_READ_INFO *pRequestInfo,
    BACSTAC_PROPERTY_CONTENTS *pContents,
    BACSTAC_ERROR *pError
);
```

If pfACR parameter value is NULL than the BACstac request will be accomplished *synchronously* (see *Programmers' Reference* for details). In the case the request was initiated successfully (in the other case the corresponding status value is returned), BACstacReadPropertyEx2() will not return until the request is complete (i.e. acknowledgment is received or final timeout expires). Upon that the control is returned to the caller so that the results can be processed. In this case BACstac no longer uses any application-supplied pointers (pError, pContents and pReadInfo in our example) so that the application can free them if it needs to. When raw API routine returns the transaction is destroyed, so the phTransaction parameter is not used, and if passed in, gets the BACSTAC_INVALID_HANDLE value.

Otherwise we consider the asynchronous request. When a pointer is passed for pfACR the request is initiated likely to the previous case but the routine will not wait for the request's completion. It will return to the caller, instead, and the return status will indicate success or failure of the request initiating. In case of success it will also return a handle to the transaction created which serves as the unique identifier of the request in progress through its completion.

Note that the ACR can be called by the BACstac in a concurrent thread in any moment not depending on the routine's work state. Hence the ACR can be called before the APIx routine return.

The values of the parameters having "out" semantics (the ones pContents and pError point to in case of BACstacReadPropertyEx2() routine) cannot be considered valid until the ACR is called. When the request is actually completed, the associated ACR is called. Final request's completion status and original raw API routine call "out" parameters (pContents and pError in our example) are passed in. From this point on these pointers are not used by the BACstac, so if application needs to free them, it can do it safely within the ACR.

The ACR for BACstacReadPropertyEx2() is an exception: pReadInfo parameter, having "in" semantics, is also passed to the ACR where it can be freed by the application if it needs to. In other cases the "in" parameters can be freed (if needed) after they had been passed to the low-level API routine.

Source and destination address values passed into a raw API routine follow a different policy. Address values (not pointers) are copied by the BACstac before the raw API routine returns. So, application can free these pointers as soon as the raw API routine returns, but ACR will get different pointers, ones that belong to the BACstac, and must not be freed by the application.

There is an important aspect of the BACstac asynchronous request design which applications can count on. After having initiated request, raw API routines do not compete with ACR-invocation mechanism for any resources, and are not synchronized with them in any way. This means that applications can call raw API routines having locked same mutex that is acquired in the ACR and release it afterwards without causing a deadlock.

Consider an application that issues arbitrary number of asynchronous ReadProperty requests simultaneously. Apparently, it needs to keep track of pending requests so that when they complete application could match acknowledgments with requests. To achieve this, application will typically maintain some data structure to store request contexts. When a new request is issued, its context will be stored. When ACR is called, application will look up request context in its storage, take whatever action is appropriate, and, most likely, remove request context from the storage. What follows is an example of such processing:

```
//
..
CriticalSection    requestContextLock;
MyStatus MyReadProperty (/*...*/)
{
//...
context = MyAllocAndInitContext (&readInfo);
EnterCriticalSection (&requestContextLock);
status = BACstacReadPropertyEx2 (
    NULL,            // source address is only used in gateways
    &destAddress,
    &readInfo,
    MyReadPropertyAcr,
    NULL,            // use default APDU parameters
    &hTransaction,
    &propContents    // out; will be passed into the ACR
    &netError,    // out, will be passed into the ACR
    NULL              // in, allocator
);
// Check status
// ...
MyStoreContext (context, hTransaction);
LeaveCriticalSection (&requestContextLock);
//...
}
void BACstac_callback MyReadPropertyAcr (
    BACSTAC_HTSM hTransaction,
    const BACSTAC_ADDRESS *pSourceAddress,
    const BACSTAC_ADDRESS *pDestinationAddress,
    BACSTAC_STATUS status,
    BACSTAC_READ_INFO *pRequestInfo,
    BACSTAC_PROPERTY_CONTENTS *pContents,
    BACSTAC_ERROR *pError)
{
// Check status
//...
// Lookup request context
    EnterCriticalSection (&requestContextLock);
    context = MyLookupAndRemoveContext (hTransaction);
    LeaveCriticalSection (&requestContextLock);
// Take whatever action we want to complete it
//...
// Clean up request context
```

```
    MyFreeContext (context);
// ...
}
```

hTransaction parameter passed in the ACR is the same handle that was returned by the original raw API routine call and can be used for the original call context identification. The transaction handle is valid until the ACR returns.

In the variable pointed to by the hTransaction parameter BACstac will pass a handle to the TSM created. This handle will identify the request until it is completed.

Since the request context storage is accessed from multiple threads (those that call BACstacReadPropertyEx2(), and those that call ACR(s)), access to the storage is guarded with some synchronization means. Notice, that BACstacReadPropertyEx2() is called within the critical section guarded by the lock: if the lock were acquired after the BACstacReadPropertyEx2() call, it would be possible for the ACR to be called within the time interval before the EnterCriticalSection () is called, and ACR would fail to find the match for the transaction it is passed in. Such application design is possible because BACstac guarantees that deadlock will not occur.

APDU Parameters

BACstac Application Layer Protocol Data Unit Properties (pAPDUParams in the example in the beginning of the chapter) value defines a set of transaction properties and limitations. The parameters included in the BACSTAC_APDU_PROPERTIES structure are the following:

- APDU maximum length;
- Segmentation support;
- Maximum number of segments;
- Segmentation window size;
- Segment timeout;
- APDU timeout;
- Number of APDU retries.

More detailed information on the parameters listed above see in the *Programmers' Reference*.

The parameters in this set are used to control the transaction and are stored in the TSM (see 7.5). The APDU properties value are invariable until the end of transaction. Generally speaking, every transaction can have its own set of APDU parameters.

Your application's possibility to control and alter APDU properties depends on the transaction type. If a *client* transaction (for the difference between client and *server* TSMs see 7.5) is created by your application with the help of low-level API routines, you can configure the APDU properties structure yourself using specialized access routines. Otherwise, the default values are involved. For the details on the APDU properties handling routines refer to the *Programmers' Reference*. In case the TSM is created via high level API the default values APDU properties are used.

Server TSMs are always created by the BACstac without application control; in this case BACstac uses default values of the current application's APDU properties. For APDU maximum length the smaller value of received request and current application's APDU properties is used. Analogous rule affects segmentation support (the weakest value used).

The *default* APDU parameters are taken by user application from Registry when the application starts. For the properties omitted in Registry default values are used (refer to the *Installation Guide* for details). After the application is started use BACstacSetDefaultAPDUProperties() routine (see *Programmers' Reference*) to alter the values of APDU

parameters at any time. Such changes only affect current application, i.e. each application has its own copy of default APDU properties. If another copy of application starts after the values in Registry have been changed, it will get updated values.

The application passes currently determined values of APDU properties to TSMs it starts. If the user changes APDU properties at runtime every TSM started afterwards will get new APDU property values.

Since the APDU parameters of the application can be altered, the values of the corresponding properties in the application's Device object should be adjusted to the current values enabled in the application. The BACstacSynchDeviceTSMParams() routine is used for this purpose (refer to the *Programmers' Reference* for detailed information). After the call to this routine, the values of the corresponding Device object properties are corrected in the BACstac Object Database. This is connected to the high-level programming style only.

There is a limitation on the total size of the APDU which is the number of segments (in case segmentation is on; or 1 in the other case) multiply by APDU maximum length. This limitation is dictated by BACstac IPC buffer size which is determined in Registry. For details refer to the *Installation Guide*. If this product exceeds the limitation BACstacSetDefaultAPDUProperties() returns BACSTAC_FALSE value. Analogous check is also performed during BACstac application Init() procedure. If such an error occurs Init() returns BACSTAC_STATUS_BAD_CONFIG.

Consider an example of client application that collects I-Am messages from remote devices and stores needed information in a way described in 7.2; later this application initiates WriteFile service to send some data to one of these remote devices. When initiating a WriteFile request this client uses stored information to adjust APDU properties of the request (APDU maximum length and Segmentation support) to the values suitable for the particular remote device.

```
typedef struct tagBACSTAC_RD_DETAILS{
    BACSTAC_ADDRESS deviceAddr;
    BACSTAC_INSTANCE_NUMBER deviceNum;
    BACSTAC_UNSIGNED deviceAPDU;
    BACSTAC_SEGMENTATION deviceSegm;
} BACSTAC_RD_DETAILS;
BACSTAC_HOOK_STATUS MyCacheAPDUProp(        // I-Am Hook
    BACSTAC_HTSM hTSM,
    BACSTAC_ADDRESS *pSourceAddress,
    BACSTAC_ADDRESS *pDestAddress,
    const BACSTAC_I_AM_INFO *pServiceInfo)
{
    BACSTAC_RD_DETAILS *pRemote;
    //...
    pRemote = AllocateAndInitDeviceDetails(
        pSourceAddress,
        BACstacGetIAmInfoDeviceNumber(pServiceInfo),
        BACstacGetIAmInfoMaxAPDU(pServiceInfo),
        BACstacGetIAmInfoSegmentation(pServiceInfo));
    EnterCriticalSection(&critSection);
    StoreDeviceDetails(pRemote);
    LeaveCriticalSection(&critSection);
    //...
    return HOOK_STATUS_OK;
}
BACSTAC_BOOLEAN MyWriteFileProc(
    BACSTAC_ADDRESS *pDestAddress,
    BACSTAC_WRITE_FILE_INFO *pWriteInfo,
    BACSTAC_WRITE_FILE_RESULT *pResult)
{
    //...
```

