Pico C/C++ SDK

Libraries and tools for
C/C++ development on
RP2040 microcontrollers

Raspberry Pi Trading Ltd

Colophon

build-date: 2021-01-12
build-version: 4c83a5a-clean

Legal Disclaimer Notice

TECHNICAL AND RELIABILITY DATA FOR RASPBERRY PI PRODUCTS (INCLUDING DATASHEETS) AS MODIFIED FROM TIME TO TIME ("RESOURCES") ARE PROVIDED BY RASPBERRY PI (TRADING) LTD ("RPTL) "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW IN NO EVENT SHALL RPTL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THE RESOURCES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

RPTL reserves the right to make any enhancements, improvements, corrections or any other modifications to the RESOURCES or any products described in them at any time and without further notice.

The RESOURCES are intended for skilled users with suitable levels of design knowledge. Users are solely responsible for their selection and use of the RESOURCES and any application of the products described in them. User agrees to indemnify and hold RPTL harmless against all liabilities, costs, damages or other losses arising out of their use of the RESOURCES.

RPTL grants users permission to use the RESOURCES solely in conjunction with the Raspberry Pi products. All other use of the RESOURCES is prohibited. No licence is granted to any other RPTL or other third party intellectual property right.

HIGH RISK ACTIVITIES. Raspberry Pi products are not designed, manufactured or intended for use in hazardous environments requiring fail safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, weapons systems or safety-critical applications (including life support systems and other medical devices), in which the failure of the products could lead directly to death, personal injury or severe physical or environmental damage ("High Risk Activities"). RPTL specifically disclaims any express or implied warranty of fitness for High Risk Activities and accepts no liability for use or inclusions of Raspberry Pi products in High Risk Activities.

Raspberry Pi products are provided subject to RPTL's Standard Terms. RPTL's provision of the RESOURCES does not expand or otherwise modify RPTL's Standard Terms including but not limited to the disclaimers and warranties expressed in them.

Table of Contents

Chapter 1. About the Pico SDK

1.1. Introduction

The Pico SDK (Software Development Kit) provides the headers, libraries and build system necessary to write programs for the RP2040 based devices such as the Raspberry Pi Pico in C, C++ or assembly language.

The Pico SDK is designed to provide an API and programming environment that is familiar both to non-embedded C developers and embedded C developers alike. A single program runs on the device at a time with a conventional `main()` method. Standard C/C++ libraries are supported along with APIs for accessing the RP2040's hardware, including DMA, IRQs, and the wide variety fixed function peripherals and PIO (Programmable IO).

Additionally the Pico SDK provides higher level libraries for dealing with timers, USB, synchronization and multi-core programming, along with additional high level functionality built using PIO such as audio.

The Pico SDK can be used to build anything from simple applications, full fledged runtime environments such as MicroPython, to low level software such as the RP2040's on chip bootrom itself.

1.2. Design Overview

The RP2040 is a powerful chip, however it is an embedded environment, so both RAM, and program space are at premium. Additionally the trade offs between performance and other factors (e.g. edge case error handling, runtime vs compile time configuration) are necessarily much more visible to the developer than they might be on other higher level platforms.

The intention within the SDK has been for features to just work out of the box, with sensible defaults, but also to give the developer as much control and power as possible (if they want it) to fine tune every aspect of the application they are building and the libraries used.

The following sections highlight some of the design of the Pico SDK:

NOTE

> Some parts of this overview are quite technical, trying to get across the how and the why. Don't be discourage if it doesn't all make sense at first, you can always come back later when you want to dig in deeper.

1.2.1. The Build System

The Pico SDK uses CMake to manage the build. CMake is widely supported by IDEs (Integrated Development Environments), and allows a simple specification of the build (via `CMakeLists.txt` files), from which CMake can generate a build system (for use `make`, `ninja` or other build tools) customized for the platform and by any configuration variables the developer chooses (see [pico-sdk-cmake-vars] for a list of conifuration variables). TO DO: GRAHAM/LIAM: link pico-sdk-cmake-vars to the appropriate content, after it's been added

Apart from being a widely used build system for C/C++ development, CMake is fundamental to the way the Pico SDK is structured, and how applications are configured and built.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/pre_release/blink/CMakeLists.txt Lines 1 - 9

```
add_executable(blink
        blink.c
        )

```

```
# Pull in our pico_stdlib which pulls in commonly used features
target_link_libraries(blink pico_stdlib)

# create map/bin/hex file etc.
pico_add_extra_outputs(blink)
```

Looking here at the blink example, we are defining a new executable `blink` with a single source file `blink.c`, with a single dependency `pico_stdlib`. We also are using a Pico SDK provided function `pico_add_extra_outputs` to ask additional files to be produced beyond the executable itself (`.uf2`, `.hex`, `.bin`, `.map`, `.dis`).

The Pico SDK builds an executable which is `bare metal`, i.e. it includes the entirety of the code needed to run on the device (other than floating point and other optimized code contained in the bootrom within the RP2040).

`pico_stdlib` is an *INTERFACE* library and provides all of the rest of the code and configuration needed to compile and link the `blink` application. You will notice if you do a build of blink (https://github.com/raspberrypi/pico-examples/tree/pre_release/blink/blink.c) that in addition to the single `blink.c` file, the inclusion of `pico_stdlib` causes about 40 other source files to be compiled to flesh out the blink application such that it can be run on the RP2040 device.

1.2.1.1. INTERFACE Libraries are key!

Within CMake, an *INTERFACE* library is a way of aggregating:

- Source files
- Include paths
- Compiler definitions (visible to code as `#defines`)
- Compile and link options
- Dependencies (on other *INTERFACE* libraries)

When building an executable with the Pico SDK, all of the code for the executable (including any SDK libraries) is (re)compiled for that executable from source.

NOTE

This does not include the C/C++ standard libraries provided by the compiler.

In the example `CMakeLists.txt` we declare a dependency on the (*INTERFACE*) library `pico_stdlib`. This *INTERFACE* library itself depends on other *INTERFACE* libraries (`pico_runtime`, `hardware_gpio`, `hardware_uart` and others). `pico_stdlib` provides all the basic functionality needed to get a simple application running and toggling GPIOs and printing to a UART.

The *INTERFACE* libraries form a tree of dependencies with each contributing source files, include paths, compiler definitions and compile/link options to the build. To build the application each source file is compiled with the combined include paths, compiler definitions and options and linked into an executable according to the provided link options.

Building in this way allows your build configuration to specify customized settings for those libraries (e.g. Pin or DMA channel settings) at compile time allowing for faster and smaller binaries in addition of course to the ability to remove support for unwanted features from your executable entirely.

INTERFACE libraries also make it easy to aggregate functionality into readily consumable chunks (such as `pico_stdlib`).

IMPORTANT

Pico SDK functionality is grouped into separate *INTERFACE* libraries, and each *INTERFACE* library contributes the code *and* include paths for that library. Therefore you must declare a dependency on the *INTERFACE* library you need directly (or indirectly thru another *INTERFACE* library) for the header files to be found during compilation of your source file (or for code completion in your IDE).

NOTE

As all libraries within the SDK are *INTERFACE* libraries, we will simply refer to them as libraries or SDK libraries from now on.

1.2.2. Library Structure

There are a number of layers of libraries within the Pico SDK. The follow section describes them from the lowest level to the highest level

1.2.2.1. Hardware Registers (`hardware_regs` library)

This library is a complete set of include files for all the RP2040 registers, autogenerated from the hardware itself. This is all you need if you want to peek or poke a memory mapped register directly, however higher level libraries provide more user friendly ways of achieving what you want in C/C++.

TIP

You can included these hardware_regs headers from assembly files

1.2.2.2. Hardware Structures (`hardware_structs` library)

This library provides a set of C structures which represent the memory mapped layout of RP2040 registers in memory. This allows you to replace something like the following (which you'd write in C with `hardware_regs`)

```
*(volatile uint32_t *)(PIO0_BASE + PIO_SM1_SHIFTCTRL_OFFSET) |=
PIO_SM1_SHIFTCTRL_AUTOPULL_BITS;
```

with something like this (where `pio0` is a pinter to type `pio_hw_t` at address PIO0_BASE).

```
pio0->sm[1].shiftctrl |= PIO_SM1_SHIFTCTRL_AUTOPULL_BITS;
```

The structures and associated pointers to memory mapped register blocks hide the complexity and potential error-proneness of dealing with individual memory locations, pointer types and volatile access.

Additionally, you can easilty use one of the aliases of the hardware in memory to perform atomic `set`, `clear`, or `xor` aliases of a piece of hardware to *set*, *clear* or *toggle* respectively the spcified bits in a hardware register (as opposed to having the CPU perform a read/modify/write); e.g:

```
hw_set_alias(pio0)->sm[1].shiftctrl = PIO_SM1_SHIFTCTRL_AUTOPULL_BITS;
```

1.2.2.3. Hardware Libraries (`hardware_xxx` libraries)

These are individual libraries (see Hardware APIs) providing actual APIs for interacting with each piece of physical hardware/peripheral.

These libraries generally provide functions for configuring or interacting with the peripheral at a functional level, rather than accessing registers directly, e.g.

```
pio_sm_set_wrap(pio, sm, bottom, top);
```

rather than:

```
pio->sm[sm].execctrl =
     (pio->sm[sm].execctrl & ~(PIO_SM0_EXECCTRL_WRAP_TOP_BITS |
PIO_SM0_EXECCTRL_WRAP_BOTTOM_BITS)) |
     (bottom << PIO_SM0_EXECCTRL_WRAP_BOTTOM_LSB) |
     (top << PIO_SM0_EXECCTRL_WRAP_TOP_LSB);
```

NOTE

`void pio_sm_set_wrap(PIO pio, uint sm, uint bottom, uint top) {}` is actually implemented as a `static inline` function in https://github.com/raspberrypi/pico-sdk/tree/pre_release/src/rp2_common/hardware_pio/include/hardware/pio.h directly as shown above. Using `static inline` functions is common in Pico SDK header files because such methods are often called with parameters that have fixed known values at compile time. In such cases, seemingly complex intializations or other calls can often be reduced (by the compiler) to a single register write in the calling code at compile time without need to pass arguments and call another overly general function (or in this case a read, AND with a constant value, OR with a constant value, and a write).

NOTE

The `hardware_` libraries are intended to have a very minimal runtime cost. They generally do not require any or much RAM, and do not rely on other runtime infrastructure. As such they can be used by low-level or other specialized applications that doesn't want to use the rest of the Pico SDK libraries and runtime.

1.2.2.3.1. Hardware Claiming

The hardware layer does provide one small abstraction which is the notion of claiming a piece of hardware. This minimal system allows registration of peripherals or parts of peripherals (e.g. DMA channels) that are in use, and the ability to atomically claim free ones at runtime. The common use of this system - in addition to allowing for safe runtime allocation of resources - provides a better runtime experience for catching software misconfigurations or accidental use of the same piece hardware by multiple independent libraries that would otherwise be very painful to debug.

1.2.2.4. Runtime Support (`pico_runtime`, `pico_standard_link`)

These are special libraries that bundle functionality which is common to most RP2040 based applications.

`pico_runtime` aggregates the libraries (see pico_runtime) that provide a familiar C environment for executing code, including:

- Runtime startup and initialization
- Choice of language level single/double precision float point support (and access to the fast on RP2040 implementations)

- Compact `printf` support, and mapping of `stdout`
- Language level `/` and `%` support for fast division using RP2040`s hardware dividers.

`pico_standard_link` encapsulates the standard linker setup needed to configure the type of application binary layout in memory, and link to any additional C and/or C++ runtime libraries.

TIP

Both `pico_runtime` and `pico_standard_link` are included with `pico_stdlib`

1.2.2.5. Higher level functionality (`pico_xxx`)

These libraries (see High Level APIs) provide higher level functionality and abstraction more commonly associated with Operating Systems. Here you will find:

- Alarms, timers and time functions
- Multi-core support and synchronization primitives
- Audio support (via PIO)
- Low-level Video support (via PIO)
- Utility functions and data structures

These libraries are generally built upon one or more underlying `hardware_` libraries, and often depend on each other

NOTE

More libraries will be forthcoming in the future (e.g. file system support, SDIO support via (PIO)), some of which can be found as works-in-progess in the pico-extras GitHub repository.

1.2.2.6. TinyUSB (`tinyusb_dev` and `tinyusb_host`)

We provide a RP2040 port of TinyUSB as the standard device and host USB support library within the Pico SDK.

The `tinyusb_dev` or `tinyusb_host` libraries within the Pico SDK allow you to add TinyUSB device or host support to you application by simply adding a dependency in your executable in `CMakeLists.txt`

1.2.3. Directory Structure

We have discussed libraries such as `pico_stdlib` and `hardware_uart` above. Imagine you wanted to add some code using RP2040's DMA controller to your hello_world sample. To do this you need to add a dependency on another library `hardware_dma` which is not included by default by `pico_stdlib` (`hardware_uart` is).

You would change your `CMakeLists.txt` such that you now did the following to add both `pico_stdlib` and `hardware_dma` to the `hello_world` target (executable) (note the line breaks are not required, but are perhaps clearer)

```
target_link_libraries(hello_world
    pico_stdlib
    hardware_dma)
```

And in your source code you would:

```
#include "hardware/dma.h"
```

This is the convention for all toplevel Pico SDK library headers. The library is call `foo_bar` and the associated header is `"foo/bar.h"`

NOTE

> Some libraries have additional headers which are located in foo/bar/other.h

1.2.3.1. Location of files

Of course you may want to actually find the files in question (although most IDEs will do this for you). The on disk files are actually split into multiple toplevel directories…

Whilst you are probably currently focused on building a binary to run on Raspberry Pi Pico which uses a RP2040 the Pico SDK is structured in a more general way. This is for two reasons

1. To support other future chips in the RP2 family
2. To support testing of your code off device (this is *host* mode)

The latter is super useful for writing and running unit tests, but also as you develop your software (for example your debugging code or work in progress software might actually be too big or use too much RAM to fit on the device).

The code is thus split into top level directories as follows:

Table 1. Top-level directories

Path	Description
`src/rp2040/`	This contains the `hardware_regs` and `hardware_structs` libraries mentioned earlier, which are specific to the RP2040.
`src/rp2_common/`	This contains the `hardware_` library implementations for individual hardware components, and `pico_` libraries or library implementations that are closely tied to the RP2040 hardware. This is separate from `/src/rp2040` as there may be future revision of the RP2040 or other chips in the *RP2* family which can use a common SDK and API, but may have subtly different register definitions.
`src/common/`	This is code that is common to all builds. This is generally headers providing hardware abstractions for functionality which are simulated in host mode, along with a lot of the `pico_` library implementations which to the extent they use hardware, do so only through those hardware abstractions.
`src/host/`	This is a basic set of replacement Pico SDK library implementations sufficient to get simple Raspberry Pi Pico applications running on your computer (Raspberry Pi OS, Linux, macOS or Windows using Cygwin or Windows Subsystem for Linux). This is not intended to be a fully functional simulator, however it is possible to inject additional implementations of libraries to provide more complete functionality.

There is a CMake variable `PICO_PLATFORM` that controls the environment you are build for:

When doing a regular RP2040 build (`PICO_PLATFORM=rp2040` the default), you get code from `common`, `rp2_common` and `rp2040`; when doing a host build (`PICO_PLATFROM=host`), you get code from `common` and `host`.

Within each top-level directory, the libraries have the following structure (reading `foo_bar` as something like `hardware_uart` or `pico_time`)

```
top-level_dir/
top-level_dir/foo_bar/include/foo/bar.h      # header file
```

```
top-level_dir/foo_bar/CMakeLists.txt          # build configuration
top-level_dir/foo_bar/bar.c                   # source file(s)
```

NOTE

> the directory `top-level_dir/foo_bar/include` is added as an include directory to the *INTERFACE* library foo_bar which is what allows you to include `"foo/bar.h"` in your application

1.2.4. Customization & Configuration using pre-processor variables

The Pico SDK allows use of compile time definitions to customize the behavior/capabilities of libraries, and to specify settings (e.g. physical pins) that are unlikely to be changed at runtime This allows for much smaller more efficient code, and avoids additional runtime overheads and the inclusion of code for configurations you *might* choose at runtime even though you actually don't (e.g. support PWM audio when you are only using I2S)!

Remember that because of the use of *INTERFACE* libraries, all the libaries your application(s) depend on are built from source for each application in your build, so you can even build multiple variants of the same application with different baked in behaviors.

Pre-processor variables may be specified in a number of ways, described in the following sections:

See TODO: config section for more information on configuration of individual libraries.

NOTE

> Whether compile time configuration or runtime configuration or both is supported/required is dependent on the particular library itself. The general philosophy however, is to allow sensible default behavior without providing any settings (beyond those provided by the board config).

1.2.4.1. Pre-processor variables via "board configuration"

Many of the common configuration settings are actually related to the particular RP2040 board being used, and include default pin settings for various Pico SDK libraries. The board being used is specified via the `PICO_BOARD` CMake variable which may be specified on the CMake command line or in the environment. The default `PICO_BOARD` if not specified is `pico`.

The board configuration provides a header file which specifies defaults if not otherwise specified; for example https://github.com/raspberrypi/pico-sdk/tree/pre_release/src/boards/include/boards/pico.h specifies

```
#ifndef PICO_DEFAULT_LED_PIN
#define PICO_DEFAULT_LED_PIN 25
#endif
```

The header `my_board_name.h` is included by all other Pico SDK headers as a result of setting `PICO_BOARD=my_board_name`. You may wish to specify your own board configuration in which case you can set PICO_BOARD_HEADER_DIRS in the environment or CMake to a semicolon separated list of paths to search for `my_board_name.h`.

See [getting_started_cmake_vars] for more information on `PICO_BOARD`, `PICO_BOARD_HEADER_DIRS` and other Pico SDK CMake variables TO DO: GRAHAM/LIAM: link getting_started_cmake_vars to the appropriate content, after it's been added

1.2.4.2. Pre-processor variables per binary (or library) as part of the build

We could modify the https://github.com/raspberrypi/pico-examples/tree/pre_release/hello_world/CMakeLists.txt with `target_compile_definitions` to specify an alternate set of UART pins to use.

Modified hello_world CMakeLists.txt specifying different UART pins

```
add_executable(hello_world
    hello_world.c
)

# SPECIFY two preprocessor definitions for the target hello_world ①
target_compile_definitions(hello_world PRIVATE
   PICO_DEFAULT_UART_TX_PIN=16
   PICO_DEFAULT_UART_RX_PIN=17
)

# Pull in our pico_stdlib which aggregates commonly used features
target_link_libraries(hello_world pico_stdlib)

# create map/bin/hex/uf2 file etc.
pico_add_extra_outputs(hello_world)
```

1.2.5. Builder Pattern for Hardware Configuration APIs

Setting up hardware registers correctly can be messy; there is often a lot of shifting ORing and MASKing involved if you work with the hardware directly (e.g. using the `hardware_regs` library).

The Pico SDK uses a *builder pattern* for the more complex configurations, which provides the following benefits:

1. Readability of code (no more dozen integer/boolean parameter methods!)
2. Tiny runtime code (thanks to the compiler)
3. Less brittle (the addition of another item to a hardware configuration will not break existing code)

Take the following hypothetical code example to (quite extensively) configure a DMA channel:

Hypothetical example of DMA configuration code

```
int dma_channel = 3; ①
dma_channel_config config = dma_get_default_channel_config(dma_channel); ②
channel_config_set_read_increment(&config, true);
channel_config_set_write_increment(&config, true);
channel_config_set_dreq(&config, DREQ_SPI0_RX);
channel_config_set_transfer_data_size(&config, DMA_SIZE_8);
dma_set_config(dma_channel, &config, false);
```

The net effect, that the compiler actually reduces all of the above to the following code.

Effective code produced by the C compiler for the DMA configuration

```
*(volatile uint32_t *)(DMA_BASE + DMA_CH3_AL1_CTRL_OFFSET) = 0x00089831;
```

NOTE

The Pico SDK code is designed to make builder patterns efficient in both *Release* and *Debug* builds. Additionally, even if not all values are known constant at compile time, the compiler can still produce the most efficient code possible based on the values that are known.

1.2.6. Function Naming

Pico SDK functions follow a common naming convention for consistency and to avoid name conflicts. Some names are quite long, but that is deliberate to be as specific as possible about functionality, and of course because the Pico SDK API is a C API and does not support function overloading.

1.2.6.1. Name prefix

Functions are prefixed by the libarary/functional area they belong to; e.g. public functions in the `hardware_dma` library are prefixed with `dma_`. Sometime the prefix regers to a sub group of library functionality (e.g. `channel_config_`ulticore)

1.2.6.2. Verb

A verb typically follows the prefix specifying that action performed by the function. `set_` and `get_` (or `is_` for booleans) are probably the most common and should always be present; i.e. a hypothetical method would be `oven_get_temperature()` and not `oven_temperature()`

1.2.6.3. Suffixes

1.2.6.3.1. Blocking/Non-Blocking Functions and Timeouts

Table 2. Pico SDK Suffixes for (non-)blocking functions and timeouts.

Suffix	Param	Description
(none)		The method is non-blocking, i.e. it does not wait on any external condition that could potentially take a long time.
`_blocking`		The method is blocking, and may potentially block indefinitely until some specific condition is met.
`_blocking_until`	`absolute_time_t until`	The method is blocking until some specific condition is met, however it will return early with a timeout condition (see Section 1.2.6.4) if the `until` time is reached.
`_timeout_ms`	`uint32_t timeout_ms`	The method is blocking until some specific condition is met, however it will return early with a timeout condition (see Section 1.2.6.4) after the specified number of milliseconds
`_timeout_us`	`uint64_t timeout_us`	The method is blocking until some specific condition is met, however it will return early with a timeout condition (see Section 1.2.6.4) after the specified number of microseconds

TODO: I know there's a few more suffixes

1.2.6.4. Return Codes and Error Handling

TODO: how to catch invalid arguments etc

TODO: ...

1.2.7. *static inline* Functions

As mentioned in Section 1.2.5, Pico SDK functions are often implemented as `static inline` functions in header files.

This is done for speed and code size! The code space needed to setup parameters for a regular call to a small function in another compilation unit can be bigger than the function implementation, and additionally the compiler can often infer the values of parameters at compile time, and so the inlined version of the function given those parameters is often significantly smaller and indeed faster than the generic version defined elsewhere. This is particularly true of functions with large numbers of arguments.

1.3. Pico SDK Runtime

For those coming from non embedded programming, or from other devices, this section will give you an idea of how various C/C++ language level concepts are handled within the Pico SDK

1.3.1. stdout/stdin support

1.3.2. Floating-point support

The Pico SDK provides a highly optimized single and double precision floating point implementation. In addition to being fast, many of the functions are actually implemented using support provided in the RP2040 bootrom, and so are not only fast, but also have a very minimal impact on your program size.

This implementation is used by default as it is the best choice in the majority of cases, however it is also possible to switch to using the regular compiler soft floating point support.

1.3.2.1. Functions

The Pico SDK provides implementations for all the standard functions from `math.h`. Additional functions can be found in `pico/float.h` and `pico/double.h`.

1.3.2.2. Speed / Tradeoffs

The overall goal for the bootrom floating-point routines is to achieve good performance within a small footprint, the emphasis is more on improved performance for the basic operations (add, subtract, multiply, divide and square root, and all conversion functions) and more on reduced footprint for the scientific functions (trigonometric functions, logarithms and exponentials).

The IEEE single- and double-precision data formats are used throughout, but in the interests of reducing code size, input denormals are treated as zero and output denormals are flushed to zero, and output NaNs are rendered as infinities. Only the round-to-nearest, even-on-tie rounding mode is supported. Traps are not supported. Whether input NaNs are treated as infinities or propagated is configurable.

The five basic operations (add, subtract, multiply, divide, sqrt) return results that are always correctly rounded (round-to-nearest).

The scientific functions always return results within 1 ULP (unit in last place) of the exact result. In many cases results are better.

The scientific functions are calculated using internal fixed-point representations so accuracy (as measured in ULP error rather than in absolute terms) is poorer in situations where converting the result back to floating point entails a large normalising shift. This occurs, for example, when calculating the sine of a value near a multiple of pi, the cosine of a value near an odd multiple of pi/2, or the logarithm of a value near 1. Accuracy of the tangent function is also poorer when the result is very large. Although covering these cases is possible, it would add considerably to the code footprint, and there are few types of program where accuracy in these situations is essential.

The following table shows the results from a benchmark

NOTE

Whilst the Pico SDK floating point support makes use of the routines in the RP2040 bootrom, they do not suffer the same limitations (e.g. sin/cos) range, but are instead designed to be largely indistinguishable from the compiler provided ones. Certain smaller functions have also been re-implemented to make them faster than the more generic bootrom version.

Table 3. Pico SDK implementation vs GCC 9 implementation for ARM AEABI floating point functions (these weridly named functions provide the support for basic operations on float and double types)

Function	ROM/SDK (us)	GCC 9 (us)	Speedup
__aeabi_fadd	72.4	99.8	138%
__aeabi_fsub	86.7	133.6	154%
__aeabi_frsub	89.8	140.6	157%
__aeabi_fmul	61.5	145	236%
__aeabi_fdiv	74.7	437.5	586%
__aeabi_fcmplt	39	61.1	157%
__aeabi_fcmple	40.5	61.1	151%
__aeabi_fcmpgt	40.5	61.2	151%
__aeabi_fcmpge	41	61.2	149%
__aeabi_fcmpeq	40	41.5	104%
__aeabi_dadd	99.4	142.5	143%
__aeabi_dsub	114.2	182	159%
__aeabi_drsub	108	181.2	168%
__aeabi_dmul	168.2	338	201%
__aeabi_ddiv	197.1	412.2	209%
__aeabi_dcmplt	53	88.3	167%
__aeabi_dcmple	54.6	88.3	162%
__aeabi_dcmpgt	54.4	86.6	159%
__aeabi_dcmpge	55	86.6	157%
__aeabi_dcmpeq	54	64.3	119%
__aeabi_f2iz	17	24.5	144%
__aeabi_f2uiz	42.5	106.5	251%
__aeabi_f2lz	63.1	1240.5	1966%
__aeabi_f2ulz	46.1	1157	2510%
__aeabi_i2f	43.5	63	145%
__aeabi_ui2f	41.5	55.8	134%
__aeabi_l2f	75.2	643.3	855%
__aeabi_ul2f	71.4	531.5	744%
__aeabi_d2iz	30.6	44.1	144%

__aeabi_d2uiz	75.7	159.1	210%
__aeabi_d2lz	81.2	1267.8	1561%
__aeabi_d2ulz	65.2	1148.3	1761%
__aeabi_i2d	44.4	61.9	139%
__aeabi_ui2d	43.4	51.3	118%
__aeabi_l2d	104.2	559.3	537%
__aeabi_ul2d	102.2	458.1	448%
__aeabi_f2d	20	31	155%
__aeabi_d2f	36.4	66	181%

1.3.2.3. Configuration and alternate implementations

There are three different floating point implementations provided

Name	Description
`default`	`The default; equivalent to rom`
`rom`	Use the fast/compact Pico SDK/bootrom implementations
`compiler`	Use the standard compiler provided soft floating point implementations
`none`	Map all functions to an runtime assertion. You can use this when you know you don't want any floating point support to make sure it isn't accidentally pulled in by some library.

These settings can be set independently for both "float" and "double":

For "float" you can call `pico_set_float_implementation(TARGET  NAME)` in your CMakeListst.txt to choose a specific implementation for a particular target, or set the CMake variable `PICO_DEFAULT_FLOAT_IMPL` to `pico_float_NAME` to set the default.

For "double" you can call `pico_set_double_implementation(TARGET  NAME)` in your CMakeListst.txt to choose a specific implementation for a particular target, or set the CMake variable `PICO_DEFAULT_DOUBLE_IMPL` to `pico_double_NAME` to set the default.

TIP

The `rom` floating point library adds very little to your binary size, however it must include implementations for any used functions that are not present in V1 of the bootrom. If you know that you are only using RP2040s with V2 of the bootrom, then you can specify defines `PICO_FLOAT_SUPPORT_ROM_V1=0` and `PICO_DOUBLE_SUPPORT_ROM_V1=0` so the extra code will not be included. Any use of those functions on a RP2040 with a V1 bootrom will cause a panic at runtime. See the RP2040 datasheet for more specific details of the bootrom functions.

1.3.2.3.1. NaN propagation

The Pico SDK implementation by default treats input *NaNs* as infinites. If you require propagation of NaN inputs to outputs, then you can set the compile definitions `PICO_FLOAT_PROPAGATE_NANS` and `PICO_DOUBLE_PROPAGATE_NANS` to 1, at the cost of a small runtime overhead.

1.3.3. Multi-core support

Multi-core support should be familiar to those used to programming with threads in other environments. The second core is just treated as a second *thread* within your application; initially the second core (`core1` as it is usually referred to; the main application thread runs on `core0`) is halted, however you can start it executing some function in parallel from your main application thread.

Care should be taken with calling C library functions from both cores simultaneously as they are generally not designed to be thread safe. You can use the `mutex_` API provided by the SDK in the `pico_sync` library (https://github.com/raspberrypi/pico-sdk/tree/pre_release/src/common/pico_sync/include/pico/mutex.h) from within your own code.

NOTE

That the Pico SDK version of printf is always safe to call from both cores. `malloc`, `calloc` and `free` are additionally wrapped to make it thread safe when you include the `pico_multicore` as a convenience for C++ programming, where some object allocations may not be obvious.

1.4. Using C++

The Pico SDK has a C style API, however the Pico SDK headers may be safely included from C++ code, and the functions called (they are declared with C linkage).

To save space, exception handling is disabled by default; this can be overriden with the CMake environment variable PICO_CXX_ENABLE_EXCEPTIONS=1 todo: make a method per binary

TODO: C++ library support; destructors, wotnot

1.5. Libraries

There are C libraries for all the hardware blocks and controllers on the Raspberry Pi Pico.