```
    BACSTAC_APDU_PROPERTIES curAPDUParams;
    BACSTAC_RD_DETAILS *curRemote;
    BACSTAC_ERROR error;
    BACSTAC_SEGMENTATION curSegm;
    BACSTAC_UINT curMaxAPDU;
    EnterCriticalSection(&critSection);
    curRemote = LookUpDeviceDetails(pDestAddress);
    LeaveCriticalSection(&critSection);
    BACstacGetDefaultAPDUProperties(&curAPDUParams);
    curSegm = BACstacGetAPDUPropSegmentation(&curAPDUParams);
    curMaxAPDU = BACstacGetAPDUPropInt(&curAPDUParams);
    curSegm = GetWeakerSegmentationValue(curRemote->deviceSegm,
        curSegm);
    curMaxAPDU = curMaxAPDU < curRemote->deviceAPDU ? curMaxAPDU:
        curRemote->deviceAPDU;
    BACstacSetAPDUPropInt(&curAPDUParams,
        BACSTAC_APDU_MAX_LENGTH,
        curMaxAPDU);
    BACstacSetAPDUPropSegmentation(&curAPDUParams, curSegm);
    status = BACstacWriteFileEx(
        NULL,                // Source address is not used
        &curRemote->deviceAddr,
        pWriteInfo
        NULL,                // Synchronous: no ACR used
        &curAPDUParams,
        NULL,                // TSM handle is not used
        pResult,
        &error);
    //... Check status...
    return BACSTAC_TRUE;
}
```

Transaction Life Cycle

The Transaction State Machine (TSM) handle is used as an argument to the completion routines to identify the transaction. The user can only access TSM via the “opaque” handle of BACSTAC_HTSM type.

Actually, a handle to TSM is used to identify a slot of TSM layer reserved for the concrete transaction. TSM maintains the transaction state. At the same time BACstac initiates a transaction it picks a free slot to store the TSM for this transaction. The number of slots available is determined by the application’s TSM *pool* size, which is read from the Registry. If Registry doesn’t contain this value it is set to 150 by default (see *Installation Guide* for details).

In both cases either user’s application sends or receives a request BACstac needs to create a TSM. The two types of TSM are distinguished: there are client TSMs and server TSMs.

Client TSM — if your application emits the request working so as a client

Server TSM — if your application replies to a request from a remote device working as a server

Accordingly, every application has two independent TSM pools: for server and client TSMs respectively. Both pools are constructed when the application is started. Being structured in such way, these two types of TSM are independent so that if your application has exhausted its limit of server TSMs it still can initiate transactions and vice versa. Sizes of the Client and Server TSM pool are specified in the Registry independently. Still, if you change the pool size every user application started later will follow these new specifications.

If high-level API functions are used in the application, the transactions (and their TSMs) are processed by BACstac entirely. In case of using “raw” API in your application, you are involved. When user application initiates a request BACstac creates a transaction and associated TSM; when BACstac receives a request from another BACnet device a TSM for this transaction is created.

The BACstac request can be synchronous or asynchronous (see ACRs description in 7.2). In both cases BACstac initiates a transaction. When processing synchronous request the transaction is destroyed as soon as the raw API routine’s returns so the user application can’t handle it (like in case of working with high level API). If a request is handled asynchronously its TSM handle is passed to the ACR and serves as an identifier of request. The transaction handle is valid until the ACR returns. After that, transaction is destroyed, and its TSM handle can be re-used by the BACstac for other transactions. Thus, using a "stale" handle (see below) can cause erroneous results.

When executing a confirmed request in a hook you should watch the transaction to be destroyed in time. If the transaction was completed normally, i.e. acknowledgment, either positive or negative, was sent, BACstac destroys it and frees the corresponding TSM. If any problem with the transaction completion appears (caused by lack of resources in BACstac, transaction timeout or some other reason) you should analyze the returned status to make a decision. You can destroy the transaction by calling BACstacDestroyTSM() routine or take some steps to try to overcome the problem.

When the server TSM is created BACstac sets up a timer. The default timeout value is 6 seconds. On this timeout expiry BACstac forces the transaction completion by sending Abort APDU with “other” reason to the initiating BACnet device, the TSM handle is marked as “aborted” and is not used or destroyed by BACstac. Application will discover this when it eventually tries to complete the transaction: BACstac will return the BACSTAC_STATUS_TRANSACTION_-ABORTED status code. In this case user application should call BACstacDestroyTSM() routine to destroy it.

Client TSM timer is also set on the TSM creation. There is some difference from server TSMs: client TSM timeout behavior is controlled by the APDU-timeout and APDU-retries APDU properties (see 7.3 for details). If the client TSM times out BACstac will return BACSTAC_STATUS_TIMEOUT status code.

You shouldn’t reuse the TSM handle value of the completed transaction. If this TSM is already used for another transaction BACstac won’t recognize it and erroneous result may be caused.

If your server application processes the request in a hook, there is a choice of two hook return values. They are HOOK_STATUS_DEFAULT and HOOK_STATUS_OK (see Chapter 7 of this Guide for the detailed description of hooks usage). In case of HOOK_STATUS_DEFAULT return BACstac passes the control to the default procedures for given request. So, BACstac processes transaction in the same way as while programming with BACstac object database. In case when the return value is “HOOK_STATUS_OK” then BACstac manages the TSM itself too: after request completed the transaction is destroyed and TSM handle is deleted. If the acknowledgment is not sent the TSM’s timeout expires (see above). In this case the user should call BACstacDestroyTSM().

Chapter 9. Notifications

Notifications allow servers to send data to interested clients as soon as an event occurs. BACnet supports two types of notifications, Events (or Alarms) and Change-Of-Value (COV). In addition, the notifications can be sent as confirmed or unconfirmed messages. The rules governing this group of services are described in Clause 13 of the BACnet standard.

COV Notifications

A COV Notification, either Confirmed or Unconfirmed, is sent when certain properties (usually present-value and status-flags) in an object change by a specified amount, as described in Clause 13.1 of the BACnet Standard.

A COV Notification is sent to all BACnet clients that have previously subscribed using the SubscribeCOV service. Subscriptions have lifetimes and can expire. Subscriptions indicate the Process ID to be used in the COV Notifications. Subscriptions can request Confirmed or Unconfirmed Notifications. The current COV subscription list is not visible in any object in the server (unlike Event subscriptions). For this reason, the source address of a subscription request is used as the destination address for the COV Notifications, meaning that clients must subscribe for themselves.

A system using COV Notifications can provide user interfaces and databases with current data with less network traffic than *polling* (using ReadProperty or ReadPropertyMultiple) at a fixed interval because messages are only sent when data has changed significantly. By using Unconfirmed Notifications, traffic can be reduced even further since the data is exchanged with one message, rather than two (a request and reply or notification and confirmation).

Event Notifications

An Event Notification, either Confirmed or Unconfirmed, is sent when an Event occurs, ie an important change in the state of an object in a BACnet server. The criteria used to detect events may be listed as properties in the object or in a related Event Enrollment object. The information on notification recipients may be recorded either in Event Enrollment object or in Notification Class object. Clients subscribe for Event notifications by adding themselves to the recipient-list property (using the AddListElement service) of the Notification Class object, or recipient property of the Event Enrollment object.

There are two main types of Events in BACnet. Intrinsic, where an object detects an Event using its own properties, and Algorithmic, where the Event is detected by comparing an object

to the properties in an Event Enrollment object. Both Confirmed and Unconfirmed Event Notification Services are used for both these types of Events. Intrinsic Events are “lightweight”, since they can be generated based on properties in the object itself. Algorithmic Events are more flexible and can be extended.

The Event Notification is sent to a specified process ID within the notification-client at a specified BACnet Address.

If the AckRequired parameter of Event Notification is set to BACSTAC_TRUE, then a secondary Event Notification, ACK_NOTIFICATION, shall be issued when an acknowledgment is received from an operator or supervisory application (in the form of an AcknowledgeAlarm service request).

Many parameters are included in the Event Notification Service using the BACSTAC_EVENT_NOTIF data structure. Some of those fields are only used for EVENT and ALARM types of notifications. All attributes of the data structures must be set according to the criteria described in Clause 13 of the BACnet Standard.

Confirmed and Unconfirmed Event notifications differ from one another only in that the former needs a confirmation that the notification has been received by the client application.

Chapter 10. Receiving Notifications

A client must tell the server that it is interested in a particular notification by *subscribing* with the server. Notifications are also identified by a Process ID, to allow notifications to be routed to multiple applications on a client system. A BACstac client must register a set of Process IDs with the local BACstac protocol stack which will identify the notifications that it wishes to receive.

Registering a Local Process ID

Prior to send any notification, the Process ID for which it is destined should be registered by the client application. The application can modify the set of Process IDs assigned to it by the BACstac protocol stack (initially an empty list) using the routines BACstacRegisterProcessID(), BACstacRegisterUniqueProcessID(), BACstacUnregisterProcessID(), BACstacGetProcessIDList(). The value 0 is reserved for future use for unconfirmed broadcasts.

A system could use a convention for assigning Process IDs, in which case a client application would be configured to know its set of Process IDs, and the BACstacRegisterProcessID() routine would provide a natural way for registration. Other applications may not care what value they use (say, for a short-term subscription), in which case they can request a unique value from the BACstac protocol stack, making use of BACstacRegisterUniqueProcessID. This is also the case of an application which cannot know the set of Process IDs of other applications running on the same host:

```
BACSTAC_UI32 processID;
/* register Process ID */
status = BACstacRegisterUniqueProcessID(&processID);
if (status != BACSTAC_STATUS_OK)
{
    /* error actions */
}
```

Next, the client must communicate the selected Process Ids to the server application. For Change of Value reporting, the SubscribeCOV service conveys the processID:

```
BACSTAC_HDEVICE hDev;       /* initialized elsewhere */
BACSTAC_SUBSCRIBE_COV_INFO subsriber;
BACSTAC_ERROR error;
BACSTAC_UI32 processIDx
/* Assign the registered value to the corresponding argument */
subscriber.processID = processIDx
...
BACstacSubscribeCOV (hDev, &subscriber, &error);
```

In the case of intrinsic reporting, the client application should write the registered Process ID into the appropriate Destination of the Recipient List of the Notification Class object used for reporting by the event-initiating object(s). (see the Section called *Subscribing for Event Notifications*). In the case of algorithmic reporting, the application should write the Process ID either into the Process

Identifier property of the Event Enrollment Object (when Notification Class object is not used) or into the appropriate Destination of the Recipient List of the Notification Class object (see the Section called *Subscribing to an Event Enrollment Object*).

Subscribing for COV Notifications

A client application must subscribe with a server to receive COV notifications. The subscription establishes a connection between the logical process within the COV Client which you want to be the notification recipient and the object within the COV Server whose properties you want to be monitored. It also determines the specific characteristics of the reporting mechanism established within the particular pair COV Client process - COV Server object. This subscription is implemented with use of BACstacSubscribeCOV() routine corresponding to SubscribeCOV service (see Clause 13.14 of the BACnet standard).

The set of properties whose values are reported by notifications depend on the type of the monitored object and is given in Table 13-1 of the BACnet standard. The criteria used for determining that a change of value has occurred depend solely on the Object type, while the parameters which may be required for applying these criteria are stored in the corresponding object database and have nothing to do with the COV subscription.

The arguments needed for BACstacSubscribeCOV() are mostly stored in the BACTAC_COV_CONTEXT structure. Prior to call BACstacSubscribeCOV(), you should initialize this structure or set its fields using the appropriate access routines. You should also obtain a device handle for the device, which contains the monitored object.

The following example illustrates the use of BACstacSubscribeCOV() for subscribing a COV Client for Confirmed notifications from all Analog-type objects of a device:

```
#include <
baccli.h>
#define MaxObjects 50
BACSTAC_HDEVICE hDev;
BACSTAC_UNSIGNED lifetime;
BACSTAC_UI32 processID;
BACSTAC_BOOLEAN issueConfNotif;
BACSTAC_STATUS status;
BACSTAC_OBJECT_ID objectID[MaxObjects];
BACSTAC_OBJECT_COUNT nObjects,n;
BACSTAC_ERROR err;
BACSTAC_OBJECT_TYPE objType;
BACSTAC_SUBSCRIBE_COV_INFO covContext;
/* previously registered the processID */
lifetime = 0;             /* infinite lifetime */
issueConfNotif = BACSTAC_TRUE;
/* get object list from device hDev initialized elsewhere */
status = BACstacReadObjectList(
    hDev,
    objectID,
    MaxObjects,
    &nObjects,
    &err);
if (status != BACSTAC_STATUS_OK)
{
    /* error actions */
}
/* look up objects for subscription */
for (n=0; n<nObjects && n<MaxObjects; n++)
{
    objType = BACstacGetObjectIDType(&objectID[n]);
    if (objType == OBJ_ANALOG_INPUT ||
        objType == OBJ_ANALOG_OUTPUT ||
        objType == OBJ_ANALOG_VALUE ||)
```

```
    {
        BACstacInitCOVSubscription(
            &covContext,
            processID,
            &objectID[n],
            issueConfNotif,
            lifetime);
        /* Subscribe for Change Of Value */
        status = BACstacSubscribeCOV(hDev,&covContext,&err);
        if (status != BACSTAC_STATUS_OK)
        {
            /* error actions */
        }
    }
}
```

Cancelling a COV Notification Subscription

A COV subscription can be cancelled by sending a SubscribeCOV request with different arguments. This is done by calling the same routine BACstacSubscribeCOV(), with the covContext structure initialized for cancellation. For that, the BACstacInitCOVCancellation() access routine should be called. To continue the above example, the following code may be provided for cancellation:

```
for (n=0; n<nObjects && n<MaxObjects; n++)
{
    objType = BACstacGetObjectIDType(&objectID[n]);
    BACstacInitCOVCancellation(
        &covContext,
        processID,
        &objectID[n]);
    /*  Cancel COV Subscription  */
    status = BACstacSubscribeCOV(hDev,&covContext,&err);
}
```

If a client requests the cancellation of a nonexistent COV subscription, the server will respond positively, as if the subscription had existed.

Receiving COV Notifications with a Hook

In general, the notification service requests may arrive as soon as the COV Client has subscribed for COV notifications. This implies the necessity of registering hooks for COV notifications prior to the call of BACstacSubscribeCOV(), as is the case in the above example, but after having initialized the Client:

```
BACSTAC_HOOK_STATUS BACstac_hook CovConfNotificationHook(
    BACSTAC_HTSM hTransaction,
    BACSTAC_ADDRESS *sourceAddress,
    BACSTAC_ADDRESS *destAddress,
    BACSTAC_COV_NOTIF_INFO *pServiceInfo);
BACstacSetHook(
    HOOK_CONF_COV_NOTIFICATION,
```

```
    (BACSTAC_HOOK_PROC) CovConfNotificationHook);
```

or, for an unconfirmed notification,

```
BACSTAC_HOOK_STATUS BACstac_hook CovUnconfNotificationHook(
    BACSTAC_HTSM hTransaction,
    BACSTAC_ADDRESS *sourceAddress,
    BACSTAC_ADDRESS *destAddress,
    BACSTAC_COV_NOTIF_INFO *pServiceInfo);
BACstacSetHook(
    HOOK_UNCONF_COV_NOTIFICATION,
    (BACSTAC_HOOK_PROC) UncovConfNotificationHook);
```

The two types of hooks correspond to the two types of COV notifications: HOOK_CONF_COV_NOTIFICATION and HOOK_UNCONF_COV_NOTIFICATION. Both these types, however, are given the service parameters with use of the same BACSTAC_COV_NOTIF_INFO structure.

No default actions are provided for either Confirmed or Unconfirmed Notification service requests. That means that the COV notification hook cannot decline to handle the notification service request by returning HOOK_STATUS_-DEFAULT.

In the case where a confirmed COV notification is received, the COV subscriber should indicate that the COV Notification service has succeeded, and the appropriate completion routine should be called by the confirmed COV notification hook. Typically, it will have the following structure:

```
BACSTAC_HOOK_STATUS BACstac_hook CovConfNotificationHook(
    BACSTAC_HTSM hTransaction,
    BACSTAC_ADDRESS *sourceAddress,
    BACSTAC_ADDRESS *destAddress,
    BACSTAC_COV_NOTIF_INFO *pServiceInfo)
{
    /* user-provided actions */
    BACstacSrvcResponse(hTransaction);
    return HOOK_STATUS_OK;
}
```

For an Unconfirmed COV Notification hook, BACstacSrvcResponse() is not called.

Subscribing for Event Notifications

Unlike COV subscriptions, which expire and must be reinitiated by the client, Event Notification subscriptions are stored in properties of Event Enrollment and Notification Class objects. The Event Notification subscriptions can often be configured permanently at system installation. Some controls will also allow dynamic subscription to Event Notifications.

In the case of intrinsic reporting, subscribing for event notifications reduces to adding an appropriate Destination to the Recipient List property of the Notification Class object referred to by the object with which the subscription is implemented.

In the case of algorithmic reporting, there are two ways of subscribing. The first is by writing the Client parameters into the Recipient property of the Event Enrollment object which monitors the given Object. The second is by adding a Destination to the Recipient List property of the Notification Class object referred to by the Event Enrollment object.

Subscribing to a Notification Class Object

To subscribe for Event Notifications in the case of intrinsic reporting:

- Discover the Notification Class of the event-initiating object
- Construct a subscription (the Destination data structure)
- Append your subscription to the Notification Class object's recipient-list

The Notification Class of the event initiating object is found by reading the object's notification-class property.

The subscription information includes the BACnet address or Device ID, and process ID of the application. It also includes the time interval each day in which notifications can be sent to this application, the type of events to send, and whether to use Confirmed or Unconfirmed Notifications. The Destination (see Table 12-24 of the BACnet Standard) is formed with use of Destination Access routines and stored in a Property Contents. Suppose you want to form a single Destination:

```
BACSTAC_DESTINATION destination;
BACSTAC_TIME fromTime, toTime;
BACSTAC_ADDRESS locAddress;
BACSTAC_UI32 processID;      /* registered previously, */
BACSTAC_RECIPIENT locRecipient;
/* set time to any time of the day */
BACstacInitTime(&fromTime, 0, 0, 0, 0);
BACstacInitTime(&toTime, 23, 59, 59, 99);
/* initialize locRecipient with Address */
BACstacGetLocalAddress(&locAddress);
BACstacInitRecipientAddress(&locRecipient, &locAddress);

/* initialize destination with preset values, allowing confirmed
   notifications */
BACstacInitDestination(
    &destination,
    &fromTime,
    &toTime,
    &locRecipient,
    processID,
    BACSTAC_TRUE);
/* set the flag Transitions for TO_OFFNORMAL */
BACstacSetBitStringBit(
    &destination.transitions,
    EVENT_TRANSITION_TO_OFFNORMAL,
    BACSTAC_TRUE);
/* set up propContents to use destination */
BACstacInitContentsToVar(
    &propContents,
    DATA_TYPE_DESTINATION,
    &destination,
    sizeof(destination));
```

The Notification Class recipient-list property should be updated using the BACnet AddListElement service. Using ReadProperty followed by WriteProperty is not recommended since another application could be trying to update the property at the same time.