Table 4. Pico SDK Hardware INTERFACE Libraries

Name	Description
hardware_adc	API to the Analog Digital Convertor (see hardware_adc)
hardware_claim	API to the hardware claim functions (see hardware_claim)
hardware_clocks	API to the clock hardware (e.g. system clock, reference clock) (see hardware_clocks)
hardware_divider	API to the hardware divider (see hardware_divider)
hardware_dma	API to the Direct Memory Access hardware (see hardware_dma)
hardware_flash	API for accessing the flash memory system (see hardware_flash)
hardware_gpio	API for accessing the RP2040 General Purpose Input Output (see hardware_gpio)
hardware_i2c	API for accessing the RP2040 I2C hardware (see hardware_i2c)
hardware_interp	API for accessing the RP2040 custom hardware interpolators (see hardware_interp)
hardware_irq	API for accessing the interrupt controller (see hardware_irq)
hardware_pio	API for accessing the Programmable Input Output block (see hardware_pio)
hardware_pll	API for accessing the Phase Lock Loop timing control (see hardware_pll)
hardware_pwm	API for accessing the RP2040 custom Pulse Width Modulation hardware (see hardware_pwm)
hardware_resets	API for accessing the hardware reset system (see hardware_resets)

Name	Description
hardware_rosc	API for setting up the Ring Oscillator (see hardware_rosc)
hardware_rtc	API for accessing the real time clock. (see hardware_rtc)
hardware_sleep	API for setting up the Sleep system (see hardware_sleep)
hardware_spi	API for accessing the RP2040 Serial Peripheral Interface hardware (see hardware_spi)
hardware_sync	API for accessing the hardware synchronisation primitives (see hardware_sync)
hardware_timer	API for setting up the Hardware timers (see hardware_timer)
hardware_uart	API for setting up the inbuilt UARTS (see hardware_uart)
hardware_watchdog	API for setting and handling the Watchdog timer (see hardware_watchdog)
hardware_xosc	API for setting up the Crystal Oscillator (see hardware_xosc)

1.6. Getting Started

Please refer to the Getting started with Raspberry Pi Pico for detailed instructions on getting up and running with the Pico SDK.

1.6.1. CMake (build) Configuration

TODO: This section is currently a bit of a dumping ground; gathering values to be documented

Note `NAME*` indicates a CMake variable that is made available to the preprocessoer as well, so is available during code compilation.

1.6.1.1. Output options

1.6.1.1.1. Board selection

PICO_BOARD*

PICO_BOARD_HEADER_DIRS (list)

PICO_BOARD_CMAKE_DIRS (list)

1.6.1.1.2. Configuration (advanced)

PICO_CONFIG_HEADER_FILES

PICO_<platform>_CONFIG_HEADER_FILES

1.6.1.1.3. Initialized by the Pico SDK

PICO_ON_DEVICE*

PICO_NO_HARDWARE*

1.6.1.1.4. Control of binary type produced (advanced)

These variables control how executables for the RP2040 are layed out in memory. The default is for the code and data to be entirely stored in flash with writable data (and some specifically marked) methods to copied into RAM at startup.

PICO_DEFAULT_BINARY_TYPE	default	The default is flash binaries which are stored in and run from flash.
	no_flash	This option selects a RAM only binaries, that does not require any flash. Note: this type of binary must be loaded on each device reboot via a UF2 file or from the debugger.
	copy_to_ram	This option selects binaries which are stored in flash, but copy themselves to RAM before executing.
	blocked_ram	
PICO_NO_FLASH*	0 / 1	Equivalent to PICO_DEFAULT_BINARY_TYPE=no_flash if 1
PICO_COPY_TO_RAM*	0 / 1	Equivalent to PICO_DEFAULT_BINARY_TYPE=copy_to_ram if 1
PICO_USE_BLOCKED_RAM*	0 / 1	Equivalent to PICO_DEFAULT_BINARY_TYPE=blocked_ram if 1

TIP

> The binary type can be set on a per executable target (as created by `add_executable`) basis by calling `pico_set_binary_type(target type)` where type is the same as for `PICO_DEFAULT_BINARY_TYPE`

TODO

todo:Note we should reference which are persistent (at least as a heading) and mention `cmake /LH ..`

PICO_BOOT_STAGE2_FILE*

PICO_INCLUDE_DIRS (list)

PICO_BARE_METAL 0/1

PICO_PLATFORM_EXTRA_LIBRARIES

PICO_DEFAULT_DIVIDER_IMPL

#PICO_DIVIDER_HARDWARE

#PICO_DIVIDER_COMPILER

PICO_DEFAULT_FLOAT_IMPL

#PICO_FLOAT_NONE

#PICO_FLOAT_ROM

#PICO_FLOAT_COMPILER

PICO_DEFAULT_DOUBLE_IMPL

#PICO_DOUBLE_NONE

#PICO_DOUBLE_ROM

#PICO_DOUBLE_COMPILER

#PICO_MULTICORE

PICO_TOOLCHAIN_PATH

PICO_COMPILER

PICO_TINYUSB_PATH

PICO_PLATFORM_CMAKE_FILE

PICO_SYMLINK_ELF_AS_FILENAME

(PICO_SDK_PATH is set)

1.6.1.1.5. C++

PICO_CXX_ENABLE_EXCEPTIONS todo make a target method

PICO_CXX_ENABLE_RTTI todo make a target method

PICO_CXX_USE_CXA_ATEXIT todo make a target method

PICO_NO_GC_SECTIONS todo make a target method

Host specific

PICO_NO_PRINTF* todo: needs love

PICO_TIMER_NO_ALARM_SUPPORT

1.6.1.2. Debug/Release builds

CMAKE_BUILD_TYPE

1.6.2. CMake Example

1.6.3. misc preprocessor

#PICO_FLOAT_PROPAGATE_NANS

#PICO_DOUBLE_PROPAGATE_NANS

#PICO_ENTER_USB_BOOT_ON_EXIT

autoset:

#PICO_BUILD=1

1.6.4. Hardware Divider

The SDK includes optimized 32- and 64-bit division functions accelerated by the RP2040 hardware divider, which are seamlessly integrated with the C `/` and `%` operators. The SDK also supplies a high level API which includes combined quotient and remainder functions for 32- and 64-bit, also accelerated by the hardware divider.

Chapter 2. Library Documentation

TODO: top level should be headings, then sub levels should be in a (2-level because of sub-sections) table with the brief documentation

i.e. something like

2.1. High Level APIs

pico_audio	Common API for audio output
pico_audio_i2s	I2S audio output using the PIO
pico_audio_pwm	PWM audio output (with optional noise shaping and error diffusion) using the PIO
pico_multicore	Adds support for running code on the second processor core (core1)
fifo	Functions for inter-core FIFO
pico_stdlib	Aggregation of a core subset of Pico SDK libraries used by most executables along with some additional utility methods
pico_sync	Synchronization primitives and mutual exclusion
critical_section	Critical Section API for short-lived mutual exclusion safe for IRQ and multi-core
mutex	Mutex API for non IRQ mutual exclusion between cores
sem	Semaphore API for restricting access to a source

Hardware APIs

- hardware_base
- hardware_claim
- hardware_adc
- hardware_clocks
- hardware_divider
- hardware_dma
- hardware_flash
- hardware_gpio
- hardware_i2c
- hardware_interp
- hardware_irq
- hardware_pio
- hardware_pll
- hardware_pwm
- hardware_resets
- hardware_rosc
- hardware_rtc
- hardware_sleep
- hardware_spi
- hardware_sync
- hardware_timer
- hardware_uart
- hardware_vreg
- hardware_watchdog
- hardware_xosc
- channel_config

- interp_config
- sm_config

High Level APIs

- pico_audio
- pico_audio_i2s
- pico_audio_pwm
- pico_multicore
- pico_stdlib
- pico_sync
- pico_scanvideo
- pico_scanvideo_dpi
- pico_time
- pico_util
- fifo
- critical_section
- mutex
- sem
- timestamp
- sleep
- alarm
- repeating_timer
- datetime
- pheap
- queue

Third-party Libraries

- tinyusb_device
- tinyusb_host

Runtime Infrastructure

- pico_base
- pico_bit_ops
- pico_bootrom
- pico_divider
- pico_double
- pico_float
- pico_int64
- pico_malloc
- pico_memory
- pico_platform
- pico_runtime
- pico_standard_link

Miscellaneous

- boot_picoboot
- boot_uf2

Chapter 3. API Documentation

3.1. Hardware APIs

3.1.1. Modules

- `hardware_base`
  Low-level types and (atomic) accessors for memory-mapped hardware registers.
- `hardware_claim`
  Lightweight hardware resource management.
- `hardware_adc`
  Analog to Digital Converter (ADC) API.
- `hardware_clocks`
  Clock Management API.
- `hardware_divider`
  Low-level hardware-divider access.
- `hardware_dma`
  DMA Controller API.
- `hardware_flash`
  Low level flash programming and erase API.
- `hardware_gpio`
  General Purpose Input/Output (GPIO) API.
- `hardware_i2c`
  I2C Controller API.
- `hardware_interp`
  Hardware Interpolator API.
- `hardware_irq`
  Hardware interrupt handling.
- `hardware_pio`
  Programmable I/O (PIO) API.
- `hardware_pll`
  Phase Locked Loop control APIs.
- `hardware_pwm`
  Hardware Pulse Width Modulation (PWM) API.
- `hardware_resets`
  Hardware Reset API.
- `hardware_rosc`
- `hardware_rtc`
  Hardware Real Time Clock API.
- `hardware_sleep`
- `hardware_spi`
  Hardware SPI API.

- `hardware_sync`
  Low level hardware spin-lock, barrier and processor event API.
- `hardware_timer`
  Low-level hardware timer API.
- `hardware_uart`
  Hardware UART API.
- `hardware_vreg`
  Voltage Regulation API.
- `hardware_watchdog`
  Hardware Watchdog Timer API.
- `hardware_xosc`
  Crystal Oscillator (XOSC) API.

3.2. hardware_base

Low-level types and (atomic) accessors for memory-mapped hardware registers. More…

3.2.1. Functions

- `static void hw_set_bits (io_rw_32 *addr, uint32_t mask)`
  Atomically set the specified bits to 1 in a HW register. More…
- `static void hw_clear_bits (io_rw_32 *addr, uint32_t mask)`
  Atomically clear the specified bits to 0 in a HW register. More…
- `static void hw_xor_bits (io_rw_32 *addr, uint32_t mask)`
  Atomically flip the specified bits in a HW register. More…
- `static void hw_write_masked (io_rw_32 *addr, uint32_t values, uint32_t write_mask)`
  Set new values for a sub-set of the bits in a HW register. More…

3.2.2. Detailed Description

Low-level types and (atomic) accessors for memory-mapped hardware registers.

hardware_base defines the low level types and access functions for memory mapped hardware registers. It is includd by default by all other hardware libraries.

The following register access typedefs codify the access type (read/write) and the bus size (8/16/32) of the hardware register. The register type names are formed by concatenating one from each of the 3 parts A, B, C

A	B	C	Meaning
io_			A Memory mapped IO register
	ro_		read-only access
	rw_		read-write access
	wo_		write-only access (can't actually be enforced via C API)
		8	8-bit wide access

A	B	C	Meaning
		16	16-bit wide access
		32	32-bit wide access

When dealing with these types, you will always using a pointer, i.e. io_rw_32 *some_reg is a pointer to a read/write 32 bit register that you can write with *some_reg = value, or read with value = *some_reg.

RP2040 hardware is also aliased to provide atomic setting, clear or flipping of a subset of the bits within a hardware register so that concurrent access by two cores is always consistent with one atomic operation being performed first, followed by the second.

See hw_set_bits(), hw_clear_bits() and hw_xor_bits() provide for atomic access via a pointer to a 32 bit register

Additionally given a pointer to a structure representing a piece of hardware (e.g. dma_hw_t *dma_hw for the DMA controller), you can get an alias to the entire structure such that writing any member (register) within the structure is equivalent to an atomic operation via hw_set_alias(), hw_clear_alias() or hw_xor_alias()…

For example hw_set_alias(dma_hw)→inte1 = 0x80; will set bit 7 of the INTE1 register of the DMA controller, leaving the other bits unchanged.

3.2.3. Function Documentation

3.2.3.1. hw_clear_bits

```
static void hw_clear_bits (io_rw_32 *addr,
       uint32_t mask)
```

Atomically clear the specified bits to 0 in a HW register.

Parameters

- addr Address of writable register
- mask Bit-mask specifying bits to clear

3.2.3.2. hw_set_bits

```
static void hw_set_bits (io_rw_32 *addr,
       uint32_t mask)
```

Atomically set the specified bits to 1 in a HW register.

Parameters

- addr Address of writable register
- mask Bit-mask specifying bits to set

3.2.3.3. hw_write_masked

```
static void hw_write_masked (io_rw_32 *addr,
       uint32_t values,
       uint32_t write_mask)
```

Set new values for a sub-set of the bits in a HW register.

Sets destination bits to values specified in `values`, if and only if corresponding bit in `write_mask` is set

Note: this method allows safe concurrent modification of bits of a register, but multiple concurrent access to the same bits is still unsafe.

Parameters

- `addr` Address of writable register
- `values` Bits values
- `write_mask` Mask of bits to change

3.2.3.4. hw_xor_bits

```
static void hw_xor_bits (io_rw_32 *addr,
       uint32_t mask)
```

Atomically flip the specified bits in a HW register.

Parameters

- `addr` Address of writable register
- `mask` Bit-mask specifying bits to invert

3.3. hardware_claim

Lightweight hardware resource management. More…

3.3.1. Functions

- `void hw_claim_or_assert (uint8_t *bits, uint bit_index, const char *message)`
  Atomically claim a resource, panicking if it is already in use. More…
- `int hw_claim_unused_from_range (uint8_t *bits, bool required, uint bit_lsb, uint bit_msb, const char *message)`
  Atomically claim one resource out of a range of resources, optionally asserting if none are free. More…
- `bool hw_is_claimed (uint8_t *bits, uint bit_index)`
  Determine if a resource is claimed at the time of the call. More…
- `void hw_claim_clear (uint8_t *bits, uint bit_index)`
  Atomically unclaim a resource. More…
- `uint32_t hw_claim_lock ()`
  Acquire the runtime mutual exclusion lock provided by the hardware_claim library. More…
- `void hw_claim_unlock (uint32_t token)`
  Release the runtime mutual exclusion lock provided by the hardware_claim library. More…

3.3.2. Detailed Description

Lightweight hardware resource management.

hardware_claim provides a simple API for management of hardware resources at runtime.

This API is usually called by other hardware specific *claiming* APIs and provides simple multi-core safe methods to manipulate compact bit-sets representing hardware resources.

This API allows any other library to cooperatively participate in a scheme by which both compile time and runtime allocation of resources can co-exist, and conflicts can be avoided or detected (depending on the use case) without the libraries having any other knowledge of each other.

Facilities are providing for:

- Claiming resources (and asserting if they are already claimed)

- Freeing (unclaiming) resources
- Finding unused resources

3.3.3. Function Documentation

3.3.3.1. hw_claim_clear

```
void hw_claim_clear (uint8_t *bits,
       uint bit_index)
```

Atomically unclaim a resource.

The resource ownership is indicated by the bit_index bit in an array of bits.

Parameters

- `bits` pointer to an array of bits (8 bits per byte)
- `bit_index` resource to unclaim (bit index into array of bits)

3.3.3.2. hw_claim_lock

```
uint32_t hw_claim_lock ()
```

Acquire the runtime mutual exclusion lock provided by the hardware_claim library.

This method is called automatically by the other `hw_claim_` methods, however it is provided as a convenience to code that might want to protect other hardware initialization code from concurrent use.

Returns

- a token to pass to `hw_claim_unlock()`

3.3.3.3. hw_claim_or_assert

```
void hw_claim_or_assert (uint8_t *bits,
       uint bit_index,
       const char *message)
```

Atomically claim a resource, panicking if it is already in use.

The resource ownership is indicated by the bit_index bit in an array of bits.

Parameters

- `bits` pointer to an array of bits (8 bits per byte)
- `bit_index` resource to claim (bit index into array of bits)
- `message` string to display if the bit cannot be claimed; note this may have a single printf format "%d" for the bit

3.3.3.4. hw_claim_unlock

```
void hw_claim_unlock (uint32_t token)
```

Release the runtime mutual exclusion lock provided by the hardware_claim library.

Parameters

- `token` the token returned by the corresponding call to hw_claim_lock()

3.3.3.5. hw_claim_unused_from_range

```
int hw_claim_unused_from_range (uint8_t *bits,
       bool required,
       uint bit_lsb,
       uint bit_msb,
       const char *message)
```

Atomically claim one resource out of a range of resources, optionally asserting if none are free.

Parameters

- `bits` pointer to an array of bits (8 bits per byte)
- `required` true if this method should panic if the a resource is not free
- `bit_lsb` the lower bound (inclusive) of the resource range to claim from
- `bit_msb` the epper bound (inclusive) of the resource range to claim from
- `message` string to display if the bit cannot be claimed

Returns

- the bit index representing the calimed or -1 if none are available in the range, and required = false

3.3.3.6. hw_is_claimed

```
bool hw_is_claimed (uint8_t *bits,
       uint bit_index)
```

Determine if a resource is claimed at the time of the call.

The resource ownership is indicated by the bit_index bit in an array of bits.

Parameters

- `bits` pointer to an array of bits (8 bits per byte)
- `bit_index` resource to unclaim (bit index into array of bits)

Returns

- true if the resource is claimed

3.4. hardware_adc

Analog to Digital Converter (ADC) API. More…

3.4.1. Functions

- `void adc_init (void)`
  Initialise the ADC HW.
- `static void adc_gpio_init (uint gpio)`
  Initialise the gpio for use as an ADC pin. More…
- `static void adc_select_input (uint input)`
  ADC input select. More…
- `static void adc_set_round_robin (uint input_mask)`
  Round Robin sampling selector. More…
- `static void adc_set_temp_sensor_enabled (bool enable)`

Enable the onboard temperature sensor. More…

- `static uint16_t adc_read (void)`
  Perform a single conversion. More…
- `static void adc_run (bool run)`
  Enable or disable free-running sampling mode. More…
- `static void adc_set_clkdiv (float clkdiv)`
  Set the ADC Clock divisor. More…
- `static void adc_fifo_setup (bool en, bool dreq_en, uint16_t dreq_thresh, bool err_in_fifo, bool byte_shift)`
  Setup the ADC FIFO. More…
- `static bool adc_fifo_is_empty (void)`
  Check FIFO empty state. More…
- `static uint8_t adc_fifo_get_level (void)`
  Get number of entries in the ADC FIFO. More…
- `static uint16_t adc_fifo_get (void)`
  Get ADC result from FIFO. More…
- `static uint16_t adc_fifo_get_blocking (void)`
  Wait for the ADC FIFO to have data. More…
- `static void adc_fifo_drain (void)`
  Drain the ADC FIFO. More…
- `static void adc_irq_set_enabled (bool enabled)`
  Enable/Disable ADC interrupts. More…

3.4.2. Detailed Description

Analog to Digital Converter (ADC) API.

The RP2040 has an internal analogue-digital converter (ADC) with the following features:

- SAR ADC
- 500 kS/s (Using an independent 48MHz clock)
- 12 bit (9.5 ENOB)
- 5 input mux:
- 4 inputs that are available on package pins shared with GPIO[29:26]
- 1 input is dedicated to the internal temperature sensor
- 4 element receive sample FIFO
- Interrupt generation
- DMA interface

Although there is only one ADC you can specify the input to it using the adc_select_input() function. In round robin mode (adc_rrobin()) will use that input and move to the next one after a read.

User ADC inputs are on 0-3 (GPIO 26-29), the temperature sensor is on input 4.

Temperature sensor values can be approximated in centigrade as:

T = 27 - (ADC_Voltage - 0.706)/0.001721

The FIFO, if used, can contain up to 4 entries.

Example

```c
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/gpio.h"
#include "hardware/adc.h"

int main() {
    stdio_init_all();
    printf("ADC Example, measuring GPIO26\n");

    adc_init();

    // Make sure GPIO is high-impedance, no pullups etc
    adc_gpio_init(26);
    // Select ADC input 0 (GPIO26)
    adc_select_input(0);

    while (1) {
        // 12-bit conversion, assume max value == ADC_VREF == 3.3 V
        const float conversion_factor = 3.3f / (1 << 12);
        uint16_t result = adc_read();
        printf("Raw value: 0x%03x, voltage: %f V\n", result, result * conversion_factor);
        sleep_ms(500);
    }
}
```

3.4.3. Function Documentation

3.4.3.1. adc_fifo_drain

`static void adc_fifo_drain (void)`

Drain the ADC FIFO.

Will wait for any conversion to complete then drain the FIFO discarding any results.

3.4.3.2. adc_fifo_get

`static uint16_t adc_fifo_get (void)`

Get ADC result from FIFO.

Pops the latest result from the ADC FIFO.

3.4.3.3. adc_fifo_get_blocking

`static uint16_t adc_fifo_get_blocking (void)`

Wait for the ADC FIFO to have data.

Blocks until data is present in the FIFO

3.4.3.4. adc_fifo_get_level

`static uint8_t adc_fifo_get_level (void)`

Get number of entries in the ADC FIFO.

The ADC FIFO is 4 entries long. This function will return how many samples are currently present.

3.4.3.5. adc_fifo_is_empty

```
static bool adc_fifo_is_empty (void)
```

Check FIFO empty state.

Returns

- Returns true if the fifo is empty

3.4.3.6. adc_fifo_setup

```
static void adc_fifo_setup (bool en,
       bool dreq_en,
       uint16_t dreq_thresh,
       bool err_in_fifo,
       bool byte_shift)
```

Setup the ADC FIFO.

FIFO is 4 samples long, if a conversion is completed and the FIFO is full the result is dropped.

Parameters

- `en` Enables write each conversion result to the FIFO
- `dreq_en` Enable DMA requests when FIFO contains data
- `dreq_thresh` Threshold for DMA requests/FIFO IRQ if enabled.
- `err_in_fifo` If enabled, bit 15 of the FIFO contains error flag for each sample
- `byte_shift` Shift FIFO contents to be one byte in size (for byte DMA) - enables DMA to byte buffers.

3.4.3.7. adc_gpio_init

```
static void adc_gpio_init (uint gpio)
```

Initialise the gpio for use as an ADC pin.

Prepare a GPIO for use with ADC, by disabling all digital functions.

Parameters

- `gpio` The GPIO number to use. Allowable GPIO numbers are 26 to 29 inclusive.

3.4.3.8. adc_init

```
void adc_init (void)
```

Initialise the ADC HW.

3.4.3.9. adc_irq_set_enabled

```
static void adc_irq_set_enabled (bool enabled)
```

Enable/Disable ADC interrupts.

Parameters

- `enabled` Set to true to enable the ADC interrupts, false to disable

3.4.3.10. adc_read

`static uint16_t adc_read (void)`

Perform a single conversion.

Performs an ADC conversion, waits for the result, and then returns it.

Returns

- Result of the conversion.

3.4.3.11. adc_run

`static void adc_run (bool run)`

Enable or disable free-running sampling mode.

Parameters

- `run` false to disable, true to enable free running conversion mode.

3.4.3.12. adc_select_input

`static void adc_select_input (uint input)`

ADC input select.

Select an ADC input. 0…3 are GPIOs 26…29 respectively. Input 4 is the onboard temperature sensor.

Parameters

- `input` Input to select.

3.4.3.13. adc_set_clkdiv

`static void adc_set_clkdiv (float clkdiv)`

Set the ADC Clock divisor.

Period of samples will be (1 + div) cycles on average. Note it takes 96 cycles to perform a conversion, so any period less than that will be clamped to 96.

Parameters

- `clkdiv` If non-zero, conversion will be started at intervals rather than back to back.

3.4.3.14. adc_set_round_robin

`static void adc_set_round_robin (uint input_mask)`

Round Robin sampling selector.

This function sets which inputs are to be run through in round robin mode. Value between 0 and 0x1f (bit 0 to bit 4 for GPIO 26 to 29 and temperature sensor input respectively)

Parameters

- `input_mask` A bit pattern indicating which of the 5 inputs are to be sampled. Write a value of 0 to disable round robin sampling.

3.4.3.15. adc_set_temp_sensor_enabled

`static void adc_set_temp_sensor_enabled (bool enable)`

Enable the onboard temperature sensor.

Parameters

- `enable` Set true to power on the onboard temperature sensor, false to power off.

3.5. hardware_clocks

Clock Management API. More…

3.5.1. Typedefs

- `typedef void(* resus_callback_t )(void)`
  Resus callback function type. More…

3.5.2. Enumerations

- `enum clock_index { clk_gpout0 = 0, clk_gpout1, clk_gpout2, clk_gpout3, clk_ref, clk_sys, clk_peri, clk_usb, clk_adc, clk_rtc, CLK_COUNT }`
  Enumeration identifying a hardware clock.

3.5.3. Functions

- `void clocks_init ()`
  Initialise the clock hardware. More…
- `bool clock_configure (enum clock_index clk_index, uint32_t src, uint32_t auxsrc, uint32_t src_freq, uint32_t freq)`
  Configure the specified clock. More…
- `void clock_stop (enum clock_index clk_index)`
  Stop the specified clock. More…
- `uint32_t clock_get_hz (enum clock_index clk_index)`
  Get the current frequency of the specified clock. More…
- `uint32_t frequency_count_khz (uint src)`
  Measure a clocks frequency using the Frequency counter. More…
- `void clock_set_reported_hz (enum clock_index clk_index, uint hz)`
  Set the "current frequency" of the clock as reported by clock_get_hz without actually changing the clock. More…
- `void clocks_enable_resus (resus_callback_t resus_callback)`
  Enable the resus function. Restarts clk_sys if it is accidentally stopped. More…
- `void clock_gpio_init (uint gpio, uint src, uint div)`
  Output an optionally divided clock to the specified gpio pin. More…
- `bool clock_configure_gpin (enum clock_index clk_index, uint gpio, uint32_t src_freq, uint32_t freq)`
  Configure a clock to come from a gpio input. More…

3.5.4. Detailed Description

Clock Management API.

This API provices a high level interface to the clock functions.

The clocks block provides independent clocks to on-chip and external components. It takes inputs from a variety of clock sources allowing the user to trade off performance against cost, board area and power consumption. From these sources it uses multiple clock generators to provide the required clocks. This architecture allows the user flexibility to start and stop clocks independently and to vary some clock frequencies whilst maintaining others at their optimum frequencies

Please refer to the datasheet for more details on the RP2040 clocks.

The clock source depends on which clock you are attempting to configure. The first table below shows main clock sources. If you are not setting the Reference clock or the System clock, or you are specifying that one of those two will be using an auxiliary clock source, then you will need to use one of the entries from the subsequent tables.

Main Clock Sources

Source	Reference Clock	System Clock
ROSC	CLOCKS_CLK_REF_CTRL_SRC_VALUE_ROSC_CLKSRC_PH	
Auxiliary	CLOCKS_CLK_REF_CTRL_SRC_VALUE_CLKSRC_CLK_REF_AUX	CLOCKS_CLK_SYS_CTRL_SRC_VALUE_CLKSRC_CLK_SYS_AUX
XOSC	CLOCKS_CLK_REF_CTRL_SRC_VALUE_XOSC_CLKSRC	
Reference		CLOCKS_CLK_SYS_CTRL_SRC_VALUE_CLK_REF

Auxiliary Clock Sources

The auxiliary clock sources available for use in the configure function depend on which clock is being configured. The following table describes the available values that can be used. Note that for clk_gpout[x], x can be 0-3.

Aux Source	clk_gpout[x]	clk_ref	clk_sys
System PLL	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS		CLOCKS_CLK_SYS_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS
GPIO in 0	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLKSRC_GPIN0	CLOCKS_CLK_REF_CTRL_AUXSRC_VALUE_CLKSRC_GPIN0	CLOCKS_CLK_SYS_CTRL_AUXSRC_VALUE_CLKSRC_GPIN0
GPIO in 1	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLKSRC_GPIN1	CLOCKS_CLK_REF_CTRL_AUXSRC_VALUE_CLKSRC_GPIN1	CLOCKS_CLK_SYS_CTRL_AUXSRC_VALUE_CLKSRC_GPIN1
USB PLL	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLKSRC_PLL_USB	CLOCKS_CLK_REF_CTRL_AUXSRC_VALUE_CLKSRC_PLL_USB	CLOCKS_CLK_SYS_CTRL_AUXSRC_VALUE_CLKSRC_PLL_USB
ROSC	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_ROSC_CLKSRC		CLOCKS_CLK_SYS_CTRL_AUXSRC_VALUE_ROSC_CLKSRC
XOSC	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_XOSC_CLKSRC		CLOCKS_CLK_SYS_CTRL_AUXSRC_VALUE_ROSC_CLKSRC
System clock	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLK_SYS		

Aux Source	clk_gpout[x]	clk_ref	clk_sys
USB Clock	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLK_USB		
ADC clock	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLK_ADC		
RTC Clock	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLK_RTC		
Ref clock	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLK_REF		

Aux Source	clk_peri	clk_usb	clk_adc
System PLL	CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS	CLOCKS_CLK_USB_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS	CLOCKS_CLK_ADC_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS
GPIO in 0	CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_CLKSRC_GPIN0	CLOCKS_CLK_USB_CTRL_AUXSRC_VALUE_CLKSRC_GPIN0	CLOCKS_CLK_ADC_CTRL_AUXSRC_VALUE_CLKSRC_GPIN0
GPIO in 1	CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_CLKSRC_GPIN1	CLOCKS_CLK_USB_CTRL_AUXSRC_VALUE_CLKSRC_GPIN1	CLOCKS_CLK_ADC_CTRL_AUXSRC_VALUE_CLKSRC_GPIN1
USB PLL	CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_CLKSRC_PLL_USB	CLOCKS_CLK_USB_CTRL_AUXSRC_VALUE_CLKSRC_PLL_USB	CLOCKS_CLK_ADC_CTRL_AUXSRC_VALUE_CLKSRC_PLL_USB
ROSC	CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_ROSC_CLKSRC_PH	CLOCKS_CLK_USB_CTRL_AUXSRC_VALUE_ROSC_CLKSRC_PH	CLOCKS_CLK_ADC_CTRL_AUXSRC_VALUE_ROSC_CLKSRC_PH
XOSC	CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_XOSC_CLKSRC	CLOCKS_CLK_USB_CTRL_AUXSRC_VALUE_XOSC_CLKSRC	CLOCKS_CLK_ADC_CTRL_AUXSRC_VALUE_XOSC_CLKSRC
System clock	CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_CLK_SYS		

Aux Source	clk_rtc
System PLL	CLOCKS_CLK_RTC_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS
GPIO in 0	CLOCKS_CLK_RTC_CTRL_AUXSRC_VALUE_CLKSRC_GPIN0
GPIO in 1	CLOCKS_CLK_RTC_CTRL_AUXSRC_VALUE_CLKSRC_GPIN1
USB PLL	CLOCKS_CLK_RTC_CTRL_AUXSRC_VALUE_CLKSRC_PLL_USB
ROSC	CLOCKS_CLK_RTC_CTRL_AUXSRC_VALUE_ROSC_CLKSRC_PH

Aux Source	clk_rtc
XOSC	CLOCKS_CLK_RTC_CTRL_AUXSRC_VALUE_XOSC_CLKSRC

Example// hello_48MHz.c

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/pll.h"
#include "hardware/clocks.h"
#include "hardware/structs/pll.h"
#include "hardware/structs/clocks.h"

void measure_freqs(void) {
    uint f_pll_sys = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_PLL_SYS_CLKSRC_PRIMARY);
    uint f_pll_usb = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_PLL_USB_CLKSRC_PRIMARY);
    uint f_rosc = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_ROSC_CLKSRC);
    uint f_clk_sys = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_SYS);
    uint f_clk_peri = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_PERI);
    uint f_clk_usb = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_USB);
    uint f_clk_adc = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_ADC);
    uint f_clk_rtc = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_RTC);

    printf("pll_sys  = %dkHz\n", f_pll_sys);
    printf("pll_usb  = %dkHz\n", f_pll_usb);
    printf("rosc     = %dkHz\n", f_rosc);
    printf("clk_sys  = %dkHz\n", f_clk_sys);
    printf("clk_peri = %dkHz\n", f_clk_peri);
    printf("clk_usb  = %dkHz\n", f_clk_usb);
    printf("clk_adc  = %dkHz\n", f_clk_adc);
    printf("clk_rtc  = %dkHz\n", f_clk_rtc);

    // Can't measure clk_ref / xosc as it is the ref
}

int main() {
    stdio_init_all();

    printf("Hello, world!\n");

    measure_freqs();

    // Change clk_sys to be 48MHz. The simplest way is to take this from PLL_USB
    // which has a source frequency of 48MHz
    clock_configure(clk_sys,
                    CLOCKS_CLK_SYS_CTRL_SRC_VALUE_CLKSRC_CLK_SYS_AUX,
                    CLOCKS_CLK_SYS_CTRL_AUXSRC_VALUE_CLKSRC_PLL_USB,
                    48 * MHZ,
                    48 * MHZ);

    // Turn off PLL sys for good measure
    pll_deinit(pll_sys);

    // CLK peri is clocked from clk_sys so need to change clk_peri's freq
    clock_configure(clk_peri,
                    0,
                    CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_CLK_SYS,
                    48 * MHZ,
                    48 * MHZ);

    // Re init uart now that clk_peri has changed
    stdio_init_all();

    measure_freqs();
    printf("Hello, 48MHz");

    return 0;
}
```

3.5.5. Function Documentation

3.5.5.1. clock_configure

```
bool clock_configure (enum clock_index clk_index,
       uint32_t src,
       uint32_t auxsrc,
       uint32_t src_freq,
       uint32_t freq)
```

Configure the specified clock.

See the tables in the description for details on the possible values for clock sources.

Parameters

- `clk_index` The clock to configure
- `src` The main clock source, can be 0.
- `auxsrc` The auxiliary clock source, which depends on which clock is being set. Can be 0
- `src_freq` Frequency of the input clock source
- `freq` Requested frequency

3.5.5.2. clock_configure_gpin

```
bool clock_configure_gpin (enum clock_index clk_index,
       uint gpio,
       uint32_t src_freq,
       uint32_t freq)
```

Configure a clock to come from a gpio input.

Parameters

- `clk_index` The clock to configure
- `gpio` The GPIO pin to run the clock from. Valid GPIOs are: 20 and 22.
- `src_freq` Frequency of the input clock source
- `freq` Requested frequency

3.5.5.3. clock_get_hz

```
uint32_t clock_get_hz (enum clock_index clk_index)
```

Get the current frequency of the specified clock.

Parameters

- `clk_index` Clock

Returns

- Clock frequency in Hz

3.5.5.4. clock_gpio_init

```
void clock_gpio_init (uint gpio,
       uint src,
       uint div)
```

Output an optionally divided clock to the specified gpio pin.

Parameters

- `gpio` The GPIO pin to output the clock to. Valid GPIOs are: 21, 23, 24, 26. These GPIOs are connected to the GPOUT0-3 clock generators.
- `src` The source clock. See the register field CLOCKS_CLK_GPOUT0_CTRL_AUXSRC for a full list. The list is the same for each GPOUT clock generator.
- `div` The amount to divide the source clock by. This is useful to not overwhelm the GPIO pin with a fast clock.

3.5.5.5. clock_set_reported_hz

```
void clock_set_reported_hz (enum clock_index clk_index,
       uint hz)
```

Set the "current frequency" of the clock as reported by clock_get_hz without actually changing the clock.

See also

- clock_get_hz

3.5.5.6. clock_stop

```
void clock_stop (enum clock_index clk_index)
```

Stop the specified clock.

Parameters

- `clk_index` The clock to stop

3.5.5.7. clocks_enable_resus

```
void clocks_enable_resus (resus_callback_t resus_callback)
```

Enable the resus function. Restarts clk_sys if it is accidentally stopped.

The resuscitate function will restart the system clock if it falls below a certain speed (or stops). This could happen if the clock source the system clock is running from stops. For example if a PLL is stopped.

Parameters

- `resus_callback` a function pointer provided by the user to call if a resus event happens.

3.5.5.8. clocks_init

```
void clocks_init ()
```

Initialise the clock hardware.

Must be called before any other clock function.

3.5.5.9. frequency_count_khz

`uint32_t frequency_count_khz (uint src)`

Measure a clocks frequency using the Frequency counter.

Uses the inbuilt frequency counter to measure the specified clocks frequency. Currently, this function is accurate to +-1KHz. See the datasheet for more details.

3.6. hardware_divider

Low-level hardware-divider access. More…

3.6.1. Functions

- `static void hw_divider_divmod_s32_start (int32_t a, int32_t b)`
  Start a signed asynchronous divide. More…
- `static void hw_divider_divmod_u32_start (uint32_t a, uint32_t b)`
  Start an unsigned asynchronous divide. More…
- `static void hw_divider_wait_ready ()`
  Wait for a divide to complete. More…
- `static divmod_result_t hw_divider_result_nowait ()`
  Return result of HW divide, nowait. More…
- `static divmod_result_t hw_divider_result_wait ()`
  Return result of last asynchronous HW divide. More…
- `static uint32_t hw_divider_u32_quotient_wait ()`
  Return result of last asynchronous HW divide, unsigned quotient only. More…
- `static int32_t hw_divider_s32_quotient_wait ()`
  Return result of last asynchronous HW divide, signed quotient only. More…
- `static uint32_t hw_divider_u32_remainder_wait ()`
  Return result of last asynchronous HW divide, unsigned remainder only. More…
- `static int32_t hw_divider_s32_remainder_wait ()`
  Return result of last asynchronous HW divide, signed remainder only. More…
- `divmod_result_t hw_divider_divmod_s32 (int32_t a, int32_t b)`
  Do a signed HW divide and wait for result. More…
- `divmod_result_t hw_divider_divmod_u32 (uint32_t a, uint32_t b)`
  Do an unsigned HW divide and wait for result. More…
- `static uint32_t to_quotient_u32 (divmod_result_t r)`
  Efficient extraction of unsigned quotient from 32p32 fixed point. More…
- `static int32_t to_quotient_s32 (divmod_result_t r)`
  Efficient extraction of signed quotient from 32p32 fixed point. More…
- `static uint32_t to_remainder_u32 (divmod_result_t r)`
  Efficient extraction of unsigned remainder from 32p32 fixed point. More…
- `static int32_t to_remainder_s32 (divmod_result_t r)`
  Efficient extraction of signed remainder from 32p32 fixed point. More…
- `static uint32_t hw_divider_u32_quotient (uint32_t a, uint32_t b)`
  Do an unsigned HW divide, wait for result, return quotient. More…

- static uint32_t hw_divider_u32_remainder (uint32_t a, uint32_t b)
  Do an unsigned HW divide, wait for result, return remainder. More…

- static int32_t hw_divider_quotient_s32 (int32_t a, int32_t b)
  Do a signed HW divide, wait for result, return quotient. More…

- static int32_t hw_divider_remainder_s32 (int32_t a, int32_t b)
  Do a signed HW divide, wait for result, return remainder. More…

- static void hw_divider_pause ()
  Pause for exact amount of time needed for a asynchronous divide to complete.

- static uint32_t hw_divider_u32_quotient_inlined (uint32_t a, uint32_t b)
  Do a hardware unsigned HW divide, wait for result, return quotient. More…

- static uint32_t hw_divider_u32_remainder_inlined (uint32_t a, uint32_t b)
  Do a hardware unsigned HW divide, wait for result, return remainder. More…

- static int32_t hw_divider_s32_quotient_inlined (int32_t a, int32_t b)
  Do a hardware signed HW divide, wait for result, return quotient. More…

- static int32_t hw_divider_s32_remainder_inlined (int32_t a, int32_t b)
  Do a hardware signed HW divide, wait for result, return remainder. More…

- void hw_divider_save_state (hw_divider_state_t *dest)
  Save the calling cores hardware divider state. More…

- void hw_divider_restore_state (hw_divider_state_t *src)
  Load a saved hardware divider state into the current cor'es hardware divider. More…

3.6.2. Detailed Description

Low-level hardware-divider access.

The SIO contains an 8-cycle signed/unsigned divide/modulo circuit, per core. Calculation is started by writing a dividend and divisor to the two argument registers, DIVIDEND and DIVISOR. The divider calculates the quotient / and remainder % of this division over the next 8 cycles, and on the 9th cycle the results can be read from the two result registers DIV_QUOTIENT and DIV_REMAINDER. A 'ready' bit in register DIV_CSR can be polled to wait for the calculation to complete, or software can insert a fixed 8-cycle delay

This header provides low level macros and inline functions for accessing the hardware dividers directly, and perhaps most usefully performing asynchronous divides. These functions however do not follow the regular Pico SDK conventions for saving/restoring the divider state, so are not generally safe to call from interrupt handlers

The pico_divider library provides a more user friendly set of APIs over the divider (and support for 64 bit divides), and of course by default regular C language integer divisions are redirected through that library, meaning you can just use C level / and % operators and gain the benefits of the fast hardware divider.

See also

- pico_divider

Example

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/divider.h"

// tag::hello_divider[]
int main() {
    stdio_init_all();
    printf("Hello, divider!\n");

    // This is the basic hardware divider function
```

```c
    int32_t dividend = 123456;
    int32_t divisor = -321;
    divmod_result_t result = hw_divider_divmod_s32(dividend, divisor);

    printf("%d/%d = %d remainder %d\n", dividend, divisor, to_quotient_s32(result), to_remainder_s32(result));

    // Is it right?

    printf("Working backwards! Result %d should equal %d!\n\n",
           to_quotient_s32(result) * divisor + to_remainder_s32(result), dividend);

    // This is the recommended unsigned fast divider for general use.
    int32_t udividend = 123456;
    int32_t udivisor = 321;
    divmod_result_t uresult = hw_divider_divmod_u32(udividend, udivisor);

    printf("%d/%d = %d remainder %d\n", udividend, udivisor, to_quotient_u32(uresult), to_remainder_u32(uresult));

    // Is it right?

    printf("Working backwards! Result %d should equal %d!\n\n",
           to_quotient_u32(result) * divisor + to_remainder_u32(result), dividend);

    // You can also do divides asynchronously. Divides will be complete after 8 cyles.

    hw_divider_divmod_s32_start(dividend, divisor);

    // Do something for 8 cycles!

    // In this example, our results function will wait for completion.
    // Use hw_divider_result_nowait() if you don't want to wait, but are sure you have delayed at least 8 cycles

    result = hw_divider_result_wait();

    printf("Async result %d/%d = %d remainder %d\n", dividend, divisor, to_quotient_s32(result),
           to_remainder_s32(result));

    // For a really fast divide, you can use the inlined versions... the / involves a function call as / always does
    // when using the ARM AEABI, so if you really want the best performance use the inlined versions.
    // Note that the / operator function DOES use the hardware divider by default, although you can change
    // that behavior by calling pico_set_divider_implementation in the cmake build for your target.
    printf("%d / %d = (by operator %d) (inlined %d)\n", dividend, divisor,
           dividend / divisor, hw_divider_s32_quotient_inlined(dividend, divisor));

    // Note however you must manually save/restore the divider state if you call the inlined methods from within an IRQ
    // handler.
    hw_divider_state_t state;
    hw_divider_divmod_s32_start(dividend, divisor);
    hw_divider_save_state(&state);

    hw_divider_divmod_s32_start(123, 7);
    printf("inner %d / %d = %d\n", 123, 7, hw_divider_s32_quotient_wait());

    hw_divider_restore_state(&state);
    int32_t tmp = hw_divider_s32_quotient_wait();
    printf("outer divide %d / %d = %d\n", dividend, divisor, tmp);
    return 0;
}
// end::hello_divider[]
```

3.6.3. Function Documentation

3.6.3.1. hw_divider_divmod_s32

```
divmod_result_t hw_divider_divmod_s32 (int32_t a,
       int32_t b)
```

Do a signed HW divide and wait for result.

Divide `a` by `b`, wait for calculation to complete, return result as a fixed point 32p32 value.

Parameters

- `a` The dividend
- `b` The divisor

Returns

- Results of divide as a 32p32 fixed point value.

3.6.3.2. hw_divider_divmod_s32_start

```
static void hw_divider_divmod_s32_start (int32_t a,
       int32_t b)
```

Start a signed asynchronous divide.

Start a divide of the specified signed parameters. You should wait for 8 cycles (__div_pause()) or wait for the ready bit to be set (`hw_divider_wait_ready()`) prior to reading the results.

Parameters

- `a` The dividend
- `b` The divisor

3.6.3.3. hw_divider_divmod_u32

```
divmod_result_t hw_divider_divmod_u32 (uint32_t a,
       uint32_t b)
```

Do an unsigned HW divide and wait for result.

Divide `a` by `b`, wait for calculation to complete, return result as a fixed point 32p32 value.

Parameters

- `a` The dividend
- `b` The divisor

Returns

- Results of divide as a 32p32 fixed point value.

3.6.3.4. hw_divider_divmod_u32_start

```
static void hw_divider_divmod_u32_start (uint32_t a,
       uint32_t b)
```

Start an unsigned asynchronous divide.

Start a divide of the specified unsigned parameters. You should wait for 8 cycles (__div_pause()) or wait for the ready bit to be set (`hw_divider_wait_ready()`) prior to reading the results.

Parameters

- `a` The dividend
- `b` The divisor

3.6.3.5. hw_divider_pause

```
static void hw_divider_pause ()
```

Pause for exact amount of time needed for a asynchronous divide to complete.

3.6.3.6. hw_divider_quotient_s32

```
static int32_t hw_divider_quotient_s32 (int32_t a,
       int32_t b)
```

Do a signed HW divide, wait for result, return quotient.

Divide `a` by `b`, wait for calculation to complete, return quotient.

Parameters

- `a` The dividend
- `b` The divisor

Returns

- Quotient results of the divide

3.6.3.7. hw_divider_remainder_s32

```
static int32_t hw_divider_remainder_s32 (int32_t a,
       int32_t b)
```

Do a signed HW divide, wait for result, return remainder.

Divide `a` by `b`, wait for calculation to complete, return remainder.

Parameters

- `a` The dividend
- `b` The divisor

Returns

- Remainder results of the divide

3.6.3.8. hw_divider_restore_state

```
void hw_divider_restore_state (hw_divider_state_t *src)
```

Load a saved hardware divider state into the current cor’es hardware divider.

Copy the passed hardware divider state into the hardware divider.

Parameters

- `src` the location to load the divider state from

3.6.3.9. hw_divider_result_nowait

```
static divmod_result_t hw_divider_result_nowait ()
```

Return result of HW divide, nowait.

Returns

- Current result. Most significant 32 bits are the remainder, lower 32 bits are the quotient.

3.6.3.10. hw_divider_result_wait

```
static divmod_result_t hw_divider_result_wait ()
```

Return result of last asynchronous HW divide.

This function waits for the result to be ready by calling `hw_divider_wait_ready()`.

Returns

- Current result. Most significant 32 bits are the remainder, lower 32 bits are the quotient.

3.6.3.11. hw_divider_s32_quotient_inlined

```
static int32_t hw_divider_s32_quotient_inlined (int32_t a,
       int32_t b)
```

Do a hardware signed HW divide, wait for result, return quotient.

Divide `a` by `b`, wait for calculation to complete, return quotient.

Parameters

- `a` The dividend
- `b` The divisor

Returns

- Quotient result of the divide

3.6.3.12. hw_divider_s32_quotient_wait

```
static int32_t hw_divider_s32_quotient_wait ()
```

Return result of last asynchronous HW divide, signed quotient only.

This function waits for the result to be ready by calling `hw_divider_wait_ready()`.

Returns

- Current signed quotient result.

3.6.3.13. hw_divider_s32_remainder_inlined

```
static int32_t hw_divider_s32_remainder_inlined (int32_t a,
       int32_t b)
```

Do a hardware signed HW divide, wait for result, return remainder.

Divide `a` by `b`, wait for calculation to complete, return remainder.

Parameters

- `a` The dividend
- `b` The divisor

Returns

- Remainder result of the divide

3.6.3.14. hw_divider_s32_remainder_wait

```
static int32_t hw_divider_s32_remainder_wait ()
```

Return result of last asynchronous HW divide, signed remainder only.

This function waits for the result to be ready by calling `hw_divider_wait_ready()`.

Returns

- Current remainder results.

3.6.3.15. hw_divider_save_state

```
void hw_divider_save_state (hw_divider_state_t *dest)
```

Save the calling cores hardware divider state.

Copy the current core's hardware divider state into the provided structure. This method waits for the divider results to be stable, then copies them to memory. They can be restored via `hw_divider_restore_state()`

Parameters

- `dest` the location to store the divider state

3.6.3.16. hw_divider_u32_quotient

```
static uint32_t hw_divider_u32_quotient (uint32_t a,
       uint32_t b)
```

Do an unsigned HW divide, wait for result, return quotient.

Divide `a` by `b`, wait for calculation to complete, return quotient.

Parameters

- `a` The dividend
- `b` The divisor

Returns

- Quotient results of the divide

3.6.3.17. hw_divider_u32_quotient_inlined

```
static uint32_t hw_divider_u32_quotient_inlined (uint32_t a,
       uint32_t b)
```

Do a hardware unsigned HW divide, wait for result, return quotient.

Divide `a` by `b`, wait for calculation to complete, return quotient.

Parameters

- a The dividend
- b The divisor

Returns

- Quotient result of the divide

3.6.3.18. hw_divider_u32_quotient_wait

```
static uint32_t hw_divider_u32_quotient_wait ()
```

Return result of last asynchronous HW divide, unsigned quotient only.

This function waits for the result to be ready by calling `hw_divider_wait_ready()`.

Returns

- Current unsigned quotient result.

3.6.3.19. hw_divider_u32_remainder

```
static uint32_t hw_divider_u32_remainder (uint32_t a,
        uint32_t b)
```

Do an unsigned HW divide, wait for result, return remainder.

Divide `a` by `b`, wait for calculation to complete, return remainder.

Parameters

- a The dividend
- b The divisor

Returns

- Remainder results of the divide

3.6.3.20. hw_divider_u32_remainder_inlined

```
static uint32_t hw_divider_u32_remainder_inlined (uint32_t a,
        uint32_t b)
```

Do a hardware unsigned HW divide, wait for result, return remainder.

Divide `a` by `b`, wait for calculation to complete, return remainder.

Parameters

- a The dividend
- b The divisor

Returns

- Remainder result of the dividearm

3.6.3.21. hw_divider_u32_remainder_wait

```
static uint32_t hw_divider_u32_remainder_wait ()
```

Return result of last asynchronous HW divide, unsigned remainder only.

This function waits for the result to be ready by calling `hw_divider_wait_ready()`.

Returns

- Current unsigned remainder result.

3.6.3.22. hw_divider_wait_ready

`static void hw_divider_wait_ready ()`

Wait for a divide to complete.

Wait for a divide to complete

3.6.3.23. to_quotient_s32

`static int32_t to_quotient_s32 (divmod_result_t r)`

Efficient extraction of signed quotient from 32p32 fixed point.

Parameters

- `r` 32p32 fixed point value.

Returns

- Unsigned quotient

3.6.3.24. to_quotient_u32

`static uint32_t to_quotient_u32 (divmod_result_t r)`

Efficient extraction of unsigned quotient from 32p32 fixed point.

Parameters

- `r` 32p32 fixed point value.

Returns

- Unsigned quotient

3.6.3.25. to_remainder_s32

`static int32_t to_remainder_s32 (divmod_result_t r)`

Efficient extraction of signed remainder from 32p32 fixed point.

Parameters

- `r` 32p32 fixed point value.

Returns

- Signed remainder

3.6.3.26. to_remainder_u32

`static uint32_t to_remainder_u32 (divmod_result_t r)`

Efficient extraction of unsigned remainder from 32p32 fixed point.

Parameters

- `r` 32p32 fixed point value.

Returns

- Unsigned remainder

3.7. hardware_dma

DMA Controller API. More…

3.7.1. Modules

- `channel_config`
  DMA channel configuration.

3.7.2. Enumerations

- `enum dma_channel_transfer_size { DMA_SIZE_8 = 0, DMA_SIZE_16 = 1, DMA_SIZE_32 = 2 }`
  Enumeration of available DMA channel transfer sizes. More…

3.7.3. Functions

- `void dma_channel_claim (uint channel)`
  Mark a dma channel as used. More…
- `void dma_claim_mask (uint32_t channel_mask)`
  Mark multiple dma channels as used. More…
- `void dma_channel_unclaim (uint channel)`
  Mark a dma channel as no longer used. More…
- `int dma_claim_unused_channel (bool required)`
  Claim a free dma channel. More…
- `static void dma_channel_set_config (uint channel, const dma_channel_config *config, bool trigger)`
  Set a channel configuration. More…
- `static void dma_channel_set_read_addr (uint channel, const volatile void *read_addr, bool trigger)`
  Set the DMA initial read address. More…
- `static void dma_channel_set_write_addr (uint channel, volatile void *write_addr, bool trigger)`
  Set the DMA initial read address. More…
- `static void dma_channel_set_trans_count (uint channel, uint32_t trans_count, bool trigger)`
  Set the number of bus transfers the channel will do. More…
- `static void dma_channel_configure (uint channel, const dma_channel_config *config, volatile void *write_addr, const volatile void *read_addr, uint transfer_count, bool trigger)`
  Configure all DMA parameters and optionally start transfer. More…
- `static void dma_channel_transfer_from_buffer_now (uint channel, void *read_addr, uint32_t transfer_count)`
  Start a DMA transfer from a buffer immediately. More…
- `static void dma_channel_transfer_to_buffer_now (uint channel, void *write_addr, uint32_t transfer_count)`
  Start a DMA transfer to a buffer immediately. More…
- `static void dma_start_channel_mask (uint32_t chan_mask)`
  Start one or more channels simultaneously. More…
- `static void dma_channel_start (uint channel)`
  Start a single DMA channel. More…

- `static void dma_channel_abort (uint channel)`
  Stop a DMA transfer. More...
- `static void dma_channel_set_irq0_enabled (uint channel, bool enabled)`
  Enable single DMA channel interrupt 0. More...
- `static void dma_set_irq0_channel_mask_enabled (uint32_t channel_mask, bool enabled)`
  Enable multiple DMA channels interrupt 0. More...
- `static void dma_channel_set_irq1_enabled (uint channel, bool enabled)`
  Enable single DMA channel interrupt 1. More...
- `static void dma_set_irq1_channel_mask_enabled (uint32_t channel_mask, bool enabled)`
  Enable multiple DMA channels interrupt 0. More...
- `static bool dma_channel_is_busy (uint channel)`
  Check if DMA channel is busy. More...
- `static void dma_channel_wait_for_finish_blocking (uint channel)`
  Wait for a DMA channel transfer to complete. More...
- `static void dma_sniffer_enable (uint channel, uint mode, bool force_channel_enable)`
  Enable the DMA sniffing targeting the specified channel. More...
- `static void dma_sniffer_set_byte_swap_enabled (bool swap)`
  Enable the Sniffer byte swap function. More...
- `static void dma_sniffer_disable ()`
  Disable the DMA sniffer.
- `void pio_sm_claim (PIO pio, uint sm)`
  Mark a state machine as used. More...
- `void pio_claim_sm_mask (PIO pio, uint sm_mask)`
  Mark multiple state machines as used. More...
- `void pio_sm_unclaim (PIO pio, uint sm)`
  Mark a state machine as no longer used. More...
- `int pio_claim_unused_sm (PIO pio, bool required)`
  Claim a free state machine on a PIO instance. More...

3.7.4. Detailed Description

DMA Controller API.

The RP2040 Direct Memory Access (DMA) master performs bulk data transfers on a processor's behalf. This leaves processors free to attend to other tasks, or enter low-power sleep states. The data throughput of the DMA is also significantly higher than one of RP2040's processors.

The DMA can perform one read access and one write access, up to 32 bits in size, every clock cycle. There are 12 independent channels, which each supervise a sequence of bus transfers, usually in one of the following scenarios:

- Memory to peripheral
- Peripheral to memory
- Memory to memory

3.7.5. Function Documentation

3.7.5.1. dma_channel_abort

```
static void dma_channel_abort (uint channel)
```

Stop a DMA transfer.

Function will only return once the DMA has stopped.

Parameters

- `channel` DMA channel

3.7.5.2. dma_channel_claim

```
void dma_channel_claim (uint channel)
```

Mark a dma channel as used.

Method for cooperative claiming of hardware. Will cause a panic if the channel is already claimed. Use of this method by libraries detects accidental configurations that would fail in unpredictable ways.

Parameters

- `channel` the dma channel

3.7.5.3. dma_channel_configure

```
static void dma_channel_configure (uint channel,
       const dma_channel_config *config,
       volatile void *write_addr,
       const volatile void *read_addr,
       uint transfer_count,
       bool trigger)
```

Configure all DMA parameters and optionally start transfer.

Parameters

- `channel` DMA channel
- `config` Pointer to DMA config structure
- `write_addr` Initial write address
- `read_addr` Initial read address
- `transfer_count` Number of transfers to perform
- `trigger` True to start the transfer immediately

3.7.5.4. dma_channel_is_busy

```
static bool dma_channel_is_busy (uint channel)
```

Check if DMA channel is busy.

Parameters

- `channel` DMA channel

Returns

- true if the channel is currently busy

3.7.5.5. dma_channel_set_config

```
static void dma_channel_set_config (uint channel,
       const dma_channel_config *config,
       bool trigger)
```

Set a channel configuration.

Parameters

- `channel` DMA channel
- `config` Pointer to a config structure with required configuration
- `trigger` True to trigger the transfer immediately

3.7.5.6. dma_channel_set_irq0_enabled

```
static void dma_channel_set_irq0_enabled (uint channel,
       bool enabled)
```

Enable single DMA channel interrupt 0.

Parameters

- `channel` DMA channel
- `enabled` true to enable interrupt 0 on specified channel, false to disable.

3.7.5.7. dma_channel_set_irq1_enabled

```
static void dma_channel_set_irq1_enabled (uint channel,
       bool enabled)
```

Enable single DMA channel interrupt 1.

Parameters

- `channel` DMA channel
- `enabled` true to enable interrupt 1 on specified channel, false to disable.

3.7.5.8. dma_channel_set_read_addr

```
static void dma_channel_set_read_addr (uint channel,
       const volatile void *read_addr,
       bool trigger)
```

Set the DMA initial read address.

Parameters

- `channel` DMA channel
- `read_addr` Initial read address of transfer.
- `trigger` True to start the transfer immediately

3.7.5.9. dma_channel_set_trans_count

```
static void dma_channel_set_trans_count (uint channel,
       uint32_t trans_count,
       bool trigger)
```

Set the number of bus transfers the channel will do.

Parameters

- `channel` DMA channel
- `trans_count` The number of transfers (not NOT bytes, see channel_config_set_transfer_data_size)
- `trigger` True to start the transfer immediately

3.7.5.10. dma_channel_set_write_addr

```
static void dma_channel_set_write_addr (uint channel,
       volatile void *write_addr,
       bool trigger)
```

Set the DMA initial read address.

Parameters

- `channel` DMA channel
- `write_addr` Initial write address of transfer.
- `trigger` True to start the transfer immediately

3.7.5.11. dma_channel_start

```
static void dma_channel_start (uint channel)
```

Start a single DMA channel.

Parameters

- `channel` DMA channel

3.7.5.12. dma_channel_transfer_from_buffer_now

```
static void dma_channel_transfer_from_buffer_now (uint channel,
       void *read_addr,
       uint32_t transfer_count)
```

Start a DMA transfer from a buffer immediately.

Parameters

- `channel` DMA channel
- `read_addr` Sets the initial read address
- `transfer_count` Number of transfers to make. Not bytes, but the number of transfers of channel_config_set_transfer_data_size() to be sent.

3.7.5.13. dma_channel_transfer_to_buffer_now

```
static void dma_channel_transfer_to_buffer_now (uint channel,
       void *write_addr,
       uint32_t transfer_count)
```

Start a DMA transfer to a buffer immediately.

Parameters

- `channel` DMA channel

- `write_addr` Sets the initial write address
- `transfer_count` Number of transfers to make. Not bytes, but the number of transfers of channel_config_set_transfer_data_size() to be sent.

3.7.5.14. dma_channel_unclaim

```
void dma_channel_unclaim (uint channel)
```

Mark a dma channel as no longer used.

Method for cooperative claiming of hardware.