```
BACSTAC_ERROR err;
BACSTAC_ELEMENT_COUNT firstFailed;
BACstacAddListElement(
    hObjNC,
    PROP_RECIPIENT_LIST,
    BACSTAC_VOID_INDEX,
    &propContents,
    &err,
    &firstFailed);
```

Cancelling a Notification Class Subscription

An Event Subscription is cancelled by using the BACnet RemoveListElement service. The list element value in the request is identical to the value in the initial AddListElement request.

```
BACSTAC_DESTINATION addList[MAX_DEST_NUM];
/* Initialize propContents so that it contains the second element
   in addList */
BACstacInitContentsToVar(
    &propContents,
    DATA_TYPE_DESTINATION,
    &addList[1],
    sizeof(BACSTAC_DESTINATION));
/* Remove the element from hObjNC object */
BACstacRemoveListElement(
    hObjNC,
    PROP_RECIPIENT_LIST,
    BACSTAC_VOID_INDEX,
    &propContents,
    &err,
    &firstFailed);
```

Subscribing to an Event Enrollment Object

Normally an Event Enrollment object will use a Notification Class object to handle its subscriptions. It is also possible for an Event Enrollment object to manage a single subscription by itself. The subscription information is contained in a series of properties of the Event Enrollment object, which can be updated using the BACnet WriteProperty service.

If an Event Enrollment object is managing a single subscription, its recipient property (see 12.11.13 of the BACnet Standard) will contain a Recipient (either a BACnet address or Device ID), otherwise it will be NULL.

An application can always try to write a value into the recipient property. The Event Enrollment object may choose to disallow the request if it doesn't want a previous subscription to be overwritten or if wants subscriptions to occur only via the Notification Class object. In addition, the process-identifier, priority and issue-confirmed-notifications properties must be set.

A direct subscription to an Event Enrollment object can be cancelled by writing NULL to the recipient property.

Requesting Alarm and Enrollment Summaries

The GetAlarmSummary() routine corresponds to the GetAlarmSummary service and is used by the Client application to obtain a list of so-called active alarms (see Clause 13.10 of the BACnet standard).

```
BACSTAC_HDEVICE hDev;      /* initialized elsewhere */
BACSTAC_ALARM_SUMMARY aSummaryBuffer[MAX_SUMMARIES];
BACSTAC_UINT nSummaryAvailable;
BACSTAC_ERROR err;
BACSTAC_STATUS status;
Status = GetAlarmSummary(
    hDev,
    aSummaryBuffer,
    MAX_SUMMARIES,
    &nSummaryAvailable,
    &err);
```

The GetEnrollmentSummary service is a more general query. Before calling BACstacGetEnrollmentSummary(), it is necessary to set up its filter argument. The simplest way to do that is to call BACstacInitEnrlFilter(). This routine will set the only mandatory BACSTAC_ACK_FILTER argument to the indicated value, leaving all other filters inactive (see 13.11.1.1.1 of the BACnet Standard).

```
BACSTAC_HDEVICE hDev;      /* initialized elsewhere */
BACSTAC_ENROLLMENT_FILTER filter;
BACSTAC_ENROLLMENT_SUMMARY aSummaryBuffer[MAX_SUMMARIES];
BACSTAC_UINT nSummaryAvailable;
BACSTAC_ERROR err;
BACSTAC_STATUS status;
/* initialize the filter so that the returned summary contains
   all event-initiating objects */
BACstacInitEnrlFilter(&filter, ACK_FILTER_ALL);
```

In actuality, the Summary of event-initiating Objects should be restricted to maximum degree. Among other reasons, that will speed up the summary return. This purpose is served by access routines for the filter argument. Any of these routines permits to restrict the Summary to such or such category of event-initiating Objects .

```
/* restrict the summary to objects whose Event State is not Normal */
BACstacSetEnrlFilterEventState(&filter, STATE_FILTER_ACTIVE);
status = BACstacGetEnrollmentSummary(
    hDev,
    &filter,
    aSummaryBuffer,
    MAX_SUMMARIES,
    &nSummaryAvailable,
    &err);
```

The returned Summary in that specific example represents a superset of a Summary that would have been obtained through the use of the GetAlarmSummary() routine: it will include any event-initiating Object whose Event State property is not NORMAL regardless of whether its Notify Type property is set to ALARM or not.

Chapter 11. Sending Notifications

A server must keep track of the list of interested clients (responding to subscription requests). A server must monitor its objects to detect conditions when it needs to send notifications. When those conditions occur, a server must assemble the data needed for the notification and generate the notifications.

Handling COV Subscriptions with a Hook

The list of current COV subscriptions is not visible in any object in the server, and is not maintained by the BACstac library. Maintaining that list is the responsibility of the server application.

The SubscribeCOV service is used to create a list of subscribers and maintain it. An application using the COV notification services must also register a hook of type BACSTAC_SUBSCRIBE_COV_HOOK_PROC:

```
main (void)
{
    ...
    BACSTAC_STATUS status;
    ...
    status = BACstacSetHook(HOOK_SUBSCRIBE_COV,
    (BACSTAC_HOOK_PROC) SubscribeCOVHook);
    ...
}
BACSTAC_HOOK_STATUS BACstac_hook SubscribeCOVHook(
    BACSTAC_HTSM hTSM,
    BACSTAC_ADDRESS *pSourceAddress,
    BACSTAC_ADDRESS *pDestAddress,
    const BACSTAC_SUBSCRIBE_COV_INFO *pServiceInfo)
{
    ...
```

The main purpose of such hook consists in adding new elements in the list of subscribers (these elements should incorporate information stored mostly in a COV context variable) maintained for each monitored object and updating the already existing elements. The basic actions you should take with that aim depend on whether the SubscribeCOV request conveys subscription or cancellation and may include the following steps:

For the case of subscription, the hook should first check to see if the object is already being monitored. Second, the list of subscribers for the given object should be searched for the subscriber with the BACnet address and processID corresponding to the request source Address and processID. If no such subscriber is found, an element representing information on the subscriber should be added to the subscriber list. If, on the contrary, the subscriber is already present, then the request should be considered re-subscription and its COV context renewed.

For the case of cancellation, the steps are similar. If the object is already being monitored and a subscriber with matching source Address and processID is found, this subscription should be cancelled. If no matching subscription is found, no actions are taken. In both cases, however, the Result(+) must be returned (see BACnet standard, p.241). This implies the necessity to call the completion routine BACstacSrvcResponse(). The end of the hook thus takes the following form

```
    ...
    status = BACstacSrvcResponse(hTSM);
    return HOOK_STATUS_OK;
}
```

Since no default actions are provided for that type of hook, HOOK_STATUS_OK

is the only value it may return.

Detecting COV Events

A COV Notification, either Confirmed or Unconfirmed, is sent when the present-value property of a local Object changes by a predetermined amount, or the status-flags property changes. The COV Notification typically reports the values of those two properties. The Object types, for which COV reporting is supported, the criteria for reporting and the reported properties are enumerated in Table 13-1 of the BACnet standard.

In addition, COV Notifications are only sent to clients with current subscriptions. When a value changes, the server should look at the subscriptions for that object. COV Notifications should be sent to permanent subscriptions. For the case of temporary subscription, the new time remaining for the subscription expiry must be calculated, since it is included in the Notification. If the subscription has expired, it can be removed from the subscription list.

Generating COV Notifications

The COV Notification is sent to a specified process ID within the notification-client at a specified BACnet Address. A Confirmed COV notification differs from an Unconfirmed COV notification only in that it needs a confirmation that the notification has been received by the client application.

You can send a COV Notification to a particular recipient by calling BACstacConfCOVNotification() or BACstacUnconfCOVNotification(). To select the appropriate routine and set its arguments with appropriate values, you need the information stored in the list of subscribers. Apart from COV context, this information should include a destination BACnet address (passed by BACstac to the corresponding SubscribeCOV hook as a Source Address, see above) and, unless the subscription is permanent, the time of subscription. The following example, assuming the Object to be of an analog type, illustrates how you may set the BACstacConfCOVNotification() and BACstacUnconfCOVNotification() arguments.

```
#include <
bacsrv.h>
BACSTAC_HDEVICE hLocDev;
BACSTAC_HOBJECT hObj;              /* the object which changed */
/*  this information is from the subscription list for the object */
BACSTAC_BOOLEAN issueConfirmedNotifications;
BACSTAC_UI32 processID
BACSTAC_UNSIGNED timeRemaining;
BACSTAC_ADDRESS destAddress;
/*  these values are used to generate the notification */
#define NumPropValues 2            /* for objects of an analog type */
BACSTAC_PROPERTY_VALUE propValue[NumPropValues];
BACSTAC_PROPERTY_CONTENTS propContents[NumPropValues];
BACSTAC_BIT_STRING statusFlags;
BACSTAC_REAL realv;
BACSTAC_ERROR err;
BACSTAC_STATUS status;
BACSTAC_BOOLEAN boolStatus;
/* set propValue array with values to be reported */
/* set the first propValue element with Present_Value */
BACstacInitContentsToVar(
```