Parameters

- `channel` the dma channel to release

3.7.5.15. dma_channel_wait_for_finish_blocking

```
static void dma_channel_wait_for_finish_blocking (uint channel)
```

Wait for a DMA channel transfer to complete.

Parameters

- `channel` DMA channel

3.7.5.16. dma_claim_mask

```
void dma_claim_mask (uint32_t channel_mask)
```

Mark multiple dma channels as used.

Method for cooperative claiming of hardware. Will cause a panic if any of the channels are already claimed. Use of this method by libraries detects accidental configurations that would fail in unpredictable ways.

Parameters

- `channel_mask` Bitfield of all required channels to claim (bit 0 == channel 0, bit 1 == channel 1 etc)

3.7.5.17. dma_claim_unused_channel

```
int dma_claim_unused_channel (bool required)
```

Claim a free dma channel.

Parameters

- `required` if true the function will panic if none are available

Returns

- the dma channel number or -1 if required was false, and none were free

3.7.5.18. dma_set_irq0_channel_mask_enabled

```
static void dma_set_irq0_channel_mask_enabled (uint32_t channel_mask,
        bool enabled)
```

Enable multiple DMA channels interrupt 0.

Parameters

- channel_mask Bitmask of all the channels to enable/disable. Channel 0 = bit 0, channel 1 = bit 1 etc.
- enabled true to enable all the interrupts specified in the mask, false to disable all the interrupts specified in the mask.

3.7.5.19. dma_set_irq1_channel_mask_enabled

```
static void dma_set_irq1_channel_mask_enabled (uint32_t channel_mask,
       bool enabled)
```

Enable multiple DMA channels interrupt 0.

Parameters

- channel_mask Bitmask of all the channels to enable/disable. Channel 0 = bit 0, channel 1 = bit 1 etc.
- enabled true to enable all the interrupts specified in the mask, false to disable all the interrupts specified in the mask.

3.7.5.20. dma_sniffer_disable

```
static void dma_sniffer_disable ()
```

Disable the DMA sniffer.

3.7.5.21. dma_sniffer_enable

```
static void dma_sniffer_enable (uint channel,
       uint mode,
       bool force_channel_enable)
```

Enable the DMA sniffing targeting the specified channel.

The mode can be one of the following:

Mode	Function
0x0	Calculate a CRC-32 (IEEE802.3 polynomial)
0x1	Calculate a CRC-32 (IEEE802.3 polynomial) with bit reversed data
0x2	Calculate a CRC-16-CCITT
0x3	Calculate a CRC-16-CCITT with bit reversed data
0xe	XOR reduction over all data. == 1 if the total 1 population count is odd.
0xf	Calculate a simple 32-bit checksum (addition with a 32 bit accumulator)

Parameters

- channel DMA channel
- mode See description
- force_channel_enable Set true to also turn on sniffing in the channel configuration (this is usually what you want, but sometimes you might have a chain DMA with only certain segments of the chain sniffed, in which case you might pass false).

3.7.5.22. dma_sniffer_set_byte_swap_enabled

```
static void dma_sniffer_set_byte_swap_enabled (bool swap)
```

Enable the Sniffer byte swap function.

Locally perform a byte reverse on the sniffed data, before feeding into checksum.

Note that the sniff hardware is downstream of the DMA channel byteswap performed in the read master: if `channel_config_set_bswap()` and `dma_sniffer_set_byte_swap_enabled()` are both enabled, their effects cancel from the sniffer's point of view.

Parameters

- `swap` Set true to enable byte swapping

3.7.5.23. dma_start_channel_mask

```
static void dma_start_channel_mask (uint32_t chan_mask)
```

Start one or more channels simultaneously.

Parameters

- `chan_mask` Bitmask of all the channels requiring starting. Channel 0 = bit 0, channel 1 = bit 1 etc.

3.7.5.24. pio_claim_sm_mask

```
void pio_claim_sm_mask (PIO pio,
       uint sm_mask)
```

Mark multiple state machines as used.

Method for cooperative claiming of hardware. Will cause a panic if any of the state machines are already claimed. Use of this method by libraries detects accidental configurations that would fail in unpredictable ways.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm_mask` Mask of state machine indexes

3.7.5.25. pio_claim_unused_sm

```
int pio_claim_unused_sm (PIO pio,
       bool required)
```

Claim a free state machine on a PIO instance.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `required` if true the function will panic if none are available

Returns

- the state machine index or -1 if required was false, and none were free

3.7.5.26. pio_sm_claim

```
void pio_sm_claim (PIO pio,
       uint sm)
```

Mark a state machine as used.

Method for cooperative claiming of hardware. Will cause a panic if the state machine is already claimed. Use of this method by libraries detects accidental configurations that would fail in unpredictable ways.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)

3.7.5.27. pio_sm_unclaim

```
void pio_sm_unclaim (PIO pio,
      uint sm)
```

Mark a state machine as no longer used.

Method for cooperative claiming of hardware.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)

3.8. hardware_flash

Low level flash programming and erase API. More…

3.8.1. Functions

- `void flash_range_erase (uint32_t flash_offs, size_t count)`
  Erase areas of flash. More…
- `void flash_range_program (uint32_t flash_offs, const uint8_t *data, size_t count)`
  Program flash. More…

3.8.2. Detailed Description

Low level flash programming and erase API.

Note these functions are *unsafe* if you have two cores concurrently executing from flash. In this case you must perform your own synchronisation to make sure no XIP accesses take place during flash programming.

If PICO_NO_FLASH=1 is not defined (i.e. if the program is built to run from flash) then these functions will make a static copy of the second stage bootloader in SRAM, and use this to reenter execute-in-place mode after programming or erasing flash, so that they can safely be called from flash-resident code.

Example

```
#include <stdio.h>
#include <stdlib.h>

#include "pico/stdlib.h"
#include "hardware/flash.h"

// We're going to erase and reprogram a region 256k from the start of flash.
// Once done, we can access this at XIP_BASE + 256k.
#define FLASH_TARGET_OFFSET (256 * 1024)

```

```c
const uint8_t *flash_target_contents = (const uint8_t *) (XIP_BASE + FLASH_TARGET_OFFSET);

void print_buf(const uint8_t *buf, size_t len) {
    for (size_t i = 0; i < len; ++i) {
        printf("%02x", buf[i]);
        if (i % 16 == 15)
            printf("\n");
        else
            printf(" ");
    }
}

int main() {
    stdio_init_all();
    uint8_t random_data[FLASH_PAGE_SIZE];
    for (int i = 0; i < FLASH_PAGE_SIZE; ++i)
        random_data[i] = rand() >> 16;

    printf("Generated random data:\n");
    print_buf(random_data, FLASH_PAGE_SIZE);

    // Note that a whole number of sectors must be erased at a time.
    printf("\nErasing target region...\n");
    flash_range_erase(FLASH_TARGET_OFFSET, FLASH_SECTOR_SIZE);
    printf("Done. Read back target region:\n");
    print_buf(flash_target_contents, FLASH_PAGE_SIZE);

    printf("\nProgramming target region...\n");
    flash_range_program(FLASH_TARGET_OFFSET, random_data, FLASH_PAGE_SIZE);
    printf("Done. Read back target region:\n");
    print_buf(flash_target_contents, FLASH_PAGE_SIZE);

    bool mismatch = false;
    for (int i = 0; i < FLASH_PAGE_SIZE; ++i) {
        if (random_data[i] != flash_target_contents[i])
            mismatch = true;
    }
    if (mismatch)
        printf("Programming failed!\n");
    else
        printf("Programming successful!\n");
}
```

3.8.3. Function Documentation

3.8.3.1. flash_range_erase

```
void flash_range_erase (uint32_t flash_offs,
        size_t count)
```

Erase areas of flash.

Parameters

- `flash_offs` Offset into flash, in bytes, to start the erase. Must be aligned to a 4096-byte flash sector.
- `count` Number of bytes to be erased. Must be a multiple of 4096 bytes (one sector).

3.8.3.2. flash_range_program

```
void flash_range_program (uint32_t flash_offs,
       const uint8_t *data,
       size_t count)
```

Program flash.

Parameters

- `flash_offs` Flash address of the first byte to be programmed. Must be aligned to a 256-byte flash page.
- `data` Pointer to the data to program into flash
- `count` Number of bytes to program. Must be a multiple of 256 bytes (one page).

3.9. hardware_gpio

General Purpose Input/Output (GPIO) API. More…

3.9.1. Enumerations

- `enum gpio_function { GPIO_FUNC_XIP = 0, GPIO_FUNC_SPI = 1, GPIO_FUNC_UART = 2, GPIO_FUNC_I2C = 3, GPIO_FUNC_PWM = 4, GPIO_FUNC_SIO = 5, GPIO_FUNC_PIO0 = 6, GPIO_FUNC_PIO1 = 7, GPIO_FUNC_GPCK = 8, GPIO_FUNC_USB = 9, GPIO_FUNC_NULL = 0xf }`
  GPIO function definitions for use with function select. More…
- `enum gpio_irq_level { GPIO_IRQ_LEVEL_LOW = 0x1u, GPIO_IRQ_LEVEL_HIGH = 0x2u, GPIO_IRQ_EDGE_FALL = 0x4u, GPIO_IRQ_EDGE_RISE = 0x8u }`
  GPIO Interrupt level definitions. More…

3.9.2. Functions

- `void gpio_set_function (uint gpio, enum gpio_function fn)`
  Select GPIO function. More…
- `void gpio_set_pulls (uint gpio, bool up, bool down)`
  Select up and down pulls on specific GPIO. More…
- `static void gpio_pull_up (uint gpio)`
  Set specified GPIO to be pulled up. More…
- `static bool gpio_is_pulled_up (uint gpio)`
  Determine if the specified GPIO is pulled up. More…
- `static void gpio_pull_down (uint gpio)`
  Set specified GPIO to be pulled down. More…
- `static bool gpio_is_pulled_down (uint gpio)`
  Determine if the specified GPIO is pulled down. More…
- `static void gpio_disable_pulls (uint gpio)`
  Disable pulls on specified GPIO. More…
- `void gpio_set_outover (uint gpio, uint value)`
  Set GPIO output override. More…
- `void gpio_set_inover (uint gpio, uint value)`
  Select GPIO input override. More…
- `void gpio_set_oeover (uint gpio, uint value)`

Select GPIO output enable override. More…

- `void gpio_set_input_enabled (uint gpio, bool enabled)`
  Enable GPIO input. More…
- `void gpio_set_irq_enabled (uint gpio, uint32_t events, bool enabled)`
  Enable or disable interrupts for specified GPIO. More…
- `void gpio_set_irq_enabled_with_callback (uint gpio, uint32_t events, bool enabled, gpio_irq_callback_t callback)`
  Enable interrupts for specified GPIO. More…
- `void gpio_set_dormant_irq_enabled (uint gpio, uint32_t events, bool enabled)`
  Enable dormant wake up interrupt for specified GPIO. More…
- `void gpio_acknowledge_irq (uint gpio, uint32_t events)`
  Acknowledge a GPIO interrupt. More…
- `void gpio_init (uint gpio)`
  Initialise a GPIO for (enabled I/O and set func to GPIO_FUNC_SIO) More…
- `void gpio_init_mask (uint gpio_mask)`
  Initialise multiple GPIOs (enabled I/O and set func to GPIO_FUNC_SIO) More…
- `static bool gpio_get (uint gpio)`
  Get state of a single specified GPIO. More…
- `static uint32_t gpio_get_all ()`
  Get raw value of all GPIOs. More…
- `static void gpio_set_mask (uint32_t mask)`
  Drive high every GPIO appearing in mask. More…
- `static void gpio_clr_mask (uint32_t mask)`
  Drive low every GPIO appearing in mask. More…
- `static void gpio_xor_mask (uint32_t mask)`
  Toggle every GPIO appearing in mask. More…
- `static void gpio_put_masked (uint32_t mask, uint32_t value)`
  Drive GPIO high/low depending on parameters. More…
- `static void gpio_put_all (uint32_t value)`
  Drive all pins simultaneously. More…
- `static void gpio_put (uint gpio, bool value)`
  Drive a single GPIO high/low. More…
- `static void gpio_set_dir_out_masked (uint32_t mask)`
  Set a number of GPIOs to output. More…
- `static void gpio_set_dir_in_masked (uint32_t mask)`
  Set a number of GPIOs to input. More…
- `static void gpio_set_dir_masked (uint32_t mask, uint32_t value)`
  Set multiple GPIO directions. More…
- `static void gpio_set_dir_all_bits (uint32_t values)`
  Set direction of all pins simultaneously. More…
- `static void gpio_set_dir (uint gpio, bool out)`
  Set a single GPIO direction. More…
- `static bool gpio_is_dir_out (uint gpio)`
  Check if a specific GPIO direction is OUT. More…
- `static uint gpio_get_dir (uint gpio)`
  Get a specific GPIO direction. More…

3.9.3. Detailed Description

General Purpose Input/Output (GPIO) API.

RP2040 has 36 multi-functional General Purpose Input / Output (GPIO) pins, divided into two banks. In a typical use case, the pins in the QSPI bank (QSPI_SS, QSPI_SCLK and QSPI_SD0 to QSPI_SD3) are used to execute code from an external flash device, leaving the User bank (GPIO0 to GPIO29) for the programmer to use. All GPIOs support digital input and output, but GPIO26 to GPIO29 can also be used as inputs to the chip's Analogue to Digital Converter (ADC). Each GPIO can be controlled directly by software running on the processors, or by a number of other functional blocks.

The function allocated to each GPIO is selected by calling the gpio_set_function function. Not all functions are available on all pins.

Each GPIO can have one function selected at a time. Likewise, each peripheral input (e.g. UART0 RX) should only be selected on one *GPIO* at a time. If the same peripheral input is connected to multiple GPIOs, the peripheral sees the logical OR of these GPIO inputs. Please refer to the datsheet for more information on GPIO function select.

Table 5. Function Select Table

GPIO	F1	F2	F3	F4	F5	F6	F7	F8	F9
0	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1		USB OVCUR DET
1	SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO	PIO0	PIO1		USB VBUS DET
2	SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO	PIO0	PIO1		USB VBUS EN
3	SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO	PIO0	PIO1		USB OVCUR DET
4	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1		USB VBUS DET
5	SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO	PIO0	PIO1		USB VBUS EN
6	SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO	PIO0	PIO1		USB OVCUR DET
7	SPI0 TX	UART1 RTS	I2C1 SCL	PWM3 B	SIO	PIO0	PIO1		USB VBUS DET
8	SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1		USB VBUS EN
9	SPI1 CSn	UART1 RX	I2C0 SCL	PWM4 B	SIO	PIO0	PIO1		USB OVCUR DET
10	SPI1 SCK	UART1 CTS	I2C1 SDA	PWM5 A	SIO	PIO0	PIO1		USB VBUS DET
11	SPI1 TX	UART1 RTS	I2C1 SCL	PWM5 B	SIO	PIO0	PIO1		USB VBUS EN
12	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1		USB OVCUR DET

GPIO	F1	F2	F3	F4	F5	F6	F7	F8	F9
13	SPI1 CSn	UART0 RX	I2C0 SCL	PWM6 B	SIO	PIO0	PIO1		USB VBUS DET
14	SPI1 SCK	UART0 CTS	I2C1 SDA	PWM7 A	SIO	PIO0	PIO1		USB VBUS EN
15	SPI1 TX	UART0 RTS	I2C1 SCL	PWM7 B	SIO	PIO0	PIO1		USB OVCUR DET
16	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1		USB VBUS DET
17	SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO	PIO0	PIO1		USB VBUS EN
18	SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO	PIO0	PIO1		USB OVCUR DET
19	SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO	PIO0	PIO1		USB VBUS DET
20	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1	CLOCK GPIN0	USB VBUS EN
21	SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO	PIO0	PIO1	CLOCK GPOUT0	USB OVCUR DET
22	SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO	PIO0	PIO1	CLOCK GPIN1	USB VBUS DET
23	SPI0 TX	UART1 RTS	I2C1 SCL	PWM3 B	SIO	PIO0	PIO1	CLOCK GPOUT1	USB VBUS EN
24	SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1	CLOCK GPOUT2	USB OVCUR DET
25	SPI1 CSn	UART1 RX	I2C0 SCL	PWM4 B	SIO	PIO0	PIO1	CLOCK GPOUT3	USB VBUS DET
26	SPI1 SCK	UART1 CTS	I2C1 SDA	PWM5 A	SIO	PIO0	PIO1		USB VBUS EN
27	SPI1 TX	UART1 RTS	I2C1 SCL	PWM5 B	SIO	PIO0	PIO1		USB OVCUR DET
28	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1		USB VBUS DET
29	SPI1 CSn	UART0 RX	I2C0 SCL	PWM6 B	SIO	PIO0	PIO1		USB VBUS EN

3.9.4. Function Documentation

3.9.4.1. gpio_acknowledge_irq

```
void gpio_acknowledge_irq (uint gpio,
       uint32_t events)
```

Acknowledge a GPIO interrupt.

Parameters

- `gpio` GPIO number
- `events` Bitmask of events to clear. See gpio_set_irq_enabled for details.

3.9.4.2. gpio_clr_mask

```
static void gpio_clr_mask (uint32_t mask)
```

Drive low every GPIO appearing in mask.

Parameters

- `mask` Bitmask of GPIO values to clear, as bits 0-29

3.9.4.3. gpio_disable_pulls

```
static void gpio_disable_pulls (uint gpio)
```

Disable pulls on specified GPIO.

Parameters

- `gpio` GPIO number

3.9.4.4. gpio_get

```
static bool gpio_get (uint gpio)
```

Get state of a single specified GPIO.

Parameters

- `gpio` GPIO number

Returns

- Current state of the GPIO. 0 for low, non-zero for high

3.9.4.5. gpio_get_all

```
static uint32_t gpio_get_all ()
```

Get raw value of all GPIOs.

Returns

- Bitmask of raw GPIO values, as bits 0-29

3.9.4.6. gpio_get_dir

```
static uint gpio_get_dir (uint gpio)
```

Get a specific GPIO direction.

Parameters

- `gpio` GPIO number

Returns

- 1 for out, 0 for in

3.9.4.7. gpio_init

`void gpio_init (uint gpio)`

Initialise a GPIO for (enabled I/O and set func to GPIO_FUNC_SIO)

Clear the output enable (i.e. set to input) Clear any output value.

Parameters

- `gpio` GPIO number

3.9.4.8. gpio_init_mask

`void gpio_init_mask (uint gpio_mask)`

Initialise multiple GPIOs (enabled I/O and set func to GPIO_FUNC_SIO)

Clear the output enable (i.e. set to input) Clear any output value.

Parameters

- `gpio_mask` Mask with 1 bit per GPIO number to initialize

3.9.4.9. gpio_is_dir_out

`static bool gpio_is_dir_out (uint gpio)`

Check if a specific GPIO direction is OUT.

Parameters

- `gpio` GPIO number

Returns

- true if the direction for the pin is OUT

3.9.4.10. gpio_is_pulled_down

`static bool gpio_is_pulled_down (uint gpio)`

Determine if the specified GPIO is pulled down.

Parameters

- `gpio` GPIO number

Returns

- true if the GPIO is pulled down

3.9.4.11. gpio_is_pulled_up

`static bool gpio_is_pulled_up (uint gpio)`

Determine if the specified GPIO is pulled up.

Parameters

- `gpio` GPIO number

Returns

- true if the GPIO is pulled up

3.9.4.12. gpio_pull_down

```
static void gpio_pull_down (uint gpio)
```

Set specified GPIO to be pulled down.

Parameters

- `gpio` GPIO number

3.9.4.13. gpio_pull_up

```
static void gpio_pull_up (uint gpio)
```

Set specified GPIO to be pulled up.

Parameters

- `gpio` GPIO number

3.9.4.14. gpio_put

```
static void gpio_put (uint gpio,
       bool value)
```

Drive a single GPIO high/low.

Parameters

- `gpio` GPIO number
- `value` If false clear the GPIO, otherwise set it.

3.9.4.15. gpio_put_all

```
static void gpio_put_all (uint32_t value)
```

Drive all pins simultaneously.

Parameters

- `value` Bitmask of GPIO values to change, as bits 0-29

3.9.4.16. gpio_put_masked

```
static void gpio_put_masked (uint32_t mask,
       uint32_t value)
```

Drive GPIO high/low depending on parameters.

For each 1 bit in `mask`, drive that pin to the value given by corresponding bit in `value`, leaving other pins unchanged. Since this uses the TOGL alias, it is concurrency-safe with e.g. an IRQ bashing different pins from the same core.

Parameters

- `mask` Bitmask of GPIO values to change, as bits 0-29

- `value` Value to set

3.9.4.17. gpio_set_dir

```
static void gpio_set_dir (uint gpio,
        bool out)
```

Set a single GPIO direction.

Parameters

- `gpio` GPIO number
- `out` true for out, false for in

3.9.4.18. gpio_set_dir_all_bits

```
static void gpio_set_dir_all_bits (uint32_t values)
```

Set direction of all pins simultaneously.

Parameters

- `values` individual settings for each gpio; for GPIO #n, bit N is 1 for out, 0 for in

3.9.4.19. gpio_set_dir_in_masked

```
static void gpio_set_dir_in_masked (uint32_t mask)
```

Set a number of GPIOs to input.

Parameters

- `mask` Bitmask of GPIO to set to input, as bits 0-29

3.9.4.20. gpio_set_dir_masked

```
static void gpio_set_dir_masked (uint32_t mask,
        uint32_t value)
```

Set multiple GPIO directions.

For each 1 bit in "mask", switch that pin to the direction given by corresponding bit in "value", leaving other pins unchanged. E.g. gpio_set_dir_masked(0x3, 0x2); → set pin 0 to input, pin 1 to output, simultaneously.

Parameters

- `mask` Bitmask of GPIO to set to input, as bits 0-29
- `value` Values to set

3.9.4.21. gpio_set_dir_out_masked

```
static void gpio_set_dir_out_masked (uint32_t mask)
```

Set a number of GPIOs to output.

Switch all GPIOs in "mask" to output

Parameters

- `mask` Bitmask of GPIO to set to output, as bits 0-29

3.9.4.22. gpio_set_dormant_irq_enabled

```
void gpio_set_dormant_irq_enabled (uint gpio,
       uint32_t events,
       bool enabled)
```

Enable dormant wake up interrupt for specified GPIO.

This configures IRQs to restart the XOSC or ROSC when they are disabled in dormant mode

Parameters

- `gpio` GPIO number
- `events` Which events will cause an interrupt. See gpio_set_irq_enabled for details.
- `enabled` Enable/disable flag

3.9.4.23. gpio_set_function

```
void gpio_set_function (uint gpio,
       enum gpio_function fn)
```

Select GPIO function.

Parameters

- `gpio` GPIO number
- `fn` Which GPIO function select to use from list gpio_selectors

3.9.4.24. gpio_set_inover

```
void gpio_set_inover (uint gpio,
       uint value)
```

Select GPIO input override.

Parameters

- `gpio` GPIO number
- `value` See gpio_override

3.9.4.25. gpio_set_input_enabled

```
void gpio_set_input_enabled (uint gpio,
       bool enabled)
```

Enable GPIO input.

Parameters

- `gpio` GPIO number
- `enabled` true to enable input on specified GPIO

3.9.4.26. gpio_set_irq_enabled

```
void gpio_set_irq_enabled (uint gpio,
       uint32_t events,
       bool enabled)
```

Enable or disable interrupts for specified GPIO.

Events is a bitmask of the following:

bit	interrupt
0	Low level
1	High level
2	Edge low
3	Edge high

Parameters

- `gpio` GPIO number
- `events` Which events will cause an interrupt
- `enabled` Enable or disable flag

3.9.4.27. gpio_set_irq_enabled_with_callback

```
void gpio_set_irq_enabled_with_callback (uint gpio,
       uint32_t events,
       bool enabled,
       gpio_irq_callback_t callback)
```

Enable interrupts for specified GPIO.

Parameters

- `gpio` GPIO number
- `events` Which events will cause an interrupt See gpio_set_irq_enabled for details.
- `enabled` Enable or disable flag
- `callback` user function to call on GPIO irq. Note only one of these can be set per processor.

3.9.4.28. gpio_set_mask

```
static void gpio_set_mask (uint32_t mask)
```

Drive high every GPIO appearing in mask.

Parameters

- `mask` Bitmask of GPIO values to set, as bits 0-29

3.9.4.29. gpio_set_oeover

```
void gpio_set_oeover (uint gpio,
       uint value)
```

Select GPIO output enable override.

Parameters

- `gpio` GPIO number
- `value` See gpio_override

3.9.4.30. gpio_set_outover

```
void gpio_set_outover (uint gpio,
       uint value)
```

Set GPIO output override.

Parameters

- `gpio` GPIO number
- `value` See gpio_override

3.9.4.31. gpio_set_pulls

```
void gpio_set_pulls (uint gpio,
       bool up,
       bool down)
```

Select up and down pulls on specific GPIO.

Parameters

- `gpio` GPIO number
- `up` If true set a pull up on the GPIO
- `down` If true set a pull down on the GPIO

3.9.4.32. gpio_xor_mask

```
static void gpio_xor_mask (uint32_t mask)
```

Toggle every GPIO appearing in mask.

Parameters

- `mask` Bitmask of GPIO values to toggle, as bits 0-29

3.10. hardware_i2c

I2C Controller API. More…

3.10.1. Functions

- `uint i2c_init (i2c_inst_t *i2c, uint baudrate)`
  Initialise the I2C HW block. More…
- `void i2c_deinit (i2c_inst_t *i2c)`
  Disable the I2C HW block. More…
- `uint i2c_set_baudrate (i2c_inst_t *i2c, uint baudrate)`
  Set I2C baudrate. More…
- `void i2c_set_slave_mode (i2c_inst_t *i2c, bool slave, uint8_t addr)`
  Set I2C port to slave mode. More…
- `static uint i2c_hw_index (i2c_inst_t *i2c)`
  Convert I2c instance to hardware instance number. More…
- `int i2c_write_blocking_until (i2c_inst_t *i2c, uint8_t addr, const uint8_t *src, size_t len, bool nostop, absolute_time_t until)`

Attempt to write specified number of bytes to address, blocking until the specified absolute time is reached. More…

- `int i2c_read_blocking_until (i2c_inst_t *i2c, uint8_t addr, uint8_t *dst, size_t len, bool nostop, absolute_time_t until)`
  Attempt to read specified number of bytes from address, blocking until the specified absolute time is reached. More…
- `static int i2c_write_timeout_us (i2c_inst_t *i2c, uint8_t addr, const uint8_t *src, size_t len, bool nostop, uint timeout_us)`
  Attempt to write specified number of bytes to address, with timeout. More…
- `static int i2c_read_timeout_us (i2c_inst_t *i2c, uint8_t addr, uint8_t *dst, size_t len, bool nostop, uint timeout_us)`
  Attempt to read specified number of bytes from address, with timeout. More…
- `int i2c_write_blocking (i2c_inst_t *i2c, uint8_t addr, const uint8_t *src, size_t len, bool nostop)`
  Attempt to write specified number of bytes to address, blocking. More…
- `int i2c_read_blocking (i2c_inst_t *i2c, uint8_t addr, uint8_t *dst, size_t len, bool nostop)`
  Attempt to read specified number of bytes from address, blocking. More…
- `static size_t i2c_get_write_available (i2c_inst_t *i2c)`
  Determine non-blocking write space available. More…
- `static size_t i2c_get_read_available (i2c_inst_t *i2c)`
  Determine number of bytes received. More…
- `static void i2c_write_raw_blocking (i2c_inst_t *i2c, const uint8_t *src, size_t len)`
  Write direct to TX FIFO. More…
- `static void i2c_read_raw_blocking (i2c_inst_t *i2c, uint8_t *dst, size_t len)`
  Write direct to TX FIFO. More…

3.10.2. Variables

- `i2c_inst_t i2c0_inst` More…

3.10.3. Detailed Description

I2C Controller API.

The I2C bus is a two-wire serial interface, consisting of a serial data line SDA and a serial clock SCL. These wires carry information between the devices connected to the bus. Each device is recognized by a unique address and can operate as either a “transmitter” or “receiver”, depending on the function of the device. Devices can also be considered as masters or slaves when performing data transfers. A master is a device that initiates a data transfer on the bus and generates the clock signals to permit that transfer. At that time, any device addressed is considered a slave.

This API allows the controller to be set up as a master or a slave using the i2c_set_slave_mode function.

The external pins of each controller are connected to GPIO pins as defined in the GPIO muxing table in the datasheet. The muxing options give some IO flexibility, but each controller external pin should be connected to only one GPIO.

Note that the controller does NOT support High speed mode or Ultra-fast speed mode, the fastest operation being fast mode plus at up to 1000Kb/s.

See the datasheet for more information on the I2C controller and its usage.

Example

```
// Sweep through all 7-bit I2C addresses, to see if any slaves are present on
// the I2C bus. Print out a table that looks like this:
//
```

```
// I2C Bus Scan
//   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
// 0
// 1       @
// 2
// 3             @
// 4
// 5
// 6
// 7
//
// E.g. if slave addresses 0x12 and 0x34 were acknowledged.

#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/i2c.h"

// I2C reserves some addresses for special purposes. We exclude these from the scan.
// These are any addresses of the form 000 0xxx or 111 1xxx
bool reserved_addr(uint8_t addr) {
    return (addr & 0x78) == 0 || (addr & 0x78) == 0x78;
}

int main() {
    // Enable UART so we can print status output
    stdio_init_all();

    // This example will use I2C0 on GPIO4 (SDA) and GPIO5 (SCL)
    i2c_init(i2c0, 100 * 1000);
    gpio_set_function(4, GPIO_FUNC_I2C);
    gpio_set_function(5, GPIO_FUNC_I2C);
    gpio_pull_up(4);
    gpio_pull_up(5);

    printf("\nI2C Bus Scan\n");
    printf("   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F\n");

    for (int addr = 0; addr < (1 << 7); ++addr) {
        if (addr % 16 == 0) {
            printf("%02x ", addr);
        }

        // Perform a 1-byte dummy read from the probe address. If a slave
        // acknowledges this address, the function returns the number of bytes
        // transferred. If the address byte is ignored, the function returns
        // -1.

        // Skip over any reserved addresses.
        int bytes_transferred;
        uint8_t rxdata;
        if (reserved_addr(addr))
            bytes_transferred = -1;
        else
            bytes_transferred = i2c_read_blocking(i2c0, addr, &rxdata, 1, false);

        printf(bytes_transferred == -1 ? "." : "@");
        printf(addr % 16 == 15 ? "\n" : "  ");
    }
    printf("Done.\n");
    return 0;
}
```

3.10.4. Function Documentation

3.10.4.1. i2c_deinit

```
void i2c_deinit (i2c_inst_t *i2c)
```

Disable the I2C HW block.

Disable the I2C again if it is no longer used. Must be reinitialised before being used again.

Parameters

- `i2c` Either i2c0 or i2c1

3.10.4.2. i2c_get_read_available

```
static size_t i2c_get_read_available (i2c_inst_t *i2c)
```

Determine number of bytes received.

Parameters

- `i2c` Either i2c0 or i2c1

Returns

- 0 if no data available, if return is nonzero at least that many bytes can be read without blocking.

3.10.4.3. i2c_get_write_available

```
static size_t i2c_get_write_available (i2c_inst_t *i2c)
```

Determine non-blocking write space available.

Parameters

- `i2c` Either i2c0 or i2c1

Returns

- 0 if no space is available in the I2C to write more data. If return is nonzero, at least that many bytes can be written without blocking.

3.10.4.4. i2c_hw_index

```
static uint i2c_hw_index (i2c_inst_t *i2c)
```

Convert I2c instance to hardware instance number.

Parameters

- `i2c` I2C instance

Returns

- Number of UART, 0 or 1.

3.10.4.5. i2c_init

```
uint i2c_init (i2c_inst_t *i2c,
       uint baudrate)
```

Initialise the I2C HW block.

Put the I2C hardware into a known state, and enable it. Must be called before other functions. By default, the I2C is configured to operate as a master.

The I2C bus frequency is set as close as possible to requested, and the return actual rate set is returned

Parameters

- `i2c` Either i2c0 or i2c1
- `baudrate` Baudrate in Hz (e.g. 100kHz is 100000)

Returns

- Actual set baudrate

3.10.4.6. i2c_read_blocking

```
int i2c_read_blocking (i2c_inst_t *i2c,
       uint8_t addr,
       uint8_t *dst,
       size_t len,
       bool nostop)
```

Attempt to read specified number of bytes from address, blocking.

Parameters

- `i2c` Either i2c0 or i2c1
- `addr` Address of device to read from
- `dst` Pointer to buffer to receive data
- `len` Length of data in bytes to receive
- `nostop` If true, master retains control of the bus at the end of the transfer (no Stop is issued), and the next transfer will begin with a Restart rather than a Start.

Returns

- Number of bytes read, or PICO_ERROR_GENERIC if address not acknowledged, no device present.

3.10.4.7. i2c_read_blocking_until

```
int i2c_read_blocking_until (i2c_inst_t *i2c,
       uint8_t addr,
       uint8_t *dst,
       size_t len,
       bool nostop,
       absolute_time_t until)
```

Attempt to read specified number of bytes from address, blocking until the specified absolute time is reached.

Parameters

- `i2c` Either i2c0 or i2c1
- `addr` Address of device to read from
- `dst` Pointer to buffer to receive data
- `len` Length of data in bytes to receive
- `nostop` If true, master retains control of the bus at the end of the transfer (no Stop is issued), and the next transfer will begin with a Restart rather than a Start.
- `until` The absolute time that the block will wait until the entire transaction is complete.

Returns

- Number of bytes read, or PICO_ERROR_GENERIC if address not acknowledged, no device present, or PICO_ERROR_TIMEOUT if a timeout occured.

3.10.4.8. i2c_read_raw_blocking

```
static void i2c_read_raw_blocking (i2c_inst_t *i2c,
       uint8_t *dst,
       size_t len)
```

Write direct to TX FIFO.

Reads directly from the I2C RX FIFO which us mainly useful for slave-mode operation.

Parameters

- `i2c` Either i2c0 or i2c1
- `dst` Buffer to accept data
- `len` Number of bytes to send

3.10.4.9. i2c_read_timeout_us

```
static int i2c_read_timeout_us (i2c_inst_t *i2c,
       uint8_t addr,
       uint8_t *dst,
       size_t len,
       bool nostop,
       uint timeout_us)
```

Attempt to read specified number of bytes from address, with timeout.

Parameters

- `i2c` Either i2c0 or i2c1
- `addr` Address of device to read from
- `dst` Pointer to buffer to receive data
- `len` Length of data in bytes to receive
- `nostop` If true, master retains control of the bus at the end of the transfer (no Stop is issued), and the next transfer will begin with a Restart rather than a Start.
- `timeout_us` The time that the function will wait for the entire transaction to complete

Returns

- Number of bytes read, or PICO_ERROR_GENERIC if address not acknowledged, no device present, or PICO_ERROR_TIMEOUT if a timeout occured.

3.10.4.10. i2c_set_baudrate

```
uint i2c_set_baudrate (i2c_inst_t *i2c,
       uint baudrate)
```

Set I2C baudrate.

Set I2C bus frequency as close as possible to requested, and return actual rate set. Baudrate may not be as exactly requested due to clocking limitations.

Parameters

- i2c Either i2c0 or i2c1
- baudrate Baudrate in Hz (e.g. 100kHz is 100000)

Returns

- Actual set baudrate

3.10.4.11. i2c_set_slave_mode

```
void i2c_set_slave_mode (i2c_inst_t *i2c,
       bool slave,
       uint8_t addr)
```

Set I2C port to slave mode.

Parameters

- i2c Either i2c0 or i2c1
- slave true to use slave mode, false to use master mode
- addr If slave is true, set the slave address to this value

3.10.4.12. i2c_write_blocking

```
int i2c_write_blocking (i2c_inst_t *i2c,
       uint8_t addr,
       const uint8_t *src,
       size_t len,
       bool nostop)
```

Attempt to write specified number of bytes to address, blocking.

Parameters

- i2c Either i2c0 or i2c1
- addr Address of device to write to
- src Pointer to data to send
- len Length of data in bytes to send
- nostop If true, master retains control of the bus at the end of the transfer (no Stop is issued), and the next transfer will begin with a Restart rather than a Start.

Returns

- Number of bytes written, or PICO_ERROR_GENERIC if address not acknowledged, no device present.

3.10.4.13. i2c_write_blocking_until

```
int i2c_write_blocking_until (i2c_inst_t *i2c,
       uint8_t addr,
       const uint8_t *src,
       size_t len,
       bool nostop,
       absolute_time_t until)
```

Attempt to write specified number of bytes to address, blocking until the specified absolute time is reached.

Parameters

- i2c Either i2c0 or i2c1

- addr Address of device to write to
- src Pointer to data to send
- len Length of data in bytes to send
- nostop If true, master retains control of the bus at the end of the transfer (no Stop is issued), and the next transfer will begin with a Restart rather than a Start.
- until The absolute time that the block will wait until the entire transaction is complete. Note, an individual timeout of this value divided by the length of data is applied for each byte transfer, so if the first or subsequent bytes fails to transfer within that sub timeout, the function will return with an error.

Returns

- Number of bytes written, or PICO_ERROR_GENERIC if address not acknowledged, no device present, or PICO_ERROR_TIMEOUT if a timeout occured.

3.10.4.14. i2c_write_raw_blocking

```
static void i2c_write_raw_blocking (i2c_inst_t *i2c,
       const uint8_t *src,
       size_t len)
```

Write direct to TX FIFO.

Writes directly to the to I2C TX FIFO which us mainly useful for slave-mode operation.

Parameters

- i2c Either i2c0 or i2c1
- src Data to send
- len Number of bytes to send

3.10.4.15. i2c_write_timeout_us

```
static int i2c_write_timeout_us (i2c_inst_t *i2c,
       uint8_t addr,
       const uint8_t *src,
       size_t len,
       bool nostop,
       uint timeout_us)
```

Attempt to write specified number of bytes to address, with timeout.

Parameters

- i2c Either i2c0 or i2c1
- addr Address of device to write to
- src Pointer to data to send
- len Length of data in bytes to send
- nostop If true, master retains control of the bus at the end of the transfer (no Stop is issued), and the next transfer will begin with a Restart rather than a Start.
- timeout_us The time that the function will wait for the entire transaction to complete. Note, an individual timeout of this value divided by the length of data is applied for each byte transfer, so if the first or subsequent bytes fails to transfer within that sub timeout, the function will return with an error.

Returns

- Number of bytes written, or PICO_ERROR_GENERIC if address not acknowledged, no device present, or PICO_ERROR_TIMEOUT if a timeout occured.

3.11. hardware_interp

Hardware Interpolator API. More…

3.11.1. Modules

- `interp_config`
  Interpolator configuration.

3.11.2. Functions

- `void interp_claim_lane (interp_hw_t *interp, uint lane)`
  Claim the interpolator lane specified. More…
- `void interp_claim_lane_mask (interp_hw_t *interp, uint lane_mask)`
  Claim the interpolator lanes specified in the mask. More…
- `void interp_unclaim_lane (interp_hw_t *interp, uint lane)`
  Release a previously claimed interpolator lane. More…
- `static void interp_add_force_bits (interp_hw_t *interp, uint lane, uint bits)`
  Directly set the force bits on a specified lane. More…
- `void interp_save (interp_hw_t *interp, interp_hw_save_t *saver)`
  Save the specified interpolator state. More…
- `void interp_restore (interp_hw_t *interp, interp_hw_save_t *saver)`
  Restore an interpolator state. More…
- `static void interp_set_base (interp_hw_t *interp, uint lane, uint32_t val)`
  Sets the interpolator base register by lane. More…
- `static uint32_t interp_get_base (interp_hw_t *interp, uint lane)`
  Gets the content of interpolator base register by lane. More…
- `static void interp_set_base_both (interp_hw_t *interp, uint32_t val)`
  Sets the interpolator base registers simultenously. More…
- `static void interp_set_accumulator (interp_hw_t *interp, uint lane, uint32_t val)`
  Sets the interpolator accumulator register by lane. More…
- `static uint32_t interp_get_accumulator (interp_hw_t *interp, uint lane)`
  Gets the content of the interpolator accumulator register by lane. More…
- `static uint32_t interp_pop_lane_result (interp_hw_t *interp, uint lane)`
  Read lane result, and write lane results to both accumulators to update the interpolator. More…
- `static uint32_t interp_peek_lane_result (interp_hw_t *interp, uint lane)`
  Read lane result. More…
- `static uint32_t interp_pop_full_result (interp_hw_t *interp)`
  Read lane result, and write lane results to both accumulators to update the interpolator. More…
- `static uint32_t interp_peek_full_result (interp_hw_t *interp)`
  Read lane result. More…
- `static void interp_add_accumulater (interp_hw_t *interp, uint lane, uint32_t val)`
  Add to accumulator. More…

- `static uint32_t interp_get_raw (interp_hw_t *interp, uint lane)`
  Get raw lane value. More…

3.11.3. Detailed Description

Hardware Interpolator API.

Each core is equipped with two interpolators (INTERP0 and INTERP1) which can be used to accelerate tasks by combining certain pre-configured simple operations into a single processor cycle. Intended for cases where the pre-configured operation is repeated a large number of times, this results in code which uses both fewer CPU cycles and fewer CPU registers in the time critical sections of the code.

The interpolators are used heavily to accelerate audio operations within the Pico SDK, but their flexible configuration make it possible to optimise many other tasks such as quantization and dithering, table lookup address generation, affine texture mapping, decompression and linear feedback.

Please refer to the RP2040 datasheet for more information on the HW interpolators and how they work.

3.11.4. Function Documentation

3.11.4.1. interp_add_accumulater

```
static void interp_add_accumulater (interp_hw_t *interp,
       uint lane,
       uint32_t val)
```

Add to accumulator.

Atomically add the specified value to the accumulator on the specified lane

Parameters

- `interp` Interpolator instance, interp0 or interp1.
- `lane` The lane number, 0 or 1
- `val` Value to add

Returns

- The content of the FULL register

3.11.4.2. interp_add_force_bits

```
static void interp_add_force_bits (interp_hw_t *interp,
       uint lane,
       uint bits)
```

Directly set the force bits on a specified lane.

These bits are ORed into bits 29:28 of the lane result presented to the processor on the bus. There is no effect on the internal 32-bit datapath.

Useful for using a lane to generate sequence of pointers into flash or SRAM, saving a subsequent OR or add operation.

Parameters

- `interp` Interpolator instance, interp0 or interp1.
- `lane` The lane to set
- `bits` The bits to set (bits 0 and 1, value range 0-3)

3.11.4.3. interp_claim_lane

```
void interp_claim_lane (interp_hw_t *interp,
       uint lane)
```

Claim the interpolator lane specified.

Use this function to claim exclusive access to the specfied interpolator lane.

This function will panic if the lane is already claimed.

Parameters

- `interp` Interpolator on which to claim a lane. interp0 or interp1
- `lane` The lane number, 0 or 1.

3.11.4.4. interp_claim_lane_mask

```
void interp_claim_lane_mask (interp_hw_t *interp,
       uint lane_mask)
```

Claim the interpolator lanes specified in the mask.

Parameters

- `interp` Interpolator on which to claim lanes. interp0 or interp1
- `lane_mask` Bit pattern of lanes to claim (only bits 0 and 1 are valid)

3.11.4.5. interp_get_accumulator

```
static uint32_t interp_get_accumulator (interp_hw_t *interp,
       uint lane)
```

Gets the content of the interpolator accumulator register by lane.

Parameters

- `interp` Interpolator instance, interp0 or interp1.
- `lane` The lane number, 0 or 1

Returns

- The current content of the register

3.11.4.6. interp_get_base

```
static uint32_t interp_get_base (interp_hw_t *interp,
       uint lane)
```

Gets the content of interpolator base register by lane.

Parameters

- `interp` Interpolator instance, interp0 or interp1.
- `lane` The lane number, 0 or 1 or 2

Returns

- The current content of the lane base register

3.11.4.7. interp_get_raw

```
static uint32_t interp_get_raw (interp_hw_t *interp,
        uint lane)
```

Get raw lane value.

Returns the raw shift and mask value from the specified lane, BASE0 is NOT added

Parameters

- `interp` Interpolator instance, interp0 or interp1.
- `lane` The lane number, 0 or 1

Returns

- The raw shift/mask value

3.11.4.8. interp_peek_full_result

```
static uint32_t interp_peek_full_result (interp_hw_t *interp)
```

Read lane result.

Parameters

- `interp` Interpolator instance, interp0 or interp1.

Returns

- The content of the FULL register

3.11.4.9. interp_peek_lane_result

```
static uint32_t interp_peek_lane_result (interp_hw_t *interp,
        uint lane)
```

Read lane result.

Parameters

- `interp` Interpolator instance, interp0 or interp1.
- `lane` The lane number, 0 or 1

Returns

- The content of the lane result register

3.11.4.10. interp_pop_full_result

```
static uint32_t interp_pop_full_result (interp_hw_t *interp)
```

Read lane result, and write lane results to both accumulators to update the interpolator.

Parameters

- `interp` Interpolator instance, interp0 or interp1.

Returns

- The content of the FULL register

3.11.4.11. interp_pop_lane_result

```
static uint32_t interp_pop_lane_result (interp_hw_t *interp,
       uint lane)
```

Read lane result, and write lane results to both accumulators to update the interpolator.

Parameters

- `interp` Interpolator instance, interp0 or interp1.
- `lane` The lane number, 0 or 1

Returns

- The content of the lane result register

3.11.4.12. interp_restore

```
void interp_restore (interp_hw_t *interp,
       interp_hw_save_t *saver)
```

Restore an interpolator state.

Parameters

- `interp` Interpolator instance, interp0 or interp1.
- `saver` Pointer to save structure to reapply to the specified interpolator

3.11.4.13. interp_save

```
void interp_save (interp_hw_t *interp,
       interp_hw_save_t *saver)
```

Save the specified interpolator state.

Can be used to save state if you need an interpolator for another purpose, state can then be recovered afterwards and continue from that point

Parameters

- `interp` Interpolator instance, interp0 or interp1.
- `saver` Pointer to the save structure to fill in

3.11.4.14. interp_set_accumulator

```
static void interp_set_accumulator (interp_hw_t *interp,
       uint lane,
       uint32_t val)
```

Sets the interpolator accumulator register by lane.

Parameters

- `interp` Interpolator instance, interp0 or interp1.
- `lane` The lane number, 0 or 1
- `val` The value to apply to the register

3.11.4.15. interp_set_base

```
static void interp_set_base (interp_hw_t *interp,
       uint lane,
       uint32_t val)
```

Sets the interpolator base register by lane.

Parameters

- `interp` Interpolator instance, interp0 or interp1.
- `lane` The lane number, 0 or 1 or 2
- `val` The value to apply to the register

3.11.4.16. interp_set_base_both

```
static void interp_set_base_both (interp_hw_t *interp,
       uint32_t val)
```

Sets the interpolator base registers simultenously.

The lower 16 bits go to BASE0, upper bits to BASE1 simultaneously. Each half is sign-extended to 32 bits if that lane's SIGNED flag is set.

Parameters

- `interp` Interpolator instance, interp0 or interp1.
- `val` The value to apply to the register

3.11.4.17. interp_unclaim_lane

```
void interp_unclaim_lane (interp_hw_t *interp,
       uint lane)
```

Release a previously claimed interpolator lane.

Parameters

- `interp` Interpolator on which to release a lane. interp0 or interp1
- `lane` The lane number, 0 or 1

3.12. hardware_irq

Hardware interrupt handling. More…

3.12.1. Typedefs

- `typedef void(* irq_handler_t )()`
  Interrupt handler function type. More…

3.12.2. Functions

- `void irq_set_priority (uint num, uint8_t hardware_priority)`
  Set specified interrupts priority. More…
- `void irq_set_enabled (uint num, bool enabled)`

Enable or disable a specific interrupt on the executing core. More…

- `bool irq_is_enabled (uint num)`
  Determine if a specific interrupt is enabled on the executing core. More…
- `void irq_set_mask_enabled (uint32_t mask, bool enabled)`
  Enable/disable multiple interrupts on the executing core. More…
- `void irq_set_exclusive_handler (uint num, irq_handler_t handler)`
  Set an exclusive interrupt handler for an interrupt on the executing core. More…
- `void irq_add_shared_handler (uint num, irq_handler_t handler, uint8_t order_priority)`
  Add a shared interrupt handler for an interrupt on the executing core. More…
- `void irq_remove_handler (uint num, irq_handler_t handler)`
  Remove a specific interrupt handler for the given irq number on the executing core. More…
- `irq_handler_t irq_get_vtable_handler (uint num)`
  Get the current IRQ handler for the specified IRQ from the currently installed hardware vector table (VTOR) of the execution core. More…
- `static void irq_clear (uint int_num)`
  Clear a specific interrupt on the executing core. More…
- `void irq_set_pending (uint num)`
  Force an interrupt to pending on the executing core. More…

3.12.3. Detailed Description

Hardware interrupt handling.

The RP2040 uses the standard ARM nested vectored interrupt controller (NVIC).

Interrupts are identified by a number from 0 to 31.

On the RP2040, only the lower 26 IRQ signals are connected on the NVIC; IRQs 26 to 31 are tied to zero (never firing).

There is one NVIC per core, and each core's NVIC has the same hardware interrupt lines routed to it, with the exception of the IO interrupts where there is one IO interrupt per bank, per core. These are completely independent, so for example, processor 0 can be interrupted by GPIO 0 in bank 0, and processor 1 by GPIO 1 in the same bank.

That all IRQ APIs affect the executing core only (i.e. the core calling the function).

You should not enable the same (shared) IRQ number on both cores, as this will lead to race conditions or starvation of one of the cores. Additionally don't forget that disabling interrupts on one core does not disable interrupts on the other core.

There are three different ways to set handlers for an IRQ:

- Calling irq_add_shared_handler() at runtime to add a handler for a multiplexed interrupt (e.g. GPIO bank) on the current core. Each handler, should check and clear the relevant hardware interrupt source
- Calling irq_set_exclusive_handler() at runtime to install a single handler for the interrupt on the current core
- Defining the interrupt handler explicitly in your application (e.g. by defining void isr_dma_0 will make that function the handler for the DMA_IRQ_0 on core 0, and you will not be able to change it using the above APIs at runtime). Using this method can cause link conflicts at runtime, and offers no runtime performance benefit (i.e, it should not generally be used).

If an IRQ is enabled and fires with no handler installed, a breakpoint will be hit and the IRQ number will be in r0.

Interrupt NumbersInterrupts are numbered as follows, a set of defines is available (intctrl.h) with these names to avoid using the numbers directly.

IRQ	Interrupt Source
0	TIMER_IRQ_0
1	TIMER_IRQ_1
2	TIMER_IRQ_2
3	TIMER_IRQ_3
4	PWM_IRQ_WRAP
5	USBCTRL_IRQ
6	XIP_IRQ
7	PIO0_IRQ_0
8	PIO0_IRQ_1
9	PIO1_IRQ_0
10	PIO1_IRQ_1
11	DMA_IRQ_0
12	DMA_IRQ_1
13	IO_IRQ_BANK0
14	IO_IRQ_QSPI
15	SIO_IRQ_PROC0
16	SIO_IRQ_PROC1
17	CLOCKS_IRQ
18	SPI0_IRQ
19	SPI1_IRQ
20	UART0_IRQ
21	UART1_IRQ
22	ADC0_IRQ_FIFO
23	I2C0_IRQ
24	I2C1_IRQ
25	RTC_IRQ

3.12.4. Function Documentation

3.12.4.1. irq_add_shared_handler

```
void irq_add_shared_handler (uint num,
        irq_handler_t handler,
        uint8_t order_priority)
```

Add a shared interrupt handler for an interrupt on the executing core.

Use this method to add a handler on an irq number shared between multiple distinct hardware sources (e.g. GPIO, DMA or PIO IRQs). Handlers added by this method will all be called in sequence from highest order_priority to lowest. The `irq_set_exclusive_handler()` method should be used instead if you know there will or should only ever be one handler for the interrupt.

This method will assert if there is an exclusive interrupt handler set for this irq number on this core, or if the (total across all IRQs on both cores) maximum (configurable via PICO_MAX_SHARED_IRQ_HANDLERS) number of shared handlers would be exceeded.

Parameters

- `num` Interrupt number
- `handler` The handler to set. See irq_handler_t
- `order_priority` The order priority controls the order that handlers for the same IRQ number on the core are called. The shared irq handlers for an interrupt are all called when an IRQ fires, however the order of the calls is based on the order_priority (higher priorities are called first, identical priorities are called in undefined order). A good rule of thumb is to use PICO_SHARED_IRQ_HANDLER_DEFAULT_ORDER_PRIORITY if you don't much care, as it is in the middle of the priority range by default.

See also

- irq_set_exclusive_handler

3.12.4.2. irq_clear

```
static void irq_clear (uint int_num)
```

Clear a specific interrupt on the executing core.

Parameters

- `int_num` Interrupt number Interrupt Numbers

3.12.4.3. irq_get_vtable_handler

```
irq_handler_t irq_get_vtable_handler (uint num)
```

Get the current IRQ handler for the specified IRQ from the currently installed hardware vector table (VTOR) of the execution core.

Parameters

- `num` Interrupt number Interrupt Numbers

Returns

- the address stored in the VTABLE for the given irq number

3.12.4.4. irq_is_enabled

```
bool irq_is_enabled (uint num)
```

Determine if a specific interrupt is enabled on the executing core.

Parameters

- `num` Interrupt number Interrupt Numbers

Returns

- true if the interrupt is enabled

3.12.4.5. irq_remove_handler

```
void irq_remove_handler (uint num,
       irq_handler_t handler)
```

Remove a specific interrupt handler for the given irq number on the executing core.

This method may be used to remove an irq set via either `irq_set_exclusive_handler()` or `irq_add_shared_handler()`, and will assert if the handler is not currently installed for the given IRQ number

Parameters

- `num` Interrupt number Interrupt Numbers
- `handler` The handler to removed.

See also

- irq_set_exclusive_handler
- irq_add_shared_handler

3.12.4.6. irq_set_enabled

```
void irq_set_enabled (uint num,
       bool enabled)
```

Enable or disable a specific interrupt on the executing core.

Parameters

- `num` Interrupt number Interrupt Numbers
- `enabled` true to enable the interrupt, false to disable

3.12.4.7. irq_set_exclusive_handler

```
void irq_set_exclusive_handler (uint num,
       irq_handler_t handler)
```

Set an exclusive interrupt handler for an interrupt on the executing core.

Use this method to set a handler for single IRQ source interrupts, or when your code, use case or performance requirements dictate that there should no other handlers for the interrupt.

This method will assert if there is already any sort of interrupt handler installed for the specified irq number.

Parameters

- `num` Interrupt number Interrupt Numbers
- `handler` The handler to set. See irq_handler_t

See also

- irq_add_shared_handler

3.12.4.8. irq_set_mask_enabled

```
void irq_set_mask_enabled (uint32_t mask,
       bool enabled)
```

Enable/disable multiple interrupts on the executing core.

Parameters

- `mask` 32-bit mask with one bits set for the interrupts to enable/disable
- `enabled` true to enable the interrupts, false to disable them.

3.12.4.9. irq_set_pending

```
void irq_set_pending (uint num)
```

Force an interrupt to pending on the executing core.

This should generally not be used for IRQs connected to hardware.

Parameters

- `num` Interrupt number Interrupt Numbers

3.12.4.10. irq_set_priority

```
void irq_set_priority (uint num,
        uint8_t hardware_priority)
```

Set specified interrupts priority.

Parameters

- `num` Interrupt number
- `hardware_priority` Priority to set. Hardware priorities range from 0 (lowest) to 255 (highest) though only the top 2 bits are significant on ARM Cortex M0+. To make it easier to specify higher or lower priorities than the default, all IRQ priorities are initialized to PICO_DEFAULT_IRQ_PRIORITY by the SDK runtime at startup. PICO_DEFAULT_IRQ_PRIORITY defaults to 0x80

3.13. hardware_pio

Programmable I/O (PIO) API. More…

3.13.1. Modules

- **`sm_config`**
  PIO state machine configuration.

3.13.2. Enumerations

- `enum pio_fifo_join { PIO_FIFO_JOIN_NONE = 0, PIO_FIFO_JOIN_TX = 1, PIO_FIFO_JOIN_RX = 2 }`
  FIFO join states.

3.13.3. Functions

- `static void pio_sm_set_config (PIO pio, uint sm, const pio_sm_config *config)`
  Apply a state machine configuration to a state machine. More…
- `static uint pio_get_index (PIO pio)`
  Return the instance number of a PIO instance. More…
- `static void pio_gpio_init (PIO pio, uint pin)`
  Setup the function select for a GPIO to use output from the given PIO instance. More…
- `static uint pio_get_dreq (PIO pio, uint sm, bool is_tx)`
  Return the DREQ to use for pacing transfers to a particular state machine. More…
- `bool pio_can_add_program (PIO pio, const pio_program_t *program)`
  Determine whether the given program can (at the time of the call) be loaded onto the PIO instance. More…

- bool pio_can_add_program_at_offset (PIO pio, const pio_program_t *program, uint offset)
  Determine whether the given program can (at the time of the call) be loaded onto the PIO instance starting at a particular location. More…

- uint pio_add_program (PIO pio, const pio_program_t *program)
  Attempt to load the program, panicking if not possible. More…

- void pio_add_program_at_offset (PIO pio, const pio_program_t *program, uint offset)
  Attempt to load the program at the specified instruction memory offset, panicking if not possible. More…

- void pio_remove_program (PIO pio, const pio_program_t *program, uint loaded_offset)
  Remove a program from a PIO instance's instruction memory. More…

- void pio_clear_instruction_memory (PIO pio)
  Clears all of a PIO instance's instruction memory. More…

- void pio_sm_init (PIO pio, uint sm, uint initial_pc, const pio_sm_config *config)
  Resets the state machine to a consisten state, and configures it. More…

- static void pio_sm_set_enabled (PIO pio, uint sm, bool enabled)
  Enable or disable a PIO state machine. More…

- static void pio_set_sm_mask_enabled (PIO pio, uint32_t mask, bool enable)
  Enable or disable multiple PIO state machines. More…

- static void pio_sm_restart (PIO pio, uint sm)
  Restart a state machine with a known state. More…

- static void pio_restart_sm_mask (PIO pio, uint32_t mask)
  Restart multiple state machine with a known state. More…

- static void pio_sm_clkdiv_restart (PIO pio, uint sm)
  Restart a state machine's clock divider (resetting the fractional count) More…

- static void pio_clkdiv_restart_sm_mask (PIO pio, uint32_t mask)
  Restart multiple state machines' clock dividers (resetting the fractional count) More…

- static void pio_enable_sm_mask_in_sync (PIO pio, uint32_t mask)
  Enable multiple PIO state machines synchronizing their clock dividers. More…

- static uint8_t pio_sm_get_pc (PIO pio, uint sm)
  Return the current program counter for a state machine. More…

- static void pio_sm_exec (PIO pio, uint sm, uint instr)
  Immediately execute an instruction on a state machine. More…

- static bool pio_sm_is_exec_stalled (PIO pio, uint sm)
  Determine if an instruction set by pio_sm_exec() is stalled executing. More…

- static void pio_sm_exec_wait_blocking (PIO pio, uint sm, uint instr)
  Immediately execute an instruction on a state machine and wait for it to complete. More…

- static void pio_sm_set_wrap (PIO pio, uint sm, uint wrap_target, uint wrap)
  Set the current wrap configuration for a state machine. More…

- static void pio_sm_put (PIO pio, uint sm, uint32_t data)
  Write a word of data to a state machine's TX FIFO. More…

- static uint32_t pio_sm_get (PIO pio, uint sm)
  Read a word of data from a state machine's RX FIFO. More…

- static bool pio_sm_is_rx_fifo_full (PIO pio, uint sm)
  Determine if a state machine's RX FIFO is full. More…

- static bool pio_sm_is_rx_fifo_empty (PIO pio, uint sm)
  Determine if a state machine's RX FIFO is empty. More…

- `static uint pio_sm_get_rx_fifo_level (PIO pio, uint sm)`
  Return the number of elements currently in a state machine's RX FIFO. More…
- `static bool pio_sm_is_tx_fifo_full (PIO pio, uint sm)`
  Determine if a state machine's TX FIFO is full. More…
- `static bool pio_sm_is_tx_fifo_empty (PIO pio, uint sm)`
  Determine if a state machine's TX FIFO is empty. More…
- `static uint pio_sm_get_tx_fifo_level (PIO pio, uint sm)`
  Return the number of elements currently in a state machine's TX FIFO. More…
- `static void pio_sm_put_blocking (PIO pio, uint sm, uint32_t data)`
  Write a word of data to a state machine's TX FIFO, blocking if the FIFO is full. More…
- `static uint32_t pio_sm_get_blocking (PIO pio, uint sm)`
  Read a word of data from a state machine's RX FIFO, blocking if the FIFO is empty. More…
- `void pio_sm_drain_tx_fifo (PIO pio, uint sm)`
  Empty out a state machine's TX FIFO. More…
- `static void pio_sm_set_clkdiv (PIO pio, uint sm, float div)`
  set the current clock divider for a state machine More…
- `static void pio_sm_set_clkdiv_int_frac (PIO pio, uint sm, uint16_t div_int, uint8_t div_frac)`
  set the current clock divider for a state machine using a 16:8 fraction More…
- `static void pio_sm_clear_fifos (PIO pio, uint sm)`
  Clear a state machine's TX and RX FIFOFs. More…
- `void pio_sm_set_pins (PIO pio, uint sm, uint32_t pin_values)`
  Use a state machine to set a value on all pins for the PIO instance. More…
- `void pio_sm_set_pins_with_mask (PIO pio, uint sm, uint32_t pin_values, uint32_t pin_mask)`
  Use a state machine to set a value on multiple pins for the PIO instance. More…
- `void pio_sm_set_pindirs_with_mask (PIO pio, uint sm, uint32_t pin_dirs, uint32_t pin_mask)`
  Use a state machine to set the pin directions for multiple pins for the PIO instance. More…
- `void pio_sm_set_consecutive_pindirs (PIO pio, uint sm, uint pin_base, uint pin_count, bool is_out)`
  Use a state machine to set the same pin direction for multiple consecutive pins for the PIO instance. More…

3.13.4. Macros

- `#define pio0 pio0_hw` More…

3.13.5. Macros

- `#define pio1 pio1_hw` More…

3.13.6. Detailed Description

Programmable I/O (PIO) API.