```
    propContents,
    DATA_TYPE_REAL,
    &realv,
    sizeof(realv) );
BACstacRetrievePropertyInstance2(
    hObj,
    PROP_PRESENT_VALUE,
    BACSTAC_VOID_INDEX,
    propContents,
    NULL);
BACstacInitPropValue(
    propValue,
    PROP_PRESENT_VALUE,
    propContents);
/* Similarly, set the second element with Status_Flags */
...
if (issueConfirmedNotifications)
{
    /* send Confirmed... */
    status = BACstacConfCOVNotification(
        hLocDev,
        hObj,
        processID,
        timeRemaining,
        propValue,
        NumPropValues,
        &destAddress,
        &err);
    if (status != BACSTAC_STATUS_OK)
    {
        /* error actions */
    }
}
else
{
    /* ...or Unconfirmed notification */
    status = BACstacUnconfCOVNotification(
        hLocDev,
        hObj,
        processID,
        timeRemaining,
        propValue,
        NumPropValues,
        &destAddress);
    if (status != BACSTAC_STATUS_OK)
    {
        /* error actions */
    }
}
```

Detecting Intrinsic Events

Intrinsic Events are detected by monitoring Objects for changes that would cause an event transition, as described in Clause 13.2 of the BACnet Standard. A server should monitor the Present Value of all objects which support intrinsic reporting. A record should be kept of the occurrence and time of changes which are to be reported (as stated in the BACnet standard).

If the monitoring procedure discovers that the Present Value does not correspond to the Event State of the Object (for instance, the Present Value of an Analog Object is no longer between Low Limit and High Limit of the given Object while the Event State equals STATE_NORMAL value) and that such situation persists for more than the Time Delay fixed for the event-initiating object, than the following steps should be taken:

- determine the new Event State in accordance with the new Present Value and the old Event State,
- determine from the Event Enable property whether that new Event State is to be generated and the transition to it reported,
- only for Analog objects, determine from the Limit Enable property whether HighLimit or LowLimit offnormal/normal events are to be generated and the transition to them reported.
- corresponding flag in the Event Enable property (and in Limit Enable, for Analog objects) enables the event transition to the given event, the following steps should be taken:
- change Event State of the object to that new value (see Subchapter 4.4, *Property Instance*),
- clear intrinsic context flags,
- change the IN_ALARM flag of the Status Flags property of the object in accordance with the new Event State,
- if the corresponding flag in the Ack Required property of the Notification Class object referenced by the given object is set, clear the respective flag in objects' Acked Transitions property corresponding to the new event to indicate that an acknowledgment is expected, and vice versa,
- determine the Event Type in accordance with the Object type and with Table 13-2 of the BACnet Standard,
- generate the notification as described below.

Generating Intrinsic Event Notifications

Once an Intrinsic Event has been detected, the server will use information from the event-initiating object and a Notification Class object to generate the notifications.

The corresponding Notification Class Object is identified by an unsigned notification-class property of the event-initiating Object. A server application should have some way of finding the corresponding Notification Class object from the unsigned value, like an index. The unsigned value may also correspond to the instance number appearing in the object-identifier property, but this is not required.

The Notification Class Object contains the recipient-list (see p.207-209 of the BACnet Standard) to be used for the notification. For each destination found in that list, the server must:

- Check whether the destination is interested in transitions to the new Event State at the current time and day of week,
- Get all the notification parameters (mostly read them from the event-initiating and Notification Class objects),
- Send the notification.

Check the Time and Date:

For each destination, the validity of the current date and time should be checked versus the corresponding fields of that destination.

```
#define MaxSizeDestinationList 100
BACSTAC_DESTINATION   destinationList[MaxSizeDestinationList];
BACSTAC_ELEMENT_COUNT destNumber; /* num of el. in destination list */
BACSTAC_UNSIGNED      n;
BACSTAC_TIME          *pFromTime;
BACSTAC_TIME          *pToTime;
BACSTAC_TIME          timeNow;
for (n = 0; n < destNumber; n++)
{
    /* read From Time and To Time in Destination */
    pFromTime = BACstacGetDestinationFromTimePtr(&destinationList[n]);
    pToTime = BACstacGetDestinationToTimePtr(&destinationList[n]);
    GetCurrentTime( &timeNow);
    /* compare timeNow with fromTime and toTime */
}
```

After the above step, both fromTime and toTime values should be compared to the current time timeNow. Analogously, the current date should be got and tested against validDays in Destination.

Check the Event Type:

The bit in the Transitions field of the Destination corresponding to the new Event State is checked to see if the client is interested in the event (there are three bits: EVENT_TRANSITION_TO_NORMAL, EVENT_TRANSITION_-TO_OFFNORMAL, EVENT_TRANSITION_TO_FAULT):

```
BACSTAC_DESTINATION destination
BACSTAC_BIT_STRING  *pTransitionBits;
BACSTAC_EVENT_TRANSITION_BITS eventTransitionBit;
/* set eventTransitionBit with the value corresponding to the new
   Event State */
...
pTransitionBits = BACstacGetDestinationTransBitsPtr(&destination);
/* if the appropriate bit is cleared... */
if (BACstacTestBitStringBit(pTransitionBits, eventTransitionBit)
    == BACSTAC_FALSE)
{
   /* ...no notification required */
}
```

Issue Confirmed Notifications:

To choose the appropriate Notification service (Confirmed or Unconfirmed) for each recipient, you should obtain the Destination IssueConfirmedNotification Parameter:

```
BACSTAC_DESTINATION destination;
BACSTAC_BOOLEAN issueConfNotif;
issueConfNotif = BACstacGetDestinationConfFlag(&destination);
```

If issueConfNotif is TRUE, you should send the notification with use of BACstacConfEventNotification2() API routine, otherwise BACstacUnconfEventNotification2() is used. These routines have the same arguments except that BACstacUnconfEventNotification2() receives no buffer for Error.

Recipient Address:

The address of the Recipient (i.e. the destination device to receive the notification) is obtained from the Destination:

```
BACSTAC_DESTINATION destination;
BACSTAC_RECIPIENT *pRecipient;
BACSTAC_RECIPIENT_TYPE recipientType;
pRecipient = BACstacGetDestinationRecipientPtr(&destination);
recipientType = BACstacGetRecipientTag(pRecipient);
```

Further actions depend on whether the Recipient field of Destination contains the Address or Object Identifier of the Recipient:

```
BACSTAC_ADDRESS *pDestAddress;
BACSTAC_OBJECT_ID *pDeviceObjectID;
BACSTAC_INST_NUMBER recipientInstNumber;
/* if Recipient contains Address */
if (recipientType == RECIPIENT_ADDRESS)
{
    pDestAddress = BACstacGetRecipientAddressPtr( pRecipient);
    ...
}
/* if Recipient contains Device Object Identifier */
if (recipientType == RECIPIENT_DEVICE)
{
    pDeviceObjectID = BACstacGetRecipientDevicePtr(pRecipient);
    recipientInstNumber = BACstacGetObjectIDInstNum(pDeviceObjectID);
    /* get recipient Address by its Instance Number */
    BACstacResolveDeviceID(recipientInstNumber, pDestAddress, NULL);
    ...
}
```

If the recipient contains a Device ID, the server needs to discover the destination device address. There are a few different ways to do that. For the sake of simplicity, the above example uses synchronous resolution, so the calling thread is blocked until BACstacResolveDeviceID returns. To learn more about different methods to resolve Device ID, please refer to Chapter 6.

Process ID:

The handle of a process within the recipient device which will receive the notification is obtained from the corresponding Destination:

```
BACstacSetEventNotifProcessID(&eventNotif,
     BACstacGetDestinationProcessID(&destination);
```

Time Stamp:

In the case where the device issuing the service request has no clock, a sequence number should be assigned to this parameter (see Section 13.8.1.5 of the BACnet Standard):

```
BACSTAC_UNSIGNED seqNumberCurrent; /* initialized elsewhere */
BACstacInitTimeStampSeqNumber(
    BACstacGetEventNotifTimeStampPtr(&eventNotif),
    seqNumberCurrent);
seqNumberCurrent++;
```

If the device has a clock, the timeStamp parameter should be initialized with the current time:

```
BACSTAC_TIME timeNow;
BACstacGetCurrentTime(&timeNow);    /* see Examples\Utils\bactime.h */
BACstacInitTimeStampTime(BACstacGetEventNotifTimeStampPtr(&eventNotif), &timeNow);
```

or, if you want to set timeStamp with DateTime:

```
BACSTAC_DATE_TIME dateTimeNow;
GetCurrentDateTime(&dateTimeNow);
BACstacInitTimeStampDateTime(
    BACstacGetEventNotifTimeStampPtr(&eventNotif),
    &dateTimeNow);
```

Notification Class:

By the time where you form a notification, the value for the eventNotif.notifClass parameter must be already read from the Notification Class property of the event-initiating object (see above).

Priority:

The priority parameter should be got from the Priority property of the Notification Class object referenced to by the event-initiating object. First, the Priority array

```
BACSTAC_UNSIGNED priorities[3];
```

should be read with use of BACstacInitContentsToArray() and BACstacRetrievePropertyInstance2() routines. Second, the array element corresponding to the new Event State should be chosen: (alternatively, one can pass appropriate array index to BACstacRetrievePropertyInstance2)

```
BACSTAC_EVENT_TRANSITION_BITS eventTransitionBit;
/* set to the appropriate Event State elsewhere */
BACstacSetEventNotifPriority(
    &eventNotif,
    (BACSTAC_UI8) priorities[eventTransitionBit]);
```

Event Type:

By the time of forming the notification, the value for eventNotif.eventType is already determined (see above).

Notify Type:

This parameter may be simply read from the event-initiating object into a Property Contents pointing to eventNotif.notifyType.

Acknowledge Required:

The ackRequired parameter should be obtained from the Ack_Required property of the Notification Class object referenced to by the event-initiating object. After having read this property into the ackRequiredFlags buffer, you should test its flag corresponding to the new Event State:

```
BACSTAC_BIT_STRING        ackRequiredFlags;
BACSTAC_EVENT_TRANSITION_BITS eventTransitionBit;
/* set to the appropriate Event State elsewhere */
BACstacSetEventNotifAckRequired(
    &eventNotif,
    BACstacTestBitStringBit(
        &ackRequiredFlags,
        eventTransitionBit));
```

Depending on the result of this testing, the same flag in the Acked_Transitions property of the event-initiating object should be cleared or set meaning that an acknowledgment is expected or not, respectively, in response to the notification being formed:

```
BACSTAC_BIT_STRING ackedTransFlags;
/* read the Acked_Transitions property into ackedTransFlags buffer*/
...
/* change the flag corresponding to eventTransitionBit */
if (BACstacGetEventNotifAckRequired(&eventNotif) == BACSTAC_TRUE)
{
    BACstacSetBitStringBit(&ackedTransFlags, eventTransitionBit, BACSTAC_FALSE);
}
else
{
    BACstacSetBitStringBit(&ackedTransFlags, eventTransitionBit, BACSTAC_TRUE);
}
/* write the ackedTransFlags value into Acked_Transitions property*/
...
```

From-State, To-State:

The fromState field represents the previous Event State of the event-initiating object which is read by the monitoring procedure at the very beginning. The toState field represents the new Event State of the event-initiating object.

Event Values:

The eventValues field represents a set of Notification Parameters specific to the particular event type and are listed in Table 13-3 of the BACnet Standard. These parameters are mostly read from the event-initiating object. Examples are listed below for each type of event.

CHANGE_OF_STATE event type

Present Value corresponding to the new Event State is read previously (see above) by the monitoring procedure, the Status Flags is changed in accordance with the new Event State.

The New State is set to Present Value differently for Binary and Multistate objects. For Binary Input and Binary Value objects, whose Present Value is of type BinaryPV:

```
BACSTAC_PROPERTY_STATES newState;
BACSTAC_BIT_STRING statusFlags;
BACSTAC_BINARY_PV binaryPresentValue;
...
BACstacSetPropStatesBinary(&newState, binaryPresentValue);
```

For Multistate Input objects, whose Present Value is of type Unsigned:

```
BACSTAC_UNSIGNED unsignedPresentValue;
    ...
BACstacSetPropStatesUnsigned(&newState, unsignedPresentValue);
```

For both:

```
BACstacInitNPCOState(
    BACstacGetEventNotifEventValuesPtr(&eventNotif),
    &newState,
    &statusFlags);
```

OUT_OF_RANGE event type

Present Value, Deadband, and High/Low Limit (whichever is exceeded) are read previously (see above) by the monitoring procedure, the Status Flags is changed in accordance with the new Event State.

```
BACSTAC_REAL presentValue;
BACSTAC_REAL deadband;
BACSTAC_BIT_STRING statusFlags;
BACSTAC_REAL exceededLimit;
BACstacInitNPOutOfRange(
    BACstacGetEventNotifEventValuesPtr(&eventNotif),
    presentValue,
    &statusFlags,
    deadband,
    exceededLimit);
```

COMMAND_FAILURE event type

Both the Command Value (ie. commandable Present Value) and Feedback Value are read previously by the monitoring procedure, the Status Flags is changed in accordance with the new Event State. For both Binary and Multistate Output objects these values should be passed to the initialization routine:

```
BACSTAC_PROPERTY_CONTENTS cmdValue;
BACSTAC_BIT_STRING statusFlags;
BACSTAC_PROPERTY_CONTENTS feedbackValue;
BACstacInitNPCmdFailure(
    BACstacGetEventNotifEventValuesPtr(&eventNotif),
```

```
        &cmdValue,
        &statusFlags,
        &feedbackValue);
```

FLOATING_LIMIT event type

Referenced Value, Setpoint Value, and Error Limit (ie Controlled Variable Value, Setpoint, and Error Limit properties of the Loop object, respectively) are read previously by the monitoring procedure, the Status Flags is changed in accordance with the new Event State. These arguments should be passed into the Notification Parameters initialization routine:

```
BACSTAC_REAL refValue;
BACSTAC_BIT_STRING statusFlags;
BACSTAC_REAL setpointValue;
BACSTAC_REAL errorLimit;
BACstacInitNPFloatingLimit(
    BACstacGetEventNotifEventValuesPtr(&eventNotif),
    refValue,
    &statusFlags,
    setpointValue,
    errorLimit);
```

Send the Event notification:

After monitoring all the objects which support intrinsic reporting, obtaining parameters for all required notifications, you may proceed to sending the notification.

To avoid blocking the server while sending the notifications, you may want to assemble and queue the notifications before sending them. For any ALARM or EVENT notification, if an acknowledgment is required the transition, you may want to keep track of some notification parameters. These parameters can be used for sending the acknowledgment notification and will be used to identify the Acknowledge Alarm request whose receipt is notified by comparing these parameters with the arguments passed into the hook which controls sending this notification (see SubChapter 8.8).

```
BACSTAC_HOBJECT hObj;     /* initialized elsewhere */
BACSTAC_ERROR err;         /* optional */
if (issueConfNotif)
{
    /* Send Confirmed Notification */
    status = BACstacConfEventNotification2(
        hObj,
        &eventNotif,
        &destAddress,
        &err);
}
else
{
    /* Send Unconfirmed Notification */
    status = BACstacUnconfEventNotification2(
        hObj,
        &eventNotif,
        &destAddress);
}
```

Detecting Algorithmic Events

General:

Algorithmic reporting differs from intrinsic reporting in that the reported event is not a simple change of value detected for a certain time. Rather, it is defined with use of some predetermined algorithm specified by the Event Enrollment Object which belongs to the same device as the event-initiating object. That Event Enrollment Object contains information needed for sending notifications and in a sense keeps the same place as Notification Class object for intrinsic reporting. In some cases, however, a part of this information may be also provided by Notification Class object which thus complements the Event Enrollment Object (see Clause 13.3 of the BACnet Standard). Unlike Notification Class Object which may be used by multiple objects of the same device, the Event Enrollment Object refers to the only object which is identified by its Object Property Reference property.

Unlike intrinsic reporting, where an event consists in a change in Present Value property, the algorithmic reporting may be related to various properties. For code examples, see Subchapter “Detecting Intrinsic Events”.

Differences in Generating Events Between Algorithmic and Intrinsic Reporting:

Intrinsic and algorithmic reporting have much in common. The differences arise mostly from the mechanism by which events are generated and from the structure of Event Enrollment and Notification Class objects. Below is the list of differences in generating events between algorithmic and intrinsic reporting for various event types.

The event is generated under the following circumstances:

CHANGE_OF_STATE

intrinsic:

The Present Value changes to a new state

algorithmic:

The referenced property changes to one of the values contained in the List_of_Values parameter

OUT_OF_RANGE

intrinsic:

The Present_Value goes outside the range between Low_Limit and High_Limit or returns within the range between High_Limit - Deadband and Low_Limit + Deadband (see Table 13-2 of the BACnet Standard), where High_Limit, Low_Limit, and Deadband are properties of the event-initiating object

algorithmic:

The referenced property goes outside the range between Low_Limit and High_Limit or returns within the range between High_Limit - Deadband and Low_Limit + Deadband (see 13.3.6 of the BACnet Standard), where High_Limit, Low_Limit, and Deadband are parameters of the Event Enrollment object.

COMMAND_FAILURE

intrinsic:

The commanded property differs from the Value indicated by the Feedback Value of the same object

algorithmic:

The referenced property differs from the current Value of the property indicated by the Feedback_Property_Reference parameter. That property may belong to an object differing from the Reference Object

FLOATING_LIMIT

intrinsic:

|Setpoint - Controlled_Variable_Value| > Error_Limit. All quantities in this condition are properties of the Loop object.

algorithmic:

|Setpoint_Reference - Object_Property_Reference| > High_Diff_Limit (Low_Diff_Limit).

Here Setpoint_Reference and High_Diff_Limit (Low_Diff_Limit) are Event Enrollment Object parameters, Object_Property_Reference is a property of the same object.

Detecting a New Event State and Implementing the Transition:

First, the occurance of event transition must be detected; a procedure should monitor the referenced property. If there is more than one such property in the device, then they should all be monitored, perhaps as a list of monitored properties.

It is generally desirable to store the information having relation to changes in some special data structure. In the first place, such a structure should contain a BACSTAC_OBJ_PROP_REF variable referencing the monitored property. This structure helps the application to detect a transition occurrence. This requires measuring the time elapsed since the change of property. Also, flags may be included for particular event types (e.g., for the OUT_OF_RANGE event type, a flag indicating which of two limits - low or high - is exceeded). A variable that would store the old value of the changed property may also be useful as a component of this structure.

All monitored properties may be inserted in a queue of such structures. The criteria for event occurrences are described in Clause 13.3 of the BACnet Standard. The transition may be implemented when the monitoring procedure discovers the occurrence of one of the following conditions:

For all event types except CHANGE_OF_VALUE:

The referenced property does not correspond to the current value of the Event State property of the Event Enrollment Object, or

For CHANGE_OF_VALUE event type, referenced property of type real

the new referenced property value differs from the old one by more than the Referenced_Property_Increment parameter - for the case of type real, or

For CHANGE_OF_VALUE event type, referenced property of type bitstring

(the new referenced property value) XOR (the old referenced property value) AND (Bitmask) parameter has at least one bit not equal to zero.

If the monitoring procedure reveals that such situation persists longer than the time indicated by the Time_Delay parameter, the following steps should be taken:

- Determine the new Event State in accordance with the new Present Value and the Event Type property of the Event Enrollement Object,
- Change Event State of the Event Enrollement Object to the determined value except for the CHANGE_OF_VALUE event type,
- Clear necessary flags (flagNewState in our example) except maybe for the CHANGE_OF_VALUE event type,

- Determine from the Event Enable property of the Event Enrollement Object whether the transition should be reported.

Generating Algorithmic Event Notifications

(For code examples, see Subchapter 8.5 “Generating Intrinsic Event Notifications”).

If the transition is to be reported, the following steps should first be taken:

- if acknowledgment is required for the determined transition, clear the corresponding bit in Acked Transitions property to indicate that an acknowledgment is expected. Whether acknowledgment is required or not is determined from the AckRequired property of the Notification Class object. If Notification Class object is not used (see below), it is up to user to decode whether acknowledgment is required or not.
- for CHANGE_OF_VALUE event type only, set the old referenced property value equal to the new referenced property value,

The exact method used to create the event notification depends on whether or not the Notification Class property is used.

If the Recipient property has a non-NULL value, meaning that the Notification Class property shall not be used (see 12.11.13 of the BACnet Standard), the recipient is to be obtained from the Recipient property.

If the Recipient property has a NULL value, meaning that the Notification Class property shall be u sed, then the Notification Class object indicated by the Notification Class property should be found in the local device and the following steps should be taken:

- read the recipient list of the obtained Notification Class object (see p.206-207 of the BACnet Standard),
- for each destination found in that list check whether the destination enables transitions to the new Event State at the current time and day of week.

Generating algorithmic notifications is very similar to using intrinsic reporting. Some differences in obtaining notification parameters arise, mostly from the use of Event Enrollment object.

Issue Confirmed Notifications:

This value, indicating whether Confirmed or Unconfirmed Notification service should be used, is obtained from the Destination IssueConfirmedNotification Parameter

Recipient Address:

A pointer to the address of the destination device (the Recipient) should be obtained from the Destination.

Process ID:

The handle of a process within the recipient device, which will receive the notification, is obtained from the corresponding Destination.

Time Stamp:

In the case where the device issuing the service request has no clock, a sequence number should be assigned to this parameter (see p.236 of the BACnet Standard). If, on the contrary, the device has a clock, the current time should be obtained and the timeStamp parameter should be initialized with that time.

Notification Class:

By the time where the notification is being formed, the value for the eventNotif.notifClass parameter is already read from the Notification Class property of the Event Enrollment object.

Priority:

The priority parameter should be read from the Priority property of the Notification Class object referenced by the Event Enrollment object.

Event Type:

This parameter is read from the Event Type property of the Event Enrollment object.

Notify Type:

This parameter is read from the Notify Type property of the Event Enrollment object.

Acknowledge Required:

The ackRequired parameter is read from the Ack Required property of the Notification Class object referenced by the Event Enrollment object. First, its flag corresponding to the new Event State should be tested. Then, depending on the result of this test, the same flag in the Acked_Transitions property of the Event Enrollment object should be cleared or set indicating that an acknowledgment is expected or not, respectively.

From-State, To-State:

Represents the previous and new Event States of the Event Enrollment object which was read by the monitoring procedure at the very beginning.

Event Values:

The eventValues field represents a set of Notification Parameters specific to particular event type and listed in Table 13-4 of the BACnet Standard. These parameters are read from the reference object.

CHANGE_OF_BITSTRING event type

The value of the bitstring property corresponding to the new Event State has been read previously by the monitoring procedure (which is necessary for determining the transition occurence), the Status Flags has been changed in accordance with the new Event State. These arguments should be passed into the Notification Parameters initialization routine:

```
BACSTAC_BIT_STRING newBitString;
BACSTAC_BIT_STRING statusFlags;
    ...
/* Initialize a CHANGE OF BITSTRING set of values which will be
   conveyed, as an Event Values parameter, by an Event Notification
   Service. */
BACstacInitNPCOBitString(
    BACstacGetEventNotifEventValuesPtr(&eventNotif),
    &newBitString,
    &statusFlags);
```

CHANGE_OF_STATE event type

Present Value corresponding to the new Event State has been read previously (see above) by the monitoring procedure, the Status Flags has been changed in accordance with the new Event State.

The New State notification parameter is set to Present Value differently for Binary and Multistate objects. For Binary Input and Binary Value objects, whose Present Value is of type BinaryPV:

```
BACSTAC_EVENT_NOTIF eventNotif;
BACSTAC_PROPERTY_STATES newState;
BACSTAC_BIT_STRING statusFlags;
BACSTAC_BINARY_PV binaryPresentValue;
...
/* Set the New_State notification parameter for the
   CHANGE OF STATE event type */
BACstacSetPropStatesBinary(&newState, binaryPresentValue);
```

For Multistate Input objects, whose Present Value is of type Unsigned:

```
BACSTAC_UNSIGNED unsignedPresentValue;
...
/* Set the New State notification parameter for the
   CHANGE OF STATE event type */
BACstacSetPropStatesUnsigned(&newState, unsignedPresentValue);
```

For both:

```
/* Initialize a CHANGE OF STATE set of values which will be conveyed,
   as an Event Values parameter, by an Event Notification Service. */
BACstacInitNPCOState(
    BACstacGetEventNotifEventValuesPtr(&eventNotif),
    &newState,
    &statusFlags);
```

CHANGE_OF_VALUE event type

The referenced property corresponding to the new Event State has been read previously by the monitoring procedure, the Status Flags has been changed in accordance with the new Event State. Different initialization routines are used to set the Notification Parameters to those arguments for the cases where the referenced property is of bitstring or real type:

a) For real type:

```
BACSTAC_REAL newValue;
BACSTAC_BIT_STRING statusFlags;
/* Initialize a CHANGE OF VALUE set of values for real type */
BACstacInitNPCOVIncrement (
    BACstacGetEventNotifEventValuesPtr(&eventNotif),
    &newValue,
    &statusFlags);
```

b) For bitstring type:

```
BACSTAC_BIT_STRING newBitString;
BACSTAC_BIT_STRING statusFlags;
/* Initialize a CHANGE OF STATE set of values for bitstring type */
BACstacInitNPCOVBitString (
    BACstacGetEventNotifEventValuesPtr(&eventNotif),
    &newBitString,
    &statusFlags);
```

COMMAND_FAILURE event type

Both the Command Value (i.e commandable Present Value) and Feedback Value have been read previously by the monitoring procedure, the Status Flags has been changed in accordance with the new Event State. For both Binary and Multistate Output objects these values should be passed to the initialization routine:

```
BACSTAC_PROPERTY_CONTENTS cmdValue;
BACSTAC_BIT_STRING statusFlags;
BACSTAC_PROPERTY_CONTENTS feedbackValue;
/* Initialize a COMMAND_FAILURE set of values */
BACstacInitNPCmdFailure(
    BACstacGetEventNotifEventValuesPtr(&eventNotif),
    &cmdValue,
    &statusFlags,
    &feedbackValue);
```

FLOATING_LIMIT event type

Referenced Value, Setpoint Value, and Error Limit (ie Controlled Variable Value, Setpoint, and Error Limit properties of the Loop object, respectively) have been read previously by the monitoring procedure, the Status Flags has been changed in accordance with the new Event State. These arguments should be passed into the Notification Parameters initialization routine:

```
BACSTAC_REAL refValue;
BACSTAC_BIT_STRING statusFlags;
BACSTAC_REAL setpointValue;
BACSTAC_REAL errorLimit;
/* Initialize a FLOATING_LIMIT set of values */
BACstacInitNPFloatingLimit(
    BACstacGetEventNotifEventValuesPtr(&eventNotif),
    refValue,
    &statusFlags,
```

```
        setpointValue,
        errorLimit);
```

OUT_OF_RANGE event type

Present Value, Deadband, and High/Low Limit (whichever is exceeded) have been read previously (see above) by the monitoring procedure, the Status Flags has been changed in accordance with the new Event State.

```
BACSTAC_REAL presentValue;
BACSTAC_REAL deadband;
BACSTAC_BIT_STRING statusFlags;
BACSTAC_REAL exceededLimit;
/* Initialize an OUT_OF_RANGE set of values */
BACstacInitNPOutOfRange(
    BACstacGetEventNotifEventValuesPtr(&eventNotif),
    presentValue,
    &statusFlags,
    deadband,
    exceededLimit);
```

Handling Event Acknowledgments with a Hook

In the case where an Acknowledgment is required (i.e. the flag of the corresponding event transition is set in the Ack-Requiered property of the Notification Class object) and where that acknowledgment proves successful, in response to that acknowledgment the Server application should issue a notification called Acknowledgment Notification. This notification

has its Notify type parameter set to NOTIFY_TYPE_ACK_NOTIFICATION.

In order to control sending such notification, the Server application, after having initialized the Server, should register a hook of type HOOK_ACK_ALARM:

```
BACSTAC_STATUS status;
status = BACstacSetHook(
    HOOK_ACK_ALARM,
    (BACSTAC_HOOK_PROC) AcknowledgeAlarmHook);
```

That hook should take the following actions:

First, compare the Service Information and Source Address arguments of the hook to the corresponding fields of the previously stored ACK_NOTIFICATION_INFO structure (the Section called *Generating Intrinsic Event Notifications*)

Next, if Service Information and/or Source Address counterparts do not match ACK_NOTIFICATION_INFO structure, then generate Result(-):

```
BACSTAC_HTSM hTSM,
BACSTAC_ERROR_CLASS errClass,
BACSTAC_ERROR_CODE errCode
status = BACstacSrvcError(hTSM, errClass, errCode);
/* indicate that the hook job is completed successfully */
return HOOK_STATUS_OK;
```

Then, optain the parameters for Ack Alarm notification to be put into event notification queue (the Section called *Generating Intrinsic Event Notifications*):

- Process Identifier and Event Object Identifier: from Service Information argument of the BACSTAC_ACK_-ALARM_HOOK_PROC hook;
- Notification Service Destination Address: from Source Address argument of the BACSTAC_ACK_ALARM_-HOOK_PROC hook;
- issueConfNotif of ACK_NOTIFICATION_INFO structure was previously stored should be used to send Confirmed or Unconfirmed notification;
- the Notify Type parameter should be set to NOTIFY_TYPE_ACK_NOTIFICATION;
- the timeStamp parameter should be initialized.

Set the Acked_Transitions flag corresponding to the new Event Stat, indicating that the acknowledgment is no longer expected:

```
BACSTAC_EVENT_TRANSITION_BITS eventTransitionBit;
GetTransitionBitFromEventState(
    pServiceInfo->eventState,
    &eventTransitionBit);
/* set Acked_Transitions flag of the event-initiating object */
SetAckedTransitionsFlag(
    pServiceInfo->eventObjectID,
    eventTransitionBit);
```

To complete the hook indicate that the Acknowledgment Alarm service request has succeeded:

```
/* call the response completion routine generating Result(+) */
status = BACstacSrvcResponse(hTSM);
/* indicate that the hook job is completed successfully */
return HOOK_STATUS_OK;
```

Handling Summary Requests with a Hook

The task of the GetEnrollmentSummary hook falls into the following parts:

- Look up objects which: first, are event-initiating, second, fit the filter argument of the GetEnrollmentSummary request (see 13.11 of the BACnet Standard).
- Form an Enrollment Summary List
- Generate the reply.

From the standpoint of technique of sending Notifications the event-initiating Objects fall into three categories:

1) Objects which support intrinsic reporting (see Clause 13.2 of the BACnet Standard).

These Objects are of one of the following types: Analog Input, Analog Output, Analog Value, Binary Input, Binary Output, Binary Value, Loop, Multistate Input, Multistate Output. To send Notifications, these Objects always refer to a Notification Class (NC) object. Here they are briefly referred to as input/output objects.

2) Event Enrollment (EE) objects which monitor input/output Objects and use, for sending Notifications, a NC object in just the same way as input/output Objects do.

3) The same Event Enrollment objects which, in contrast to 2), do not use any NC object. Rather, for sending Notifications they use the information contained in their own properties.

Both 2) and 3) support algorithmic reporting (see Clause 13.3 of the BACnet Standard).

To establish whether an Object is to be included in a Summary list, it should pass subsequently through a series of filters, the first of them being destined to determine if that Object supports intrinsic/algorithmic reporting at all. Only those Objects, which pass all the filters up to the end, are to be included in the Summary list.

The Summary formation and sending in itself is a simple matter. All you have to do to convey the Summary list is to call a special completion routine.

The GetEnrollmentSummary hook should be registered (see 4.13) by your application:

```
main (void)
{  BACSTAC_STATUS status;
   ...
status = BACstacSetHook(HOOK_GET_ENROLLMENT_SUMMARY,
(BACSTAC_HOOK_PROC) GetEnrlSummaryHook);
   ...
};
```

Suppose you have to search a single Device for Objects to be included in the Summary List. You then have to obtain a full list of Objects in that Device and test them in a loop.

In the loop your first obtain an Object handle for the given Object, and determine whether it supports intrinsic reporting. For that, find out whether it has a Property (Acked Transitions

in our example whose presence may serve as a criterion for supporting intrinsic reporting.

Retain only those objects which do have such a Property.

Determine which of the three categories does the object belong to. With that purpose, first determine the Object Type, then, if it is an Event Enrollment Object, determine if its

Recipient property has a NULL value. If so, the Object does use a NC object for sending notifications (see 12.11.13 of the BACnet Standard).

Acknowledgment is the only mandatory filter component, so the filter mask should not be tested for Acknowledgment. For all other filter components, check the corresponding bit in the filter mask (see BACstac *Programmers' Reference*, I, 3.5). If the bit is set, the property must be tested against the appropriate filter component pServiceInfo -> <Filter Component>.

Table 1 presents Properties and Objects - their owners which are to be tested for each specific Filter component in order to determine whether the Object is to be included in the set restricted by this Filter component.

Table 1. Properties to be tested for Object filtering

Filter Component	**Input/Output** objects (always use **NC**)	**Event Enrollment** (objects using **NC**)	**Event Enrollment** objects (**NOT** using **NC**)

Acknowledgment	Acked Transitions **Input/Output**	Acked Transitions **Event Enrollment**	Acked Transitions **Event Enrollment**
Enrollment	Recipient List **Notification Class**	Recipient List **Notification Class**	Recipient Process Identifier **Event Enrollment**
Event State	Event State **Input/Output**	Event State **Event Enrollment**	Event State **Event Enrollment**
Event Type	Object Type **Input/Output**	Event Type **Event Enrollment**	Event Type **Event Enrollment**
(Notification) Priority	Priority **Notification Class**	Priority **Notification Class**	Priority **Event Enrollment**
Notification Class	Notification Class **Input/Output**	Recipient Notification Class **Event Enrollment**	Recipient **Event Enrollment**

For each Object which has passed through all filter components, a Summary should be created (see Table 13-11 of the BACnet Standard). Last, the completion routine BACstacEnrollmentSummResponse() should be called to convey the Summary list to the Client. If the list is empty, a NULL pointer should be passed in this routine.

Chapter 12. Transferring Files

Clause 14 of Standard defines the set of File Access Services used to access and manipulate files contained in BACnet devices.

Every file that is accessible by these services shall have a corresponding File object in the BACnet device. This File object is used to identify the particular file, and provide access to file parameters such as file's total size, creation date, and type.

The AtomicReadFile Service is used by a client BACnet-user to perform an open-read-close operation on the contents of the specified file. The file may be accessed as records or as a stream of octets, depending of type of file AccessMethod.

The AtomicWriteFile Service is used by a client BACnet-user to perform an open-write-close operation of an OCTET STRING into a specified position or a List of OCTET STRINGs into a specified group of records in a file. The file may be accessed as records or as a stream of octets, depending of type of file AccessMethod.

Typically, files will not be able to be transferred in a single BACnet request (the transfer size being limited by max-segments * APDU-size). Client applications must break file transfers into a series of AtomicRead/WriteFile requests. Client applications may also need to truncate a file by writing to the file length property of the File object. Server applications must be prepared to handle requests on arbitrary blocks or records in its File object.

Handling File Transfer Request in a Server

A server application must provide Hook procedures to process the services and map BACnet requests to operations on real files. The hooks will be invoked when a Client Application issues read or write request.

```
main (void)
{
    BACSTAC_STATUS status;
    ...
    status = BACstacSetHook(
        HOOK_READ_FILE,
        (BACSTAC_HOOK_PROC) ReadHook);
    ...
}
BACSTAC_HOOK_STATUS BACstac_hook ReadHook(
    BACSTAC_HTSM hTSM,
    BACSTAC_ADDRESS *pSourceAddress,
    BACSTAC_ADDRESS *pDestAddress,
    const BACSTAC_READ_FILE_INFO *pServiceInfo)
{
    ...
```

First two parameters are pointers to the BACnet Addresses of the request initiator and request destination. Last parameter is a pointer to structure, containing fileID (Object Type and Instance number), file access type (STREAM or RECORD), and depending of access type defines start position and number of octets (or start of record position and number of records) that subroutine must read or write.

If the requested start position is invalid, an error should be returned. If the requested octet or record count exceeds the end of the file, the result is still returned with the End-Of-File flag set. If the requested octet or record count would exceed the maximum possible reply size (limited by APDU-length * max-segments), the BACstacReadFileResponse()

routine will exit with an error. An application should anticipate this and check the request against the maximum possible reply size before allocating buffers and reading files. The application should return then an error.

```
BACSTAC_INT start, count; /* initialized earlier, and checked for validity */
BACSTAC_READ_FILE_RESULT readFileResult;
BACSTAC_OCTET_STRING buffer;
BACSTAC_BYTE aBuffer[MAX_BLOCK_SIZE];
FILE *pFile; /* opened earlier */
count = fread(aBuffer, sizeof(BACSTAC_BYTE), count, pFile);
BACstacInitOctetString(&buffer,aBuffer,sizeof(aBuffer),count);
BACstacInitRFResultStream(&readFileResult,start,&buffer);
BACstacSetRFResultEOF(&readFileResult, feof(pFile));
BACstacReadFileResponse(hTSM,&readFileResult);
```

Transferring Files to a Client

The following example breaks a file download task into series of AtomicRead requests, so each response size will not exceed the maximum size that can be sent by the server and be accepted by the client. The read file is written to a local file using standard C library functions (fopen, fwrite, fclose).

```
#define BLOCKSIZE 2048
BACSTAC_INTEGER start = 0;
BACSTAC_VALUE_SIZE size;
BACSTAC_BYTE buffer[BLOCKSIZE];
BACSTAC_READ_FILE_SPEC readFileSpec;
BACSTAC_READ_FILE_RESULT readResult;
BACSTAC_ERROR error;
BACSTAC_STATUS status;
BACSTAC_HOBJECT hFile; /* initialized previously */
char *localFileName;  /* initialized previously */
FILE *pFile;
pFile = fopen(localFileName, "wb"); /* binary mode to suppress '\n' */
if (pFile == NULL)
{
    /* error handling */
}
BACstacInitRFResultStreamBuf(&readResult, buffer, BLOCKSIZE);
/* read blocks until the last block is read, which is determined by
   * the EOF flag in the server response. */
do
{
    /* read a block */
    BACstacInitRFSpecStream(&readFileSpec, start, BLOCKSIZE);
    status = BACstacReadFile2(
        hFile,
        &readFileSpec,
        &readResult,
        &error,
        NULL);
    if (status != BACSTAC_STATUS_OK) break;
    /* write the block to the local file */
    blockCount = BACstacGetRFResultStreamDataSize(&readResult);
```

```
    if (size != fwrite(buffer, 1, size, pFile))
    {
        /* not enough disk space */
        status = BACSTAC_STATUS_VAL_OUT_OF_SPACE;
        break;
    }
    /* Advance the read position */
    if (size > (BACSTAC_VALUE_SIZE)(BACSTAC_INTEGER_MAX - start))
    {
        /* file too long */
    }
    start += (BACSTAC_INTEGER)size;
} while (!BACstacGetRFResultEOF(&readResult));
fclose(pFile);
```