A programmable input/output block (PIO) is a versatile hardware interface which can support a number of different IO standards. There are two PIO blocks in the RP2040

Each PIO is programmable in the same sense as a processor: the four state machines independently execute short, sequential programs, to manipulate GPIOs and transfer data. Unlike a general purpose processor, PIO state machines are highly specialised for IO, with a focus on determinism, precise timing, and close integration with fixed-function hardware. Each state machine is equipped with:

- Two 32-bit shift registers – either direction, any shift count
- Two 32-bit scratch registers
- 4×32 bit bus FIFO in each direction (TX/RX), reconfigurable as 8×32 in a single direction
- Fractional clock divider (16 integer, 8 fractional bits)
- Flexible GPIO mapping
- DMA interface, sustained throughput up to 1 word per clock from system DMA
- IRQ flag set/clear/status

Full details of the PIO can be found in the RP2040 datasheet.

3.13.7. Function Documentation

3.13.7.1. pio_add_program

```
uint pio_add_program (PIO pio,
       const pio_program_t *program)
```

Attempt to load the program, panicking if not possible.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `program` the program definition

Returns

- the instruction memory offset the program is loaded at

See also

- pico_can_add_program() if you need to check whether the program can be loaded

3.13.7.2. pio_add_program_at_offset

```
void pio_add_program_at_offset (PIO pio,
       const pio_program_t *program,
       uint offset)
```

Attempt to load the program at the specified instruction memory offset, panicking if not possible.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `program` the program definition
- `offset` the instruction memory offset wanted for the start of the program

Returns

- the instruction memory offset the program is loaded at

See also

- pico_can_add_program_at_offset() if you need to check whether the program can be loaded

3.13.7.3. pio_can_add_program

```
bool pio_can_add_program (PIO pio,
       const pio_program_t *program)
```

Determine whether the given program can (at the time of the call) be loaded onto the PIO instance.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `program` the program definition

Returns

- true if the program can be loaded; false if there is not suitable space in the instruction memory

3.13.7.4. pio_can_add_program_at_offset

```
bool pio_can_add_program_at_offset (PIO pio,
       const pio_program_t *program,
       uint offset)
```

Determine whether the given program can (at the time of the call) be loaded onto the PIO instance starting at a particular location.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `program` the program definition
- `offset` the instruction memory offset wanted for the start of the program

Returns

- true if the program can be loaded at that location; false if there is not space in the instruction memory

3.13.7.5. pio_clear_instruction_memory

```
void pio_clear_instruction_memory (PIO pio)
```

Clears all of a PIO instance's instruction memory.

Parameters

- `pio` The PIO instance; either pio0 or pio1

3.13.7.6. pio_clkdiv_restart_sm_mask

```
static void pio_clkdiv_restart_sm_mask (PIO pio,
       uint32_t mask)
```

Restart multiple state machines' clock dividers (resetting the fractional count)

This method can be used to guarantee that multiple state machines with fractional clock dividers are exactly in sync

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `mask` bit mask of state machine indexes to modify the enabled state of

3.13.7.7. pio_enable_sm_mask_in_sync

```
static void pio_enable_sm_mask_in_sync (PIO pio,
       uint32_t mask)
```

Enable multiple PIO state machines synchronizing their clock dividers.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `mask` bit mask of state machine indexes to modify the enabled state of

3.13.7.8. pio_get_dreq

```
static uint pio_get_dreq (PIO pio,
       uint sm,
       bool is_tx)
```

Return the DREQ to use for pacing transfers to a particular state machine.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)
- `is_tx` true for sending data to the state machine, false for received data from the state machine

3.13.7.9. pio_get_index

```
static uint pio_get_index (PIO pio)
```

Return the instance number of a PIO instance.

Parameters

- `pio` The PIO instance; either pio0 or pio1

Returns

- the PIO instance number (either 0 or 1)

3.13.7.10. pio_gpio_init

```
static void pio_gpio_init (PIO pio,
       uint pin)
```

Setup the function select for a GPIO to use output from the given PIO instance.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `pin` the GPIO pin whose function select to set

3.13.7.11. pio_remove_program

```
void pio_remove_program (PIO pio,
       const pio_program_t *program,
       uint loaded_offset)
```

Remove a program from a PIO instance's instruction memory.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `program` the program definition
- `loaded_offset` the loaded offset returned when the program was added

3.13.7.12. pio_restart_sm_mask

```
static void pio_restart_sm_mask (PIO pio,
       uint32_t mask)
```

Restart multiple state machine with a known state.

This method clears the ISR, shift counters, clock divider counter pin write flags, delay counter, latched EXEC instruction, and IRQ wait condition.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `mask` bit mask of state machine indexes to modify the enabled state of

3.13.7.13. pio_set_sm_mask_enabled

```
static void pio_set_sm_mask_enabled (PIO pio,
       uint32_t mask,
       bool enable)
```

Enable or disable multiple PIO state machines.

Note that this method just sets the enabled state of the state machine; if now enabled they continue exactly from where they left off.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `mask` bit mask of state machine indexes to modify the enabled state of
- `enabled` true to enable the state machines; false to disable

See also

- pio_enable_sm_mask_in_sync if you wish to enable multiple state machines and ensure their clock dividers are in sync.

3.13.7.14. pio_sm_clear_fifos

```
static void pio_sm_clear_fifos (PIO pio,
       uint sm)
```

Clear a state machine's TX and RX FIFOFs.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)

3.13.7.15. pio_sm_clkdiv_restart

```
static void pio_sm_clkdiv_restart (PIO pio,
       uint sm)
```

Restart a state machine's clock divider (resetting the fractional count)

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)

3.13.7.16. pio_sm_drain_tx_fifo

```
void pio_sm_drain_tx_fifo (PIO pio,
        uint sm)
```

Empty out a state machine's TX FIFO.

This method executes `pull` instructions on the state machine until the TX FIFO is empty

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)

3.13.7.17. pio_sm_exec

```
static void pio_sm_exec (PIO pio,
        uint sm,
        uint instr)
```

Immediately execute an instruction on a state machine.

This instruction is executed instead of the next instruction in the normal control flow on the state machine. Subsequent calls to this method replace the previous executed instruction if it is still running.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)
- `instr` the encoded PIO instruction

See also

- pio_sm_is_exec_stalled to see if n executed instruction is stall running (i.e. it is stalled on some condition)

3.13.7.18. pio_sm_exec_wait_blocking

```
static void pio_sm_exec_wait_blocking (PIO pio,
        uint sm,
        uint instr)
```

Immediately execute an instruction on a state machine and wait for it to complete.

This instruction is executed instead of the next instruction in the normal control flow on the state machine. Subsequent calls to this method replace the previous executed instruction if it is still running.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)
- `instr` the encoded PIO instruction

See also

- pio_sm_is_exec_stalled to see if n executed instruction is stall running (i.e. it is stalled on some condition)

3.13.7.19. pio_sm_get

```
static uint32_t pio_sm_get (PIO pio,
       uint sm)
```

Read a word of data from a state machine’s RX FIFO.

If the FIFO is empty, the return value is zero.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)

3.13.7.20. pio_sm_get_blocking

```
static uint32_t pio_sm_get_blocking (PIO pio,
       uint sm)
```

Read a word of data from a state machine’s RX FIFO, blocking if the FIFO is empty.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)

3.13.7.21. pio_sm_get_pc

```
static uint8_t pio_sm_get_pc (PIO pio,
       uint sm)
```

Return the current program counter for a state machine.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)

Returns

- the program counter

3.13.7.22. pio_sm_get_rx_fifo_level

```
static uint pio_sm_get_rx_fifo_level (PIO pio,
       uint sm)
```

Return the number of elements currently in a state machine’s RX FIFO.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)

Returns

- the number of elements in the RX FIFO

3.13.7.23. pio_sm_get_tx_fifo_level

```
static uint pio_sm_get_tx_fifo_level (PIO pio,
        uint sm)
```

Return the number of elements currently in a state machine's TX FIFO.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)

Returns

- the number of elements in the TX FIFO

3.13.7.24. pio_sm_init

```
void pio_sm_init (PIO pio,
        uint sm,
        uint initial_pc,
        const pio_sm_config *config)
```

Resets the state machine to a consisten state, and configures it.

This method:

The state machine is disabled on return from this call

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)
- `initial_pc` the initial prorgam memory offset to run from
- `config` the configuration to apply (or NULL to apply defaults)

3.13.7.25. pio_sm_is_exec_stalled

```
static bool pio_sm_is_exec_stalled (PIO pio,
        uint sm)
```

Determine if an instruction set by pio_sm_exec() is stalled executing.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)

Returns

- true if the executed instructionis still running (stalled)

3.13.7.26. pio_sm_is_rx_fifo_empty

```
static bool pio_sm_is_rx_fifo_empty (PIO pio,
        uint sm)
```

Determine if a state machine's RX FIFO is empty.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)

Returns

- true if the RX FIFO is empty

3.13.7.27. pio_sm_is_rx_fifo_full

```
static bool pio_sm_is_rx_fifo_full (PIO pio,
       uint sm)
```

Determine if a state machine's RX FIFO is full.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)

Returns

- true if the RX FIFO is full

3.13.7.28. pio_sm_is_tx_fifo_empty

```
static bool pio_sm_is_tx_fifo_empty (PIO pio,
       uint sm)
```

Determine if a state machine's TX FIFO is empty.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)

Returns

- true if the TX FIFO is empty

3.13.7.29. pio_sm_is_tx_fifo_full

```
static bool pio_sm_is_tx_fifo_full (PIO pio,
       uint sm)
```

Determine if a state machine's TX FIFO is full.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)

Returns

- true if the TX FIFO is full

3.13.7.30. pio_sm_put

```
static void pio_sm_put (PIO pio,
       uint sm,
       uint32_t data)
```

Write a word of data to a state machine's TX FIFO.

If the FIFO is full, the most recent value will be overwritten

Parameters

- pio The PIO instance; either pio0 or pio1
- sm State machine index (0..3)
- data the 32 bit data value

3.13.7.31. pio_sm_put_blocking

```
static void pio_sm_put_blocking (PIO pio,
       uint sm,
       uint32_t data)
```

Write a word of data to a state machine's TX FIFO, blocking if the FIFO is full.

Parameters

- pio The PIO instance; either pio0 or pio1
- sm State machine index (0..3)
- data the 32 bit data value

3.13.7.32. pio_sm_restart

```
static void pio_sm_restart (PIO pio,
       uint sm)
```

Restart a state machine with a known state.

This method clears the ISR, shift counters, clock divider counter pin write flags, delay counter, latched EXEC instruction, and IRQ wait condition.

Parameters

- pio The PIO instance; either pio0 or pio1
- sm State machine index (0..3)

3.13.7.33. pio_sm_set_clkdiv

```
static void pio_sm_set_clkdiv (PIO pio,
       uint sm,
       float div)
```

set the current clock divider for a state machine

Parameters

- pio The PIO instance; either pio0 or pio1
- sm State machine index (0..3)
- div the floating point clock divider

3.13.7.34. pio_sm_set_clkdiv_int_frac

```
static void pio_sm_set_clkdiv_int_frac (PIO pio,
       uint sm,
       uint16_t div_int,
       uint8_t div_frac)
```

set the current clock divider for a state machine using a 16:8 fraction

Parameters

- pio The PIO instance; either pio0 or pio1
- sm State machine index (0..3)
- div_int the integer part of the clock divider
- div_frac the fractional part of the clock divider in 1/256s

3.13.7.35. pio_sm_set_config

```
static void pio_sm_set_config (PIO pio,
       uint sm,
       const pio_sm_config *config)
```

Apply a state machine configuration to a state machine.

Parameters

- pio Handle to PIO instance; either pio0 or pio1
- sm State machine index (0..3)
- config the configuration to apply

3.13.7.36. pio_sm_set_consecutive_pindirs

```
void pio_sm_set_consecutive_pindirs (PIO pio,
       uint sm,
       uint pin_base,
       uint pin_count,
       bool is_out)
```

Use a state machine to set the same pin direction for multiple consecutive pins for the PIO instance.

This method repeatedly reconfigures the target state machine's pin configuration and executes 'set' instructions to set the pin direction on consecutive pins, before restoring the state machine's pin configuration to what it was.

This method is provided as a convenience to set initial pin directions, and should not be used against a state machine that is enabled.

Parameters

- pio The PIO instance; either pio0 or pio1
- sm State machine index (0..3) to use
- pin_base the first pin to set a direction for
- pin_count the count of consecutive pins to set the direction for
- is_out the direction to set; true = out, false = in

3.13.7.37. pio_sm_set_enabled

```
static void pio_sm_set_enabled (PIO pio,
       uint sm,
       bool enabled)
```

Enable or disable a PIO state machine.

Parameters

- pio The PIO instance; either pio0 or pio1
- sm State machine index (0..3)

- `enabled` true to enable the state machine; false to disable

3.13.7.38. pio_sm_set_pindirs_with_mask

```
void pio_sm_set_pindirs_with_mask (PIO pio,
       uint sm,
       uint32_t pin_dirs,
       uint32_t pin_mask)
```

Use a state machine to set the pin directions for multiple pins for the PIO instance.

This method repeatedly reconfigures the target state machine’s pin configuration and executes 'set' instructions to set pin directions on up to 32 pins, before restoring the state machine’s pin configuration to what it was.

This method is provided as a convenience to set initial pin directions, and should not be used against a state machine that is enabled.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3) to use
- `pin_dirs` the pin directions to set - 1 = out, 0 = in (if the corresponding bit in pin_mask is set)
- `pin_mask` a bit for each pin to indicate whether the corresponding pin_value for that pin should be applied.

3.13.7.39. pio_sm_set_pins

```
void pio_sm_set_pins (PIO pio,
       uint sm,
       uint32_t pin_values)
```

Use a state machine to set a value on all pins for the PIO instance.

This method repeatedly reconfigures the target state machine’s pin configuration and executes 'set' instructions to set values on all 32 pins, before restoring the state machine’s pin configuration to what it was.

This method is provided as a convenience to set initial pin states, and should not be used against a state machine that is enabled.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3) to use
- `the` pin values to set

3.13.7.40. pio_sm_set_pins_with_mask

```
void pio_sm_set_pins_with_mask (PIO pio,
       uint sm,
       uint32_t pin_values,
       uint32_t pin_mask)
```

Use a state machine to set a value on multiple pins for the PIO instance.

This method repeatedly reconfigures the target state machine’s pin configuration and executes 'set' instructions to set values on up to 32 pins, before restoring the state machine’s pin configuration to what it was.

This method is provided as a convenience to set initial pin states, and should not be used against a state machine that is enabled.

Parameters

- pio The PIO instance; either pio0 or pio1
- sm State machine index (0..3) to use
- pin_values the pin values to set (if the corresponding bit in pin_mask is set)
- pin_mask a bit for each pin to indicate whether the corresponding pin_value for that pin should be applied.

3.13.7.41. pio_sm_set_wrap

```
static void pio_sm_set_wrap (PIO pio,
       uint sm,
       uint wrap_target,
       uint wrap)
```

Set the current wrap configuration for a state machine.

Parameters

- pio The PIO instance; either pio0 or pio1
- sm State machine index (0..3)
- wrap_target the instruction memory address to wrap to
- wrap the instruction memory address after which to set the program counter to wrap_target if the instruction does not itself update the program_counter

3.14. hardware_pll

Phase Locked Loop control APIs. More…

3.14.1. Functions

- `void pll_init (PLL pll, uint32_t ref_div, uint32_t vco_freq, uint32_t post_div1, uint8_t post_div2)`
  Initialise specified PLL. More…
- `void pll_deinit (PLL pll)`
  Release/uninitialise specified PLL. More…

3.14.2. Detailed Description

Phase Locked Loop control APIs.

There are two PLLs in RP2040. They are:

- pll_sys - Used to generate up to a 133MHz system clock
- pll_usb - Used to generate a 48MHz USB reference clock

For details on how the PLL’s are calculated, please refer to the RP2040 datasheet.

3.14.3. Function Documentation

3.14.3.1. pll_deinit

```
void pll_deinit (PLL pll)
```

Release/uninitialise specified PLL.

This will turn off the power to the specified PLL. Note this function does not currently check if the PLL is in use before powering it off so should be used with care.

Parameters

- `pll` pll_sys or pll_usb

3.14.3.2. pll_init

```
void pll_init (PLL pll,
       uint32_t ref_div,
       uint32_t vco_freq,
       uint32_t post_div1,
       uint8_t post_div2)
```

Initialise specified PLL.

Parameters

- `pll` pll_sys or pll_usb
- `ref_div` Input clock divider.
- `vco_freq` Requested output from the VCO (voltage controlled oscillator)
- `post_div1` Post Divider 1 - range 1-7. Must be >= post_div2
- `post_div2` Post Divider 2 - range 1-7

3.15. hardware_pwm

Hardware Pulse Width Modulation (PWM) API. More…

3.15.1. Enumerations

- `enum pwm_clkdiv_mode { PWM_DIV_FREE_RUNNING, PWM_DIV_B_HIGH, PWM_DIV_B_RISING, PWM_DIV_B_FALLING }`
  PWM Divider mode settings.

3.15.2. Functions

- `static uint pwm_gpio_to_slice_num (uint gpio)`
  Determine the PWM slice that is attached to the specified GPIO. More…
- `static uint pwm_gpio_to_channel (uint gpio)`
  Determine the PWM channel that is attached to the specified GPIO. More…
- `static void pwm_config_set_phase_correct (pwm_config *c, bool phase_correct)`
  Set phase correction in a PWM configuration. More…
- `static void pwm_config_set_clkdiv (pwm_config *c, float div)`
  Set clock divider in a PWM configuration. More…
- `static void pwm_config_set_clkdiv_int (pwm_config *c, uint div)`
  Set PWM clock divider in a PWM configuration. More…
- `static void pwm_config_set_clkdiv_mode (pwm_config *c, enum pwm_clkdiv_mode mode)`
  Set PWM counting mode in a PWM configuration. More…
- `static void pwm_config_set_output_polarity (pwm_config *c, bool a, bool b)`
  Set output polarity in a PWM configuration. More…

- `static void pwm_config_set_wrap (pwm_config *c, uint16_t wrap)`
  Set PWM counter wrap value in a PWM configuration. More...
- `static void pwm_init (uint slice_num, pwm_config *c, bool start)`
  Initialise a PWM with settings from a configuration object. More...
- `static pwm_config pwm_get_default_config ()`
  Get a set of default values for PWM configuration. More...
- `static void pwm_set_wrap (uint slice_num, uint16_t wrap)`
  Set the current PWM counter wrap value. More...
- `static void pwm_set_chan_level (uint slice_num, uint chan, uint16_t level)`
  Set the current PWM counter compare value for one channel. More...
- `static void pwm_set_both_levels (uint slice_num, uint16_t level_a, uint16_t level_b)`
  Set PWM counter compare values. More...
- `static void pwm_set_gpio_level (uint gpio, uint16_t level)`
  Helper function to set the PWM level for the slice and channel associated with a GPIO. More...
- `static int16_t pwm_get_counter (uint slice_num)`
  Get PWM counter. More...
- `static void pwm_set_counter (uint slice_num, uint16_t c)`
  Set PWM counter. More...
- `static void pwm_advance_count (uint slice_num)`
  Advance PWM count. More...
- `static void pwm_retard_count (uint slice_num)`
  Retard PWM count. More...
- `static void pwm_set_clkdiv_int_frac (uint slice_num, uint8_t integer, uint8_t fract)`
  Set PWM clock divider using an 8:4 fractional value. More...
- `static void pwm_set_clkdiv (uint slice_num, float divider)`
  Set PWM clock divider. More...
- `static void pwm_set_output_polarity (uint pwm, bool a, bool b)`
  Set PWM output polarity. More...
- `static void pwm_set_clkdiv_mode (uint slice_num, enum pwm_clkdiv_mode mode)`
  Set PWM divider mode. More...
- `static void pwm_set_phase_correct (uint slice_num, bool phase_correct)`
  Set PWM phase correct on/off. More...
- `static void pwm_set_enabled (uint slice_num, bool enabled)`
  Enable/Disable PWM. More...
- `static void pwm_set_mask_enabled (uint32_t mask)`
  Enable/Disable multiple PWM slices simultaneously. More...
- `static void pwm_set_irq_enabled (uint slice_num, bool enabled)`
  Enable PWM instance interrupt. More...
- `static void pwm_set_irq_mask_enabled (uint32_t slice_mask, bool enabled)`
  Enable multiple PWM instance interrupts. More...
- `static void pwm_clear_irq (uint slice_num)`
  Clear single PWM channel interrupt. More...
- `static int32_t pwm_get_irq_status_mask ()`
  Get PWM interrupt status, raw. More...
- `static void pwm_force_irq (uint slice_num)`

Force PWM interrupt. More…

3.15.3. Detailed Description

Hardware Pulse Width Modulation (PWM) API.

The RP2040 PWM block has 8 identical slices. Each slice can drive two PWM output signals, or measure the frequency or duty cycle of an input signal. This gives a total of up to 16 controllable PWM outputs. All 30 GPIOs can be driven by the PWM block

The PWM hardware functions by continuously comparing the input value to a free-running counter. This produces a toggling output where the amount of time spent at the high output level is proportional to the input value. The fraction of time spent at the high signal level is known as the duty cycle of the signal.

The default behaviour of a PWM slice is to count upward until the wrap value (pwm_config_set_wrap) is reached, and then immediately wrap to 0. PWM slices also offer a phase-correct mode, where the counter starts to count downward after reaching TOP, until it reaches 0 again.

Example

```
// Output PWM signals on pins 0 and 1

#include "pico/stdlib.h"
#include "hardware/pwm.h"

int main() {

    // Tell GPIO 0 and 1 they are allocated to the PWM
    gpio_set_function(0, GPIO_FUNC_PWM);
    gpio_set_function(1, GPIO_FUNC_PWM);

    // Find out which PWM slice is connected to GPIO 0 (it's slice 0)
    uint slice_num = pwm_gpio_to_slice_num(0);

    // Set period of 4 cycles (0 to 3 inclusive)
    pwm_set_wrap(slice_num, 3);
    // Set channel A output high for one cycle before dropping
    pwm_set_chan_level(slice_num, PWM_CHAN_A, 1);
    // Set initial B output high for three cycles before dropping
    pwm_set_chan_level(slice_num, PWM_CHAN_B, 3);
    // Set the PWM running
    pwm_set_enabled(slice_num, true);

    // Note we could also use pwm_set_gpio_level(gpio, x) which looks up the
    // correct slice and channel for a given GPIO.
}
```

3.15.4. Function Documentation

3.15.4.1. pwm_advance_count

`static void pwm_advance_count (uint slice_num)`

Advance PWM count.

Advance the phase of a running the counter by 1 count.

This function will return once the increment is complete.

Parameters

- `slice_num` PWM slice number

3.15.4.2. pwm_clear_irq

```
static void pwm_clear_irq (uint slice_num)
```

Clear single PWM channel interrupt.

Parameters

- `slice_num` PWM slice number

3.15.4.3. pwm_config_set_clkdiv

```
static void pwm_config_set_clkdiv (pwm_config *c,
       float div)
```

Set clock divider in a PWM configuration.

If the divide mode is free-running, the PWM counter runs at clk_sys / div. Otherwise, the divider reduces the rate of events seen on the B pin input (level or edge) before passing them on to the PWM counter.

Parameters

- `c` PWM configuration struct to modify
- `div` Value to divide counting rate by. Must be greater than or equal to 1.

3.15.4.4. pwm_config_set_clkdiv_int

```
static void pwm_config_set_clkdiv_int (pwm_config *c,
       uint div)
```

Set PWM clock divider in a PWM configuration.

If the divide mode is free-running, the PWM counter runs at clk_sys / div. Otherwise, the divider reduces the rate of events seen on the B pin input (level or edge) before passing them on to the PWM counter.

Parameters

- `c` PWM configuration struct to modify
- `div` integer value to reduce counting rate by. Must be greater than or equal to 1.

3.15.4.5. pwm_config_set_clkdiv_mode

```
static void pwm_config_set_clkdiv_mode (pwm_config *c,
       enum pwm_clkdiv_mode mode)
```

Set PWM counting mode in a PWM configuration.

Configure which event gates the operation of the fractional divider. The default is always-on (free-running PWM). Can also be configured to count on high level, rising edge or falling edge of the B pin input.

Parameters

- `c` PWM configuration struct to modify
- `mode` PWM divide/count mode

3.15.4.6. pwm_config_set_output_polarity

```
static void pwm_config_set_output_polarity (pwm_config *c,
       bool a,
       bool b)
```

Set output polarity in a PWM configuration.

Parameters

- `c` PWM configuration struct to modify
- `a` true to invert output A
- `b` true to invert output B

3.15.4.7. pwm_config_set_phase_correct

```
static void pwm_config_set_phase_correct (pwm_config *c,
       bool phase_correct)
```

Set phase correction in a PWM configuration.

Setting phase control to true means that instead of wrapping back to zero when the wrap point is reached, the PWM starts counting back down. The output frequency is halved when phase-correct mode is enabled.

Parameters

- `c` PWM configuration struct to modify
- `phase_correct` true to set phase correct modulation, false to set trailing edge

3.15.4.8. pwm_config_set_wrap

```
static void pwm_config_set_wrap (pwm_config *c,
       uint16_t wrap)
```

Set PWM counter wrap value in a PWM configuration.

Set the highest value the counter will reach before returning to 0. Also known as TOP.

Parameters

- `c` PWM configuration struct to modify
- `wrap` Value to set wrap to

3.15.4.9. pwm_force_irq

```
static void pwm_force_irq (uint slice_num)
```

Force PWM interrupt.

Parameters

- `slice_num` PWM slice number

3.15.4.10. pwm_get_counter

```
static int16_t pwm_get_counter (uint slice_num)
```

Get PWM counter.

Get current value of PWM counter

Parameters

- `slice_num` PWM slice number

Returns

- Current value of PWM counter

3.15.4.11. pwm_get_default_config

```
static pwm_config pwm_get_default_config ()
```

Get a set of default values for PWM configuration.

PWM config is free running at system clock speed, no phase correction, wrapping at 0xffff, with standard polarities for channels A and B.

Returns

- Set of default values.

3.15.4.12. pwm_get_irq_status_mask

```
static int32_t pwm_get_irq_status_mask ()
```

Get PWM interrupt status, raw.

Returns

- Bitmask of all PWM interrupts currently set

3.15.4.13. pwm_gpio_to_channel

```
static uint pwm_gpio_to_channel (uint gpio)
```

Determine the PWM channel that is attached to the specified GPIO.

Each slice 0 to 7 has two channels, A and B.

Returns

- The PWM channel that controls the specified GPIO.

3.15.4.14. pwm_gpio_to_slice_num

```
static uint pwm_gpio_to_slice_num (uint gpio)
```

Determine the PWM slice that is attached to the specified GPIO.

Returns

- The PWM slice number that controls the specified GPIO.

3.15.4.15. pwm_init

```
static void pwm_init (uint slice_num,
       pwm_config *c,
       bool start)
```

Initialise a PWM with settings from a configuration object.

Use the `pwm_get_default_config()` function to initialise a config structure, make changes as needed using the pwm_config_* functions, then call this function to set up the PWM.

Parameters

- `slice_num` PWM slice number
- `c` The configuration to use
- `start` If true the PWM will be started running once configured. If false you will need to start manually using pwm_set_enabled() or pwm_set_mask_enabled()

3.15.4.16. pwm_retard_count

```
static void pwm_retard_count (uint slice_num)
```

Retard PWM count.

Retard the phase of a running counter by 1 count

This function will return once the retardation is complete.

Parameters

- `slice_num` PWM slice number

3.15.4.17. pwm_set_both_levels

```
static void pwm_set_both_levels (uint slice_num,
       uint16_t level_a,
       uint16_t level_b)
```

Set PWM counter compare values.

Set the value of the PWM counter compare values, A and B

Parameters

- `slice_num` PWM slice number
- `level_a` Value to set compare A to. When the counter reaches this value the A output is deasserted
- `level_b` Value to set compare B to. When the counter reaches this value the B output is deasserted

3.15.4.18. pwm_set_chan_level

```
static void pwm_set_chan_level (uint slice_num,
       uint chan,
       uint16_t level)
```

Set the current PWM counter compare value for one channel.

Set the value of the PWM counter compare value, for either channel A or channel B

Parameters

- `slice_num` PWM slice number
- `chan` Which channel to update. 0 for A, 1 for B.
- `level` new level for the selected output

3.15.4.19. pwm_set_clkdiv

```
static void pwm_set_clkdiv (uint slice_num,
       float divider)
```

Set PWM clock divider.

Set the clock divider. Counter increment will be on sysclock divided by this value, taking in to account the gating.

Parameters

- `slice_num` PWM slice number
- `divider` Floating point clock divider, 1.f ⇐ value < 256.f

3.15.4.20. pwm_set_clkdiv_int_frac

```
static void pwm_set_clkdiv_int_frac (uint slice_num,
       uint8_t integer,
       uint8_t fract)
```

Set PWM clock divider using an 8:4 fractional value.

Set the clock divider. Counter increment will be on sysclock divided by this value, taking in to account the gating.

Parameters

- `slice_num` PWM slice number
- `integer` 8 bit integer part of the clock divider
- `fract` 4 bit fractional part of the clock divider

3.15.4.21. pwm_set_clkdiv_mode

```
static void pwm_set_clkdiv_mode (uint slice_num,
       enum pwm_clkdiv_mode mode)
```

Set PWM divider mode.

Parameters

- `slice_num` PWM slice number
- `mode` Required divider mode

3.15.4.22. pwm_set_counter

```
static void pwm_set_counter (uint slice_num,
       uint16_t c)
```

Set PWM counter.

Set the value of the PWM counter

Parameters

- `slice_num` PWM slice number
- `c` Value to set the PWM counter to

3.15.4.23. pwm_set_enabled

```
static void pwm_set_enabled (uint slice_num,
       bool enabled)
```

Enable/Disable PWM.

Parameters

- `slice_num` PWM slice number

- `enabled` true to enable the specified PWM, false to disable

3.15.4.24. pwm_set_gpio_level

```
static void pwm_set_gpio_level (uint gpio,
       uint16_t level)
```

Helper function to set the PWM level for the slice and channel associated with a GPIO.

Look up the correct slice (0 to 7) and channel (A or B) for a given GPIO, and update the corresponding counter-compare field.

This PWM slice should already have been configured and set running. Also be careful of multiple GPIOs mapping to the same slice and channel (if GPIOs have a difference of 16).

Parameters

- `gpio` GPIO to set level of
- `level` PWM level for this GPIO

3.15.4.25. pwm_set_irq_enabled

```
static void pwm_set_irq_enabled (uint slice_num,
       bool enabled)
```

Enable PWM instance interrupt.

Used to enable a single PWM instance interrupt

Parameters

- `slice_num` PWM block to enable/disable
- `enabled` true to enable, false to disable

3.15.4.26. pwm_set_irq_mask_enabled

```
static void pwm_set_irq_mask_enabled (uint32_t slice_mask,
       bool enabled)
```

Enable multiple PWM instance interrupts.

Use this to enable multiple PWM interrupts at once.

Parameters

- `slice_mask` Bitmask of all the blocks to enable/disable. Channel 0 = bit 0, channel 1 = bit 1 etc.
- `enabled` true to enable, false to disable

3.15.4.27. pwm_set_mask_enabled

```
static void pwm_set_mask_enabled (uint32_t mask)
```

Enable/Disable multiple PWM slices simultaneously.

Parameters

- `mask` Bitmap of PWMs to enable/disable. Bits 0 to 7 enable slices 0-7 respectively

3.15.4.28. pwm_set_output_polarity

```
static void pwm_set_output_polarity (uint pwm,
       bool a,
       bool b)
```

Set PWM output polarity.

Parameters

- `slice_num` PWM slice number
- `a` true to invert output A
- `b` true to invert output B

3.15.4.29. pwm_set_phase_correct

```
static void pwm_set_phase_correct (uint slice_num,
       bool phase_correct)
```

Set PWM phase correct on/off.

Setting phase control to true means that instead of wrapping back to zero when the wrap point is reached, the PWM starts counting back down. The output frequency is halved when phase-correct mode is enabled.

Parameters

- `slice_num` PWM slice number
- `phase_correct` true to set phase correct modulation, false to set trailing edge

3.15.4.30. pwm_set_wrap

```
static void pwm_set_wrap (uint slice_num,
       uint16_t wrap)
```

Set the current PWM counter wrap value.

Set the highest value the counter will reach before returning to 0. Also known as TOP.

Parameters

- `slice_num` PWM slice number
- `wrap` Value to set wrap to

3.16. hardware_resets

Hardware Reset API. More…

3.16.1. Functions

- `static void reset_block (uint32_t bits)`
  Reset the specified HW blocks. More…
- `static void unreset_block (uint32_t bits)`
  bring specified HW blocks out of reset More…
- `static void unreset_block_wait (uint32_t bits)`
  Bring specified HW blocks out of reset and wait for completion. More…

3.16.2. Detailed Description

Hardware Reset API.

The reset controller allows software control of the resets to all of the peripherals that are not critical to boot the processor in the RP2040.

reset_bitmask

Multiple blocks are referred to using a bitmask as follows:

Block to reset	Bit
USB	24
UART 1	23
UART 0	22
Timer	21
TB Manager	20
SysInfo	19
System Config	18
SPI 1	17
SPI 0	16
RTC	15
PWM	14
PLL USB	13
PLL System	12
PIO 1	11
PIO 0	10
Pads - QSPI	9
Pads - bank 0	8
JTAG	7
IO Bank 1	6
IO Bank 0	5
I2C 1	4
I2C 0	3
DMA	2
Bus Control	1
ADC 0	0

Example

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/resets.h"

// tag::hello_reset[]
```

```
int main() {
    stdio_init_all();

    printf("Hello, reset!\n");

    // Put the PWM block into reset
    reset_block(RESETS_RESET_PWM_RST_N_BITS);

    // And bring it out
    unreset_block_wait(RESETS_RESET_PWM_RST_N_BITS);

    // Put the PWM and RTC block into reset
    reset_block(RESETS_RESET_PWM_RST_N_BITS | RESETS_RESET_RTC_RST_N_BITS);

    // Wait for both to come out of reset
    unreset_block_wait(RESETS_RESET_PWM_RST_N_BITS | RESETS_RESET_RTC_RST_N_BITS);

    return 0;
}
// end::hello_reset[]
```

3.16.3. Function Documentation

3.16.3.1. reset_block

`static void reset_block (uint32_t bits)`

Reset the specified HW blocks.

Parameters

- `bits` Bit pattern indicating blocks to reset. See reset_bitmask

3.16.3.2. unreset_block

`static void unreset_block (uint32_t bits)`

bring specified HW blocks out of reset

Parameters

- `bits` Bit pattern indicating blocks to unreset. See reset_bitmask

3.16.3.3. unreset_block_wait

`static void unreset_block_wait (uint32_t bits)`

Bring specified HW blocks out of reset and wait for completion.

Parameters

- `bits` Bit pattern indicating blocks to unreset. See reset_bitmask

3.17. hardware_rosc

3.18. hardware_rtc

Hardware Real Time Clock API. More…

3.18.1. Functions

- `void rtc_init (void)`
  Initialise the RTC system.
- `bool rtc_set_datetime (datetime_t *t)`
  Set the RTC to the specified time. More…
- `bool rtc_get_datetime (datetime_t *t)`
  Get the current time from the RTC. More…

3.18.2. Detailed Description

Hardware Real Time Clock API.

The RTC keeps track of time in human readable format and generates events when the time is equal to a preset value. Think of a digital clock, not epoch time used by most computers. There are seven fields, one each for year (12 bit), month (4 bit), day (5 bit), day of the week (3 bit), hour (5 bit) minute (6 bit) and second (6 bit), storing the data in binary format.

See also

- datetime_t

Example

```
#include <stdio.h>
#include "hardware/rtc.h"
#include "pico/stdlib.h"
#include "pico/util/datetime.h"

int main() {
    stdio_init_all();
    printf("Hello RTC!\n");

    char datetime_buf[256];
    char *datetime_str = &datetime_buf[0];

    // Start on Friday 5th of June 2020 15:45:00
    datetime_t t = {
            .year  = 2020,
            .month = 06,
            .day   = 05,
            .dotw  = 5, // 0 is Sunday, so 5 is Friday
            .hour  = 15,
            .min   = 45,
            .sec   = 00
    };

    // Start the RTC
    rtc_init();
    rtc_set_datetime(&t);

    // Print the time
    while (true) {
        rtc_get_datetime(&t);
        datetime_to_str(datetime_str, sizeof(datetime_buf), &t);
        printf("\r%s      ", datetime_str);
        sleep_ms(100);
    }

    return 0;
}
```

3.18.3. Function Documentation

3.18.3.1. rtc_get_datetime

`bool rtc_get_datetime (datetime_t *t)`

Get the current time from the RTC.

Parameters

- `t` Pointer to a datetime_t structure to receive the current RTC time

Returns

- true if datetime is valid, false if the RTC is not running.

3.18.3.2. rtc_init

`void rtc_init (void)`

Initialise the RTC system.

3.18.3.3. rtc_set_datetime

`bool rtc_set_datetime (datetime_t *t)`

Set the RTC to the specified time.

Parameters

- `t` Pointer to a datetime_t structure contains time to set

Returns

- true if set, falsw if the passed in datetime was invalid.

3.19. hardware_sleep

3.20. hardware_spi

Hardware SPI API. More…

3.20.1. Macros

- `#define spi0 ((spi_inst_t * const)spi0_hw)` More…
- `#define spi1 ((spi_inst_t * const)spi1_hw)` More…

3.20.2. Functions

- `void spi_init (spi_inst_t *spi, uint baudrate)`
  Initialise SPI instances. More…
- `void spi_deinit (spi_inst_t *spi)`
  Deinitialise SPI instances. More…
- `uint spi_set_baudrate (spi_inst_t *spi, uint baudrate)`
  Set SPI baudrate. More…
- `static uint spi_get_index (spi_inst_t *spi)`
  Convert I2c instance to hardware instance number. More…
- `static void spi_set_format (spi_inst_t *spi, uint data_bits, spi_cpol_t cpol, spi_cpha_t cpha, spi_order_t order)`
  Configure SPI. More…
- `static void spi_set_slave (spi_inst_t *spi, bool slave)`
  Set SPI master/slave. More…
- `static size_t spi_is_writable (spi_inst_t *spi)`
  Check whether a write can be done on SPI device. More…
- `static size_t spi_is_readable (spi_inst_t *spi)`
  Check whether a read can be done on SPI device. More…
- `int spi_write_read_blocking (spi_inst_t *spi, const uint8_t *src, uint8_t *dst, size_t len)`
  Write/Read to/from an SPI device. More…
- `int spi_write_blocking (spi_inst_t *spi, const uint8_t *src, size_t len)`
  Write to an SPI device, blocking. More…
- `int spi_read_blocking (spi_inst_t *spi, uint8_t repeated_tx_data, uint8_t *dst, size_t len)`
  Read from an SPI device. More…
- `int spi_write16_read16_blocking (spi_inst_t *spi, const uint16_t *src, uint16_t *dst, size_t len)`
  Write/Read half words to/from an SPI device. More…
- `int spi_write16_blocking (spi_inst_t *spi, const uint16_t *src, size_t len)`
  Write to an SPI device. More…
- `int spi_read16_blocking (spi_inst_t *spi, uint16_t repeated_tx_data, uint16_t *dst, size_t len)`
  Read from an SPI device. More…

3.20.3. Detailed Description

Hardware SPI API.

RP2040 has 2 identical instances of an Serial Peripheral Interface (SPI) controller.

The PrimeCell SSP is a master or slave interface for synchronous serial communication with peripheral devices that have Motorola SPI, National Semiconductor Microwire, or Texas Instruments synchronous serial interfaces.

Controller can be defined as master or slave using the spi_set_slave function.

Each controller can be connected to a number of GPIO pins, see the datasheet GPIO function selection table for more information.

3.20.4. Function Documentation

3.20.4.1. spi_deinit

```
void spi_deinit (spi_inst_t *spi)
```

Deinitialise SPI instances.

Puts the SPI into a disabled state. Init will need to be called to reenable the device functions.

Parameters

- `spi` SPI instance specifier, either spi0 or spi1

3.20.4.2. spi_get_index

```
static uint spi_get_index (spi_inst_t *spi)
```

Convert I2c instance to hardware instance number.

Parameters

- `spi` SPI instance

Returns

- Number of SPI, 0 or 1.

3.20.4.3. spi_init

```
void spi_init (spi_inst_t *spi,
       uint baudrate)
```

Initialise SPI instances.

Puts the SPI into a known state, and enable it. Must be called before other functions.

Parameters

- `spi` SPI instance specifier, either spi0 or spi1
- `baudrate` Baudrate required in Hz

3.20.4.4. spi_is_readable

```
static size_t spi_is_readable (spi_inst_t *spi)
```

Check whether a read can be done on SPI device.

Parameters

- `spi` SPI instance specifier, either spi0 or spi1

Returns

- Non-zero if a read is possible i.e. data is present

3.20.4.5. spi_is_writable

```
static size_t spi_is_writable (spi_inst_t *spi)
```

Check whether a write can be done on SPI device.

Parameters

- `spi` SPI instance specifier, either spi0 or spi1

Returns

- 0 if no space is available to write. Non-zero if a write is possible

3.20.4.6. spi_read16_blocking

```
int spi_read16_blocking (spi_inst_t *spi,
       uint16_t repeated_tx_data,
       uint16_t *dst,
       size_t len)
```

Read from an SPI device.

Read `len` halfwords from SPI to `dst`. Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate. `repeated_tx_data` is output repeatedly on SO as data is read in from SI. Generally this can be 0, but some devices require a specific value here, e.g. SD cards expect 0xff

Parameters

- `spi` SPI instance specifier, either spi0 or spi1
- `repeated_tx_data` Buffer of data to write
- `dst` Buffer for read data
- `len` Length of buffer dst in halfwords

Returns

- Number of bytes written/read

3.20.4.7. spi_read_blocking

```
int spi_read_blocking (spi_inst_t *spi,
       uint8_t repeated_tx_data,
       uint8_t *dst,
       size_t len)
```

Read from an SPI device.

Read `len` bytes from SPI to `dst`. Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate. `repeated_tx_data` is output repeatedly on SO as data is read in from SI. Generally this can be 0, but some devices require a specific value here, e.g. SD cards expect 0xff

Parameters

- `spi` SPI instance specifier, either spi0 or spi1
- `repeated_tx_data` Buffer of data to write
- `dst` Buffer for read data
- `len` Length of buffer dst

Returns

- Number of bytes written/read

3.20.4.8. spi_set_baudrate

```
uint spi_set_baudrate (spi_inst_t *spi,
       uint baudrate)
```

Set SPI baudrate.

Set SPI frequency as close as possible to baudrate, and return the actual achieved rate.

Parameters

- spi SPI instance specifier, either spi0 or spi1
- baudrate Baudrate required in Hz, should be capable of a bitrate of at least 2Mbps, or higher, depending on system clock settings.

Returns

- The actual baudrate set

3.20.4.9. spi_set_format

```
static void spi_set_format (spi_inst_t *spi,
       uint data_bits,
       spi_cpol_t cpol,
       spi_cpha_t cpha,
       spi_order_t order)
```

Configure SPI.

Configure how the SPI serialises and deserialises data on the wire

Parameters

- spi SPI instance specifier, either spi0 or spi1
- data_bits Number of data bits per transfer. Valid values 4..16.
- cpol SSPCLKOUT polarity, applicable to Motorola SPI frame format only.
- cpha SSPCLKOUT phase, applicable to Motorola SPI frame format only
- order Must be SPI_MSB_FIRST, no other values supported on the PL022

3.20.4.10. spi_set_slave

```
static void spi_set_slave (spi_inst_t *spi,
       bool slave)
```

Set SPI master/slave.

Configure the SPI for master- or slave-mode operation. By default, `spi_init()` sets master-mode.

Parameters

- spi SPI instance specifier, either spi0 or spi1
- slave true to set SPI device as a slave device, false for master.

3.20.4.11. spi_write16_blocking

```
int spi_write16_blocking (spi_inst_t *spi,
       const uint16_t *src,
       size_t len)
```

Write to an SPI device.

Write `len` halfwords from `src` to SPI. Discard any data received back. Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate.

Parameters

- spi SPI instance specifier, either spi0 or spi1
- src Buffer of data to write
- len Length of buffers

Returns

- Number of bytes written/read

3.20.4.12. spi_write16_read16_blocking

```
int spi_write16_read16_blocking (spi_inst_t *spi,
       const uint16_t *src,
       uint16_t *dst,
       size_t len)
```

Write/Read half words to/from an SPI device.

Write `len` halfwords from `src` to SPI. Simultaneously read `len` halfwords from SPI to `dst`. Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate.

Parameters

- `spi` SPI instance specifier, either spi0 or spi1
- `src` Buffer of data to write
- `dst` Buffer for read data
- `len` Length of BOTH buffers in halfwords

Returns

- Number of bytes written/read

3.20.4.13. spi_write_blocking

```
int spi_write_blocking (spi_inst_t *spi,
       const uint8_t *src,
       size_t len)
```

Write to an SPI device, blocking.

Write `len` bytes from `src` to SPI, and discard any data received back Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate.

Parameters

- `spi` SPI instance specifier, either spi0 or spi1
- `src` Buffer of data to write
- `len` Length of src

Returns

- Number of bytes written/read

3.20.4.14. spi_write_read_blocking

```
int spi_write_read_blocking (spi_inst_t *spi,
       const uint8_t *src,
       uint8_t *dst,
       size_t len)
```

Write/Read to/from an SPI device.

Write `len` bytes from `src` to SPI. Simultaneously read `len` bytes from SPI to `dst`. Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate.

Parameters

- `spi` SPI instance specifier, either spi0 or spi1
- `src` Buffer of data to write
- `dst` Buffer for read data
- `len` Length of BOTH buffers

Returns

- Number of bytes written/read

3.21. hardware_sync

Low level hardware spin-lock, barrier and processor event API. More…

3.21.1. Typedefs

- `typedef uint32_t spin_lock_t`
  A spin lock identifier.

3.21.2. Functions

- `static void __sev ()`
  Insert a SEV instruction in to the code path. More…
- `static void __wfe ()`
  Insert a WFE instruction in to the code path. More…
- `static void __wfi ()`
  Insert a WFI instruction in to the code path. More…
- `static void __dmb ()`
  Insert a DMB instruction in to the code path. More…
- `static void __isb ()`
  Insert a ISB instruction in to the code path. More…
- `static void __mem_fence_acquire ()`
  Acquire a memory fence.
- `static void __mem_fence_release ()`
  Release a memory fence.
- `static uint32_t save_and_disable_interrupts ()`
  Save and disable interrupts. More…
- `static void restore_interrupts (uint32_t status)`
  Restore interrupts to a specified state. More…
- `static spin_lock_t * spin_lock_instance (uint lock_num)`
  Get HW Spinlock instance from number. More…
- `static uint spin_lock_get_num (spin_lock_t *lock)`
  Get HW Spinlock number from instance. More…
- `static void spin_lock_unsafe_blocking (spin_lock_t *lock)`
  Acquire a spin lock without disabling interrupts (hence unsafe) More…
- `static void spin_unlock_unsafe (spin_lock_t *lock)`
  Release a spin lock without re-enabling interrupts. More…

- `static uint32_t spin_lock_blocking (spin_lock_t *lock)`
  Acquire a spin lock safely. More…
- `static bool is_spin_locked (const spin_lock_t *lock)`
  Check to see if a spinlock is currently acquired elsewhere. More…
- `static void spin_unlock (spin_lock_t *lock, uint32_t saved_irq)`
  Release a spin lock safely. More…
- `static uint get_core_num ()`
  Get the current core number. More…
- `spin_lock_t * spin_lock_init (uint lock_num)`
  Initialise a spin lock. More…
- `void spin_locks_reset (void)`
  Release all spin locks.
- `void spin_lock_claim (uint lock_num)`
  Mark a spin lock as used. More…
- `void spin_lock_claim_mask (uint32_t lock_num_mask)`
  Mark multiple spin locks as used. More…
- `void spin_lock_unclaim (uint lock_num)`
  Mark a spin lock as no longer used. More…
- `int spin_lock_claim_unused (bool required)`
  Claim a free spin lock. More…

3.21.3. Detailed Description

Low level hardware spin-lock, barrier and processor event API.

Functions for synchronisation between core's, HW, etc

The RP2040 provides 32 hardware spin locks, which can be used to manage mutually-exclusive access to shared software resources.

spin locks 0-15 are currently reserved for fixed uses by the SDK - i.e. if you use them other functionality may break or not function optimally

3.21.4. Function Documentation

3.21.4.1. __dmb

`static void __dmb ()`

Insert a DMB instruction in to the code path.

The DMB (data memory barrier) acts as a memory barrier, all memory accesses prior to this instruction will be observed before any explicit access after the instruction.

3.21.4.2. __isb

`static void __isb ()`

Insert a ISB instruction in to the code path.

ISB acts as an instruction synchronization barrier. It flushes the pipeline of the processor, so that all instructions following the ISB are fetched from cache or memory again, after the ISB instruction has been completed.

3.21.4.3. __mem_fence_acquire

`static void __mem_fence_acquire ()`

Acquire a memory fence.

3.21.4.4. __mem_fence_release

`static void __mem_fence_release ()`

Release a memory fence.

3.21.4.5. __sev

`static void __sev ()`

Insert a SEV instruction in to the code path.

The SEV (send event) instruction sends an event to both cores.

3.21.4.6. __wfe

`static void __wfe ()`

Insert a WFE instruction in to the code path.

The WFE (wait for event) instruction waits until one of a number of events occurs, including events signaled by the SEV instruction on either core.

3.21.4.7. __wfi

`static void __wfi ()`

Insert a WFI instruction in to the code path.

The WFI (wait for interrupt) instruction waits for a interrupt to wake up the core.

3.21.4.8. get_core_num

`static uint get_core_num ()`

Get the current core number.

Returns

- The core number the call was made from

3.21.4.9. is_spin_locked

`static bool is_spin_locked (const spin_lock_t *lock)`

Check to see if a spinlock is currently acquired elsewhere.

Parameters

- `lock` Spinlock instance

3.21.4.10. restore_interrupts

`static void restore_interrupts (uint32_t status)`

Restore interrupts to a specified state.

Parameters

- `status` Previous interrupt status from save_and_disable_interrupts()

3.21.4.11. save_and_disable_interrupts

`static uint32_t save_and_disable_interrupts ()`

Save and disable interrupts.

Returns

- The prior interrupt enable status for restoration later via `restore_interrupts()`

3.21.4.12. spin_lock_blocking

`static uint32_t spin_lock_blocking (spin_lock_t *lock)`

Acquire a spin lock safely.

This function will disable interrupts prior to acquiring the spinlock

Parameters

- `lock` Spinlock instance

Returns

- interrupt status to be used when unlocking, to restore to original state

3.21.4.13. spin_lock_claim

`void spin_lock_claim (uint lock_num)`

Mark a spin lock as used.

Method for cooperative claiming of hardware. Will cause a panic if the spin lock is already claimed. Use of this method by libraries detects accidental configurations that would fail in unpredictable ways.

Parameters

- `lock_num` the spin lock number

3.21.4.14. spin_lock_claim_mask

`void spin_lock_claim_mask (uint32_t lock_num_mask)`

Mark multiple spin locks as used.

Method for cooperative claiming of hardware. Will cause a panic if any of the spin locks are already claimed. Use of this method by libraries detects accidental configurations that would fail in unpredictable ways.

Parameters

- `lock_num_mask` Bitfield of all required spin locks to claim (bit 0 == spin lock 0, bit 1 == spin lock 1 etc)

3.21.4.15. spin_lock_claim_unused

`int spin_lock_claim_unused (bool required)`

Claim a free spin lock.

Parameters

- `required` if true the function will panic if none are aviailable

Returns

- the spin lock number or -1 if required was false, and none were free

3.21.4.16. spin_lock_get_num

`static uint spin_lock_get_num (spin_lock_t *lock)`

Get HW Spinlock number from instance.

Parameters

- `lock` The Spinlock instance

Returns

- The Spinlock ID

3.21.4.17. spin_lock_init

`spin_lock_t* spin_lock_init (uint lock_num)`

Initialise a spin lock.

The spin lock is initially unlocked

Parameters

- `lock_num` The spin lock number

Returns

- The spin lock instance

3.21.4.18. spin_lock_instance

`static spin_lock_t* spin_lock_instance (uint lock_num)`

Get HW Spinlock instance from number.

Parameters

- `lock_num` Spinlock ID

Returns

- The spinlock instance

3.21.4.19. spin_lock_unclaim

`void spin_lock_unclaim (uint lock_num)`

Mark a spin lock as no longer used.

Method for cooperative claiming of hardware.

Parameters

- `lock_num` the spin lock number to release

3.21.4.20. spin_lock_unsafe_blocking

`static void spin_lock_unsafe_blocking (spin_lock_t *lock)`

Acquire a spin lock without disabling interrupts (hence unsafe)

Parameters

- `lock` Spinlock instance

3.21.4.21. spin_locks_reset

`void spin_locks_reset (void)`

Release all spin locks.

3.21.4.22. spin_unlock

```
static void spin_unlock (spin_lock_t *lock,
       uint32_t saved_irq)
```

Release a spin lock safely.

This function will re-enable interrupts according to the parameters.

Parameters

- `lock` Spinlock instance
- `saved_irq` Return value from the spin_lock_blocking() function.

Returns

- interrupt status to be used when unlocking, to restore to original state

See also

- spin_lock_blocking()

3.21.4.23. spin_unlock_unsafe

`static void spin_unlock_unsafe (spin_lock_t *lock)`

Release a spin lock without re-enabling interrupts.

Parameters

- `lock` Spinlock instance

3.22. hardware_timer

Low-level hardware timer API. More…

3.22.1. Typedefs

- `typedef void(* hardware_alarm_callback_t )(uint alarm_num)` More…

3.22.2. Functions

- `static uint32_t time_us_32 ()`
  Return a 32 bit timestamp value in microseconds. More…
- `uint64_t time_us_64 ()`
  Return the current 64 bit timestamp value in microseconds. More…
- `void busy_wait_us_32 (uint32_t delay_us)`
  Busy wait wasting cycles for the given (32 bit) number of microseconds. More…
- `void busy_wait_us (uint64_t delay_us)`
  Busy wait wasting cycles for the given (64 bit) number of microseconds. More…
- `void busy_wait_until (absolute_time_t t)`
  Busy wait wasting cycles until after the specified timestamp. More…
- `static bool time_reached (absolute_time_t t)`
  Check if the specified timestamp has been reached. More…
- `void hardware_alarm_claim (uint alarm_num)`
  cooperatively claim the use of this hardware alarm_num More…
- `void hardware_alarm_unclaim (uint alarm_num)`
  cooperatively release the claim on use of this hardware alarm_num More…
- `void hardware_alarm_set_callback (uint alarm_num, hardware_alarm_callback_t callback)`
  Enable/Disable a callback for a hardware timer on this core. More…

3.22.3. Detailed Description

Low-level hardware timer API.

This API provides medium level access to the timer HW. See also pico_time which provides higher levels functionality using the hardware timer.

The timer peripheral on RP2040 supports the following features:

- single 64-bit counter, incrementing once per microsecond
- Latching two-stage read of counter, for race-free read over 32 bit bus
- Four alarms: match on the lower 32 bits of counter, IRQ on match.

By default the timer uses a one microsecond reference that is generated in the Watchdog (see Section 4.8.2) which is derived from the clk_ref.

The timer has 4 alarms, and can output a separate interrupt for each alarm. The alarms match on the lower 32 bits of the 64 bit counter which means they can be fired a maximum of 2^32 microseconds into the future. This is equivalent to:

- 2^32 ÷ 10^6: ~4295 seconds
- 4295 ÷ 60: ~72 minutes

The timer is expected to be used for short sleeps, if you want a longer alarm see the hardware_rtc functions.

Example

```
#include <stdio.h>
#include "pico/stdlib.h"

volatile bool timer_fired = false;

int64_t alarm_callback(alarm_id_t id, void *user_data) {
    printf("Timer %d fired!\n", (int) id);
    timer_fired = true;
    // Can return a value here in us to fire in the future
    return 0;
}

bool repeating_timer_callback(struct repeating_timer *t) {
    printf("Repeat at %lld\n", time_us_64());
    return true;
}

int main() {
    stdio_init_all();
    printf("Hello Timer!\n");

    // Call alarm_callback in 2 seconds
    add_alarm_in_ms(2000, alarm_callback, NULL, false);

    // Wait for alarm callback to set timer_fired
    while (!timer_fired) {
        tight_loop_contents();
    }

    // Create a repeating timer that calls repeating_timer_callback.
    // If the delay is > 0 then this is the delay between the previous callback ending and the next starting.
    // If the delay is negative (see below) then the next call to the callback will be exactly 500ms after the
    // start of the call to the last callback
    struct repeating_timer timer;
    add_repeating_timer_ms(500, repeating_timer_callback, NULL, &timer);
    sleep_ms(3000);
    bool cancelled = cancel_repeating_timer(&timer);
    printf("cancelled... %d\n", cancelled);
    sleep_ms(2000);

    // Negative delay so means we will call repeating_timer_callback, and call it again
    // 500ms later regardless of how long the callback took to execute
    add_repeating_timer_ms(-500, repeating_timer_callback, NULL, &timer);
    sleep_ms(3000);
    cancelled = cancel_repeating_timer(&timer);
    printf("cancelled... %d\n", cancelled);
    sleep_ms(2000);
    printf("Done\n");
    return 0;
}
```

See also

- pico_time

3.22.4. Function Documentation

3.22.4.1. busy_wait_until

`void busy_wait_until (absolute_time_t t)`

Busy wait wasting cycles until after the specified timestamp.

Parameters

- `t` Absolute time to wait until

3.22.4.2. busy_wait_us

```
void busy_wait_us (uint64_t delay_us)
```

Busy wait wasting cycles for the given (64 bit) number of microseconds.

Parameters

- `delay_us` delay amount

3.22.4.3. busy_wait_us_32

```
void busy_wait_us_32 (uint32_t delay_us)
```

Busy wait wasting cycles for the given (32 bit) number of microseconds.

Busy wait wasting cycles for the given (32 bit) number of microseconds.

Parameters

- `delay_us` delay amount

3.22.4.4. hardware_alarm_claim

```
void hardware_alarm_claim (uint alarm_num)
```

cooperatively claim the use of this hardware alarm_num

This method hard asserts if the hardware alarm is currently claimed.

Parameters

- `alarm_num` the hardware alarm to claim

See also

- hardware_claiming

3.22.4.5. hardware_alarm_set_callback

```
void hardware_alarm_set_callback (uint alarm_num,
       hardware_alarm_callback_t callback)
```

Enable/Disable a callback for a hardware timer on this core.

This method enables/disables the alarm IRQ for the specified hardware alarm on the calling core, and set the specified callback to be associated with that alarm.

This callback will be used for the timeout set via hardware_alarm_set_target

Parameters

- `alarm_num` the hardware alarm number
- `callback` the callback to install, or NULL to unset

See also

- hardware_alarm_set_target

3.22.4.6. hardware_alarm_unclaim

`void hardware_alarm_unclaim (uint alarm_num)`

cooperatively release the claim on use of this hardware alarm_num

Parameters

- `alarm_num` the hardware alarm to unclaim

See also

- hardware_claiming

3.22.4.7. time_reached

`static bool time_reached (absolute_time_t t)`

Check if the specified timestamp has been reached.

Parameters

- `t` Absolute time to compare against current time

Returns

- true if it is now after the specified timestamp

3.22.4.8. time_us_32

`static uint32_t time_us_32 ()`

Return a 32 bit timestamp value in microseconds.

Returns the low 32 bits of the hardware timer.

Returns

- the 32 bit timestamp

3.22.4.9. time_us_64

`uint64_t time_us_64 ()`

Return the current 64 bit timestamp value in microseconds.

Returns the full 64 bits of the hardware timer. The `pico_time` and other functions rely on the fact that this value monotonically increases from power up. As such it is expected that this value counts upwards and never wraps (we apologize for introducing a potential year 5851444 bug).

Return the current 64 bit timestamp value in microseconds.

Returns

- the 64 bit timestamp

3.23. hardware_uart

Hardware UART API. More…

3.23.1. Enumerations

- `enum uart_parity_t { UART_PARITY_NONE, UART_PARITY_EVEN, UART_PARITY_ODD }`
  UART Parity enumeration.

3.23.2. Functions

- `static uint uart_get_index (uart_inst_t *uart)`
  Convert UART instance to hardware instance number. More…
- `void uart_init (uart_inst_t *uart, uint baudrate)`
  Initialise a UART. More…
- `void uart_deinit (uart_inst_t *uart)`
  DeInitialise a UART. More…
- `uint uart_set_baudrate (uart_inst_t *uart, uint baudrate)`
  Set UART baud rate. More…
- `static void uart_set_hw_flow (uart_inst_t *uart, bool cts, bool rts)`
  Set UART flow control CTS/RTS. More…
- `static void uart_set_format (uart_inst_t *uart, uint data_bits, uint stop_bits, uart_parity_t parity)`
  Set UART data format. More…
- `static void uart_set_irq_enables (uart_inst_t *uart, bool rx_has_data, bool tx_needs_data)`
  Setup UART interrupts. More…
- `static bool uart_is_enabled (uart_inst_t *uart)`
  Test if specific UART is enabled. More…
- `static void uart_set_fifo_enabled (uart_inst_t *uart, bool enabled)`
  Enable/Disable the FIFOs on specified UART. More…
- `static bool uart_is_writable (uart_inst_t *uart)`
  Determine if space is available available in TX FIFO. More…
- `static void uart_tx_wait_blocking (uart_inst_t *uart)`
  Wait for the UART TX fifo to be drained. More…
- `static bool uart_is_readable (uart_inst_t *uart)`
  Determine whether data is waiting in the RX FIFO. More…
- `static void uart_write_blocking (uart_inst_t *uart, const uint8_t *src, size_t len)`
  Write to the UART for transmission. More…
- `static void uart_read_blocking (uart_inst_t *uart, uint8_t *dst, size_t len)`
  Read from the UART. More…
- `static void uart_putc_raw (uart_inst_t *uart, char c)`
  Write single character to UART for transmission. More…
- `static void uart_putc (uart_inst_t *uart, char c)`
  Write single character to UART for transmission, with optional CR/LF conversions. More…
- `static void uart_puts (uart_inst_t *uart, const char *s)`
  Write string to UART for transmission, doing any CR/LF conversions. More…
- `static char uart_getc (uart_inst_t *uart)`
  Read a single character to UART. More…
- `static void uart_set_break (uart_inst_t *uart, bool en)`
  Assert a break condition on the UART transmission. More…

- `void uart_set_crlf (uart_inst_t *uart, bool crlf)`
  Set CR/LF conversion on UART. More...

3.23.3. Variables

- `uart_inst_t const *uart0`
  Identifier for UART instance 0. More...
- `uart_inst_t const *uart1`
  Identifier for UART instance 1.

3.23.4. Detailed Description

Hardware UART API.

RP2040 has 2 identical instances of a UART peripheral, based on the ARM PL011. Each UART can be connected to a number of GPIO pins as defined in the GPIO muxing.

Only the TX, RX, RTS, and CTS signals are connected, meaning that the modem mode and IrDA mode of the PL011 are not supported.

Example

```
int main() {

   // Initialise UART 0
   uart_init(uart0, 115200);

   // Set the GPIO pin mux to the UART - 0 is TX, 1 is RX
   gpio_set_function(0, GPIO_FUNC_UART);
   gpio_set_function(1, GPIO_FUNC_UART);

   uart_puts(uart0, "Hello world!");
}
```

3.23.5. Function Documentation

3.23.5.1. uart_deinit

`void uart_deinit (uart_inst_t *uart)`

DeInitialise a UART.

Disable the UART if it is no longer used. Must be reinitialised before being used again.

Parameters

- `uart` UART instance. uart0 or uart1

3.23.5.2. uart_get_index

`static uint uart_get_index (uart_inst_t *uart)`

Convert UART instance to hardware instance number.

Parameters

- uart UART instance

Returns

- Number of UART, 0 or 1.

3.23.5.3. uart_getc

```
static char uart_getc (uart_inst_t *uart)
```

Read a single character to UART.

This function will block until the character has been read

Parameters

- uart UART instance. uart0 or uart1

Returns

- The character read.

3.23.5.4. uart_init

```
void uart_init (uart_inst_t *uart,
        uint baudrate)
```

Initialise a UART.

Put the UART into a known state, and enable it. Must be called before other functions.

Parameters

- uart UART instance. uart0 or uart1
- baudrate Baudrate of UART in Hz

3.23.5.5. uart_is_enabled

```
static bool uart_is_enabled (uart_inst_t *uart)
```

Test if specific UART is enabled.

Parameters

- uart UART instance. uart0 or uart1

Returns

- true if the UART is enabled

3.23.5.6. uart_is_readable

```
static bool uart_is_readable (uart_inst_t *uart)
```

Determine whether data is waiting in the RX FIFO.

Parameters

- uart UART instance. uart0 or uart1

Returns

- 0 if no data available, otherwise the number of bytes, at least, that can be read

3.23.5.7. uart_is_writable

```
static bool uart_is_writable (uart_inst_t *uart)
```

Determine if space is available available in TX FIFO.

Parameters

- `uart` UART instance. uart0 or uart1

Returns

- false if no space available, true otherwise

3.23.5.8. uart_putc

```
static void uart_putc (uart_inst_t *uart,
       char c)
```

Write single character to UART for transmission, with optional CR/LF conversions.

This function will block until the character has been sent

Parameters

- `uart` UART instance. uart0 or uart1
- `c` The character to send

3.23.5.9. uart_putc_raw

```
static void uart_putc_raw (uart_inst_t *uart,
       char c)
```

Write single character to UART for transmission.

This function will block until all the character has been sent

Parameters

- `uart` UART instance. uart0 or uart1
- `c` The character to send

3.23.5.10. uart_puts

```
static void uart_puts (uart_inst_t *uart,
       const char *s)
```

Write string to UART for transmission, doing any CR/LF conversions.

This function will block until the entire string has been sent

Parameters

- `uart` UART instance. uart0 or uart1
- `s` The null terminated string to send

3.23.5.11. uart_read_blocking

```
static void uart_read_blocking (uart_inst_t *uart,
       uint8_t *dst,
       size_t len)
```

Read from the UART.

This function will block until all the data has been received from the UART

Parameters

- `uart` UART instance. uart0 or uart1
- `dst` Buffer to accept received bytes
- `len` The number of bytes to receive.

3.23.5.12. uart_set_baudrate

```
uint uart_set_baudrate (uart_inst_t *uart,
       uint baudrate)
```

Set UART baud rate.

Set baud rate as close as possible to requested, and return actual rate selected.

Parameters

- `uart` UART instance. uart0 or uart1
- `baudrate` Baudrate in Hz

3.23.5.13. uart_set_break

```
static void uart_set_break (uart_inst_t *uart,
       bool en)
```

Assert a break condition on the UART transmission.

Parameters

- `uart` UART instance. uart0 or uart1
- `en` Assert break condition (TX held low) if true. Clear break condition if false.

3.23.5.14. uart_set_crlf

```
void uart_set_crlf (uart_inst_t *uart,
       bool crlf)
```

Set CR/LF conversion on UART.

Parameters

- `uart` UART instance. uart0 or uart1
- `crlf` If true, convert line feeds to carriage return on transmissions

3.23.5.15. uart_set_fifo_enabled

```
static void uart_set_fifo_enabled (uart_inst_t *uart,
       bool enabled)
```

Enable/Disable the FIFOs on specified UART.

Parameters

- `uart` UART instance. uart0 or uart1
- `enabled` true to enable FIFO (default), false to disable

3.23.5.16. uart_set_format

```
static void uart_set_format (uart_inst_t *uart,
       uint data_bits,
       uint stop_bits,
       uart_parity_t parity)
```

Set UART data format.

Configure the data format (bits etc() for the UART

Parameters

- `uart` UART instance. uart0 or uart1
- `data_bits` Number of bits of data. 5..8
- `stop_bits` Number of stop bits 1..2
- `parity` Parity option.

3.23.5.17. uart_set_hw_flow

```
static void uart_set_hw_flow (uart_inst_t *uart,
       bool cts,
       bool rts)
```

Set UART flow control CTS/RTS.

Parameters

- `uart` UART instance. uart0 or uart1
- `cts` If true enable flow control of TX by clear-to-send input
- `rts` If true enable assertion of request-to-send output by RX flow control

3.23.5.18. uart_set_irq_enables

```
static void uart_set_irq_enables (uart_inst_t *uart,
       bool rx_has_data,
       bool tx_needs_data)
```

Setup UART interrupts.

Enable the UART's interrupt output. An interrupt handler will need to be installed prior to calling this function.

Parameters

- `uart` UART instance. uart0 or uart1
- `rx_has_data` If true an interrupt will be fired when the RX FIFO contain data.
- `tx_needs_data` If true an interrupt will be fired when the TX FIFO needs data.

3.23.5.19. uart_tx_wait_blocking

```
static void uart_tx_wait_blocking (uart_inst_t *uart)
```

Wait for the UART TX fifo to be drained.

Parameters

- `uart` UART instance. uart0 or uart1

3.23.5.20. uart_write_blocking

```
static void uart_write_blocking (uart_inst_t *uart,
       const uint8_t *src,
       size_t len)
```

Write to the UART for transmission.

This function will block until all the data has been sent to the UART

Parameters

- `uart` UART instance. uart0 or uart1
- `src` The bytes to send
- `len` The number of bytes to send

3.24. hardware_vreg

Voltage Regulation API. More...

3.24.1. Functions

- `void vreg_set_voltage (enum vreg_voltage voltage)`
  Set voltage. More...

3.24.2. Detailed Description

Voltage Regulation API.

3.24.3. Function Documentation

3.24.3.1. vreg_set_voltage

```
void vreg_set_voltage (enum vreg_voltage voltage)
```

Set voltage.

Parameters

- `voltage` The voltage (from enumeration vreg_voltage) to apply to the voltage regulator

3.25. hardware_watchdog

Hardware Watchdog Timer API. More...

3.25.1. Functions

- `void watchdog_reboot (uint32_t pc, uint32_t sp, uint32_t delay_ms)`
  Define actions to perform at watchdog timeout. More...
- `void watchdog_start_tick (uint cycles)`
  Start the watchdog tick. More...

- `void watchdog_update (void)`
  Reload the watchdog counter with the amount of time set in watchdog_enable.
- `void watchdog_enable (uint32_t delay_ms, bool pause_on_debug)`
  Enable the watchdog. More…
- `bool watchdog_caused_reboot (void)`
  Did the watchdog cause the last reboot? More…
- `uint32_t watchdog_get_count (void)`
  Returns the amount of microseconds before the watchdog will reboot the chip. More…

3.25.2. Detailed Description

Hardware Watchdog Timer API.

Supporting functions for the Pico hardware watchdog timer.

The RP2040 has a built in HW watchdog Timer. This is a countdown timer that can restart parts of the chip if it reaches zero. For example, this can be used to restart the processor if the software running on it gets stuck in an infinite loop or similar. The programmer has to periodically write a value to the watchdog to stop it reaching zero.

Example

```c
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/watchdog.h"

int main() {
    stdio_init_all();

    if (watchdog_caused_reboot()) {
        printf("Rebooted by Watchdog!\n");
        return 0;
    } else {
        printf("Clean boot\n");
    }

    // Enable the watchdog, requiring the watchdog to be updated every 100ms or the chip will reboot
    // second arg is pause on debug which means the watchdog will pause when stepping through code
    watchdog_enable(100, 1);

    for (uint i = 0; i < 5; i++) {
        printf("Updating watchdog %d\n", i);
        watchdog_update();
    }

    // Wait in an infinite loop and don't update the watchdog so it reboots us
    printf("Waiting to be rebooted by watchdog\n");
    while(1);
}
```

3.25.3. Function Documentation

3.25.3.1. watchdog_caused_reboot

```
bool watchdog_caused_reboot (void)
```

Did the watchdog cause the last reboot?

Returns

- true if the watchdog timer or a watchdog force caused the last reboot
- false there has been no watchdog reboot since run has been

3.25.3.2. watchdog_enable

```
void watchdog_enable (uint32_t delay_ms,
       bool pause_on_debug)
```

Enable the watchdog.

By default the SDK assumes a 12MHz XOSC and sets the `watchdog_start_tick` appropriately.

Parameters

- `delay_ms` Amount of milliseconds before watchdog will reboot without watchdog_update being called. Maximum of 0x7fffff, which is approximately 8.3 seconds
- `pause_on_debug` If the watchdog should be paused when the debugger is stepping through code

3.25.3.3. watchdog_get_count

```
uint32_t watchdog_get_count (void)
```

Returns the amount of microseconds before the watchdog will reboot the chip.

Returns

- The number of microseconds before the watchdog will reboot the chip.

3.25.3.4. watchdog_reboot

```
void watchdog_reboot (uint32_t pc,
       uint32_t sp,
       uint32_t delay_ms)
```

Define actions to perform at watchdog timeout.

By default the SDK assumes a 12MHz XOSC and sets the `watchdog_start_tick` appropriately.

Parameters

- `pc` If Zero, a standard boot will be performed, if non-zero this is the program counter to jump to on reset.
- `sp` If pc is non-zero, this will be the stack pointer used.
- `delay_ms` Initial load value. Maximum value 0x7fffff, approximately 8.3s.

3.25.3.5. watchdog_start_tick

```
void watchdog_start_tick (uint cycles)
```

Start the watchdog tick.

Parameters

- `cycles` This needs to be a divider that when applied to the XOSC input, produces a 1MHz clock. So if the XOSC is

12MHz, this will need to be 12.

3.25.3.6. watchdog_update

```
void watchdog_update (void)
```

Reload the watchdog counter with the amount of time set in watchdog_enable.

3.26. hardware_xosc

Crystal Oscillator (XOSC) API. More…

3.26.1. Functions

- `void xosc_init (void)`
  Initialise the crystal oscillator system. More…
- `void xosc_disable (void)`
  Disable the Crystal oscillator. More…
- `void xosc_dormant (void)`
  Set the crystal oscillator system to dormant. More…

3.26.2. Detailed Description

Crystal Oscillator (XOSC) API.

3.26.3. Function Documentation

3.26.3.1. xosc_disable

```
void xosc_disable (void)
```

Disable the Crystal oscillator.

Turns off the crystal oscillator source, and waits for it to become unstable

3.26.3.2. xosc_dormant

```
void xosc_dormant (void)
```

Set the crystal oscillator system to dormant.

Turns off the crystal oscillator until it is woken by an interrupt. Thi will block and hence the entire system will stop, until an interrupt wakes it up. This function will continue to block until the oscillator becomes stable after its wakeup.

3.26.3.3. xosc_init

```
void xosc_init (void)
```

Initialise the crystal oscillator system.

This function will block until the crystal oscillator has stabilised.

3.27. channel_config

DMA channel configuration. More…

3.27.1. Functions

- `static void channel_config_set_read_increment (dma_channel_config *c, bool incr)`
  Set DMA channel read increment. More…
- `static void channel_config_set_write_increment (dma_channel_config *c, bool incr)`
  Set DMA channel write increment. More…
- `static void channel_config_set_dreq (dma_channel_config *c, uint dreq)`
  Select a transfer request signal. More…
- `static void channel_config_set_chain_to (dma_channel_config *c, uint chain_to)`
  Set DMA channel completion channel. More…
- `static void channel_config_set_transfer_data_size (dma_channel_config *c, enum dma_channel_transfer_size size)`
  Set the size of each DMA bus transfer. More…
- `static void channel_config_set_ring (dma_channel_config *c, bool write, uint size_bits)`
  Set address wrapping parameters. More…
- `static void channel_config_set_bswap (dma_channel_config *c, bool bswap)`
  Set DMA byte swapping. More…
- `static void channel_config_set_irq_quiet (dma_channel_config *c, bool irq_quiet)`
  Set IRQ quiet mode. More…
- `static void channel_config_set_enable (dma_channel_config *c, bool enable)`
  Enable/Disable the DMA channel. More…
- `static void channel_config_set_sniff_enable (dma_channel_config *c, bool sniff_enable)`
  Enable access to channel by sniff hardware. More…
- `static dma_channel_config dma_channel_get_default_config (uint channel)`
  Get the default channel configuration for a given channel. More…
- `static dma_channel_config dma_get_channel_config (uint channel)`
  Get the current configuration for the specified channel. More…
- `static uint32_t channel_config_get_ctrl_value (const dma_channel_config *config)`
  Get the raw configuration register from a channel configuration. More…

3.27.2. Detailed Description

DMA channel configuration.

A DMA channel needs to be configured, these functions provide handy helpers to set up configuration structures. See dma_channel_config

3.27.3. Function Documentation

3.27.3.1. channel_config_get_ctrl_value

```
static uint32_t channel_config_get_ctrl_value (const dma_channel_config *config)
```

Get the raw configuration register from a channel configuration.

Parameters

- `config` Pointer to a config structure.

Returns

- Register content

3.27.3.2. channel_config_set_bswap

```
static void channel_config_set_bswap (dma_channel_config *c,
       bool bswap)
```

Set DMA byte swapping.

No effect for byte data, for halfword data, the two bytes of each halfword are swapped. For word data, the four bytes of each word are swapped to reverse their order.

Parameters

- `c` Pointer to channel configuration data
- `bswap` True to enable byte swapping

3.27.3.3. channel_config_set_chain_to

```
static void channel_config_set_chain_to (dma_channel_config *c,
       uint chain_to)
```

Set DMA channel completion channel.

When this channel completes, it will trigger the channel indicated by chain_to. Disable by setting chain_to to itself (the same channel)

Parameters

- `c` Pointer to channel configuration data
- `chain_to` Channel to trigger when this channel completes.

3.27.3.4. channel_config_set_dreq

```
static void channel_config_set_dreq (dma_channel_config *c,
       uint dreq)
```

Select a transfer request signal.

The channel uses the transfer request signal to pace its data transfer rate. Sources for TREQ signals are internal (TIMERS) or external (DREQ, a Data Request from the system). 0x0 to 0x3a → select DREQ n as TREQ 0x3b → Select Timer 0 as TREQ 0x3c → Select Timer 1 as TREQ 0x3d → Select Timer 2 as TREQ (Optional) 0x3e → Select Timer 3 as TREQ (Optional) 0x3f → Permanent request, for unpaced transfers.

Parameters

- `c` Pointer to channel configuration data
- `dreq` Source (see description)

3.27.3.5. channel_config_set_enable

```
static void channel_config_set_enable (dma_channel_config *c,
       bool enable)
```

Enable/Disable the DMA channel.

When false, the channel will ignore triggers, stop issuing transfers, and pause the current transfer sequence (i.e. BUSY will remain high if already high)

Parameters

- `c` Pointer to channel configuration data
- `enabled` True to enable the DMA channel. When enabled, the channel will respond to triggering events, and start transferring data.

3.27.3.6. channel_config_set_irq_quiet

```
static void channel_config_set_irq_quiet (dma_channel_config *c,
       bool irq_quiet)
```

Set IRQ quiet mode.

In QUIET mode, the channel does not generate IRQs at the end of every transfer block. Instead, an IRQ is raised when NULL is written to a trigger register, indicating the end of a control block chain.

Parameters

- `c` Pointer to channel configuration data
- `irq_quiet` True to enable quiet mode, false to disable.

3.27.3.7. channel_config_set_read_increment

```
static void channel_config_set_read_increment (dma_channel_config *c,
       bool incr)
```

Set DMA channel read increment.

Parameters

- `c` Pointer to channel configuration data
- `incr` True to enable read address increments, if false, each read will be from the same address Usually disabled for peripheral to memory transfers

3.27.3.8. channel_config_set_ring

```
static void channel_config_set_ring (dma_channel_config *c,
       bool write,
       uint size_bits)
```

Set address wrapping parameters.

Size of address wrap region. If 0, don't wrap. For values n > 0, only the lower n bits of the address will change. This wraps the address on a (1 << n) byte boundary, facilitating access to naturally-aligned ring buffers. Ring sizes between 2 and 32768 bytes are possible (size_bits from 1 - 15)

0x0 → No wrapping.

Parameters

- `c` Pointer to channel configuration data
- `write` True to apply to write addresses, false to apply to read addresses
- `size_bits` 0 to disable wrapping. Otherwise the size in bits of the changing part of the address. Effectively wraps the address on a (1 << size_bits) byte boundary.

3.27.3.9. channel_config_set_sniff_enable

```
static void channel_config_set_sniff_enable (dma_channel_config *c,
       bool sniff_enable)
```

Enable access to channel by sniff hardware.

Sniff HW must be enabled and have this channel selected.

Parameters

- `c` Pointer to channel configuration data
- `sniff_enabled` True to enable the Sniff HW access to this DMA channel.

3.27.3.10. channel_config_set_transfer_data_size

```
static void channel_config_set_transfer_data_size (dma_channel_config *c,
       enum dma_channel_transfer_size size)
```

Set the size of each DMA bus transfer.

Set the size of each bus transfer (byte/halfword/word). The read and write addresses advance by the specific amount (1/2/4 bytes) with each transfer.

Parameters

- `c` Pointer to channel configuration data
- `size` See enum for possible values.

3.27.3.11. channel_config_set_write_increment

```
static void channel_config_set_write_increment (dma_channel_config *c,
       bool incr)
```

Set DMA channel write increment.

Parameters

- `c` Pointer to channel configuration data
- `incr` True to enable write address increments, if false, each write will be to the same address Usually disabled for memory to peripheral transfers Usually disabled for memory to peripheral transfers

3.27.3.12. dma_channel_get_default_config

```
static dma_channel_config dma_channel_get_default_config (uint channel)
```

Get the default channel configuration for a given channel.

Setting	Default
Read Increment	true
Write Increment	false
DReq	DREQ_FORCE
Chain to	self
Data size	DMA_SIZE_32
Ring	write=false, size=0 (i.e. off)
Byte Swap	false

Setting	Default
Quiet IRQs	false
Channel Enable	true
Sniff Enable	false

Parameters

- `channel` DMA channel

Returns

- the default configuration which can then be modified.

3.27.3.13. dma_get_channel_config

`static dma_channel_config dma_get_channel_config (uint channel)`

Get the current configuration for the specified channel.

Parameters

- `channel` DMA channel

Returns

- The current configuration as read from the HW register (not cached)

3.28. interp_config

Interpolator configuration. More…

3.28.1. Functions

- `static void interp_config_shift (interp_config *c, uint shift)`
  Set the interpolator shift value. More…
- `static void interp_config_mask (interp_config *c, uint mask_lsb, uint mask_msb)`
  Set the interpolator mask range. More…
- `static void interp_config_cross_input (interp_config *c, bool cross_input)`
  Enable cross input. More…
- `static void interp_config_cross_result (interp_config *c, bool cross_result)`
  Enable cross results. More…
- `static void interp_config_signed (interp_config *c, bool _signed)`
  Set sign extension. More…
- `static void interp_config_add_raw (interp_config *c, bool add_raw)`
  Set raw add option. More…
- `static void interp_config_blend (interp_config *c, bool blend)`
  Set blend mode. More…
- `static void interp_config_clamp (interp_config *c, bool clamp)`
  Set interpolator clamp mode (Interpolator 1 only) More…
- `static void interp_config_force_bits (interp_config *c, uint bits)`
  Set interpolator Force bits. More…

- `static interp_config interp_default_config ()`
  Get a default configuration. More...
- `static void interp_set_config (interp_hw_t *interp, uint lane, interp_config *config)`
  Send configuration to a lane. More...

3.28.2. Detailed Description

Interpolator configuration.

Each interpolator needs to be configured, these functions provide handy helpers to set up configuration structures.

3.28.3. Function Documentation

3.28.3.1. interp_config_add_raw

```
static void interp_config_add_raw (interp_config *c,
       bool add_raw)
```

Set raw add option.

When enabled, mask + shift is bypassed for LANE0 result. This does not affect the FULL result.

Parameters

- `c` Pointer to interpolation config
- `add_raw` If true, enable raw add option.

3.28.3.2. interp_config_blend

```
static void interp_config_blend (interp_config *c,
       bool blend)
```

Set blend mode.

If enabled, LANE1 result is a linear interpolation between BASE0 and BASE1, controlled by the 8 LSBs of lane 1 shift and mask value (a fractional number between 0 and 255/256ths)

LANE0 result does not have BASE0 added (yields only the 8 LSBs of lane 1 shift+mask value)

FULL result does not have lane 1 shift+mask value added (BASE2 + lane 0 shift+mask)

LANE1 SIGNED flag controls whether the interpolation is signed or unsig

Parameters

- `c` Pointer to interpolation config
- `blend` Set true to enable blend mode.

3.28.3.3. interp_config_clamp

```
static void interp_config_clamp (interp_config *c,
       bool clamp)
```

Set interpolator clamp mode (Interpolator 1 only)

Only present on INTERP1 on each core. If CLAMP mode is enabled:

Parameters

- c Pointer to interpolation config
- clamp Set true to enable clamp mode

3.28.3.4. interp_config_cross_input

```
static void interp_config_cross_input (interp_config *c,
       bool cross_input)
```

Enable cross input.

Allows feeding of the accumulator content from the other lane back in to this lanes shift+mask hardware. This will take effect even if the interp_config_add_raw option is set as the cross input mux is before the shift+mask bypass

Parameters

- c Pointer to interpolation config
- cross_input If true, enable the cross input.

3.28.3.5. interp_config_cross_result

```
static void interp_config_cross_result (interp_config *c,
       bool cross_result)
```

Enable cross results.

Allows feeding of the other lane's result into this lane's accumulator on a POP operation.

Parameters

- c Pointer to interpolation config
- cross_result If true, enables the cross result

3.28.3.6. interp_config_force_bits

```
static void interp_config_force_bits (interp_config *c,
       uint bits)
```

Set interpolator Force bits.

ORed into bits 29:28 of the lane result presented to the processor on the bus.

No effect on the internal 32-bit datapath. Handy for using a lane to generate sequence of pointers into flash or SRAM

Parameters

- c Pointer to interpolation config
- bits Sets the force bits to that specified. Range 0-3 (two bits)

3.28.3.7. interp_config_mask

```
static void interp_config_mask (interp_config *c,
       uint mask_lsb,
       uint mask_msb)
```

Set the interpolator mask range.

Sets the range of bits (least to most) that are allowed to pass through the interpolator

Parameters

- c Pointer to interpolation config
- mask_lsb The least significant bit allowed to pass
- mask_msb The most significant bit allowed to pass

3.28.3.8. interp_config_shift

```
static void interp_config_shift (interp_config *c,
       uint shift)
```

Set the interpolator shift value.

Sets the number of bits the accumulator is shifted before masking, on each iteration.

Parameters

- c Pointer to an interpolator config
- shift Number of bits

3.28.3.9. interp_config_signed

```
static void interp_config_signed (interp_config *c,
       bool _signed)
```

Set sign extension.

Enables signed mode, where the shifted and masked accumulator value is sign-extended to 32 bits before adding to BASE1, and LANE1 PEEK/POP results appear extended to 32 bits when read by processor.

Parameters

- c Pointer to interpolation config
- _signed If true, enables sign extension

3.28.3.10. interp_default_config

```
static interp_config interp_default_config ()
```

Get a default configuration.

Returns

- A default interpolation configuration

3.28.3.11. interp_set_config

```
static void interp_set_config (interp_hw_t *interp,
       uint lane,
       interp_config *config)
```

Send configuration to a lane.

If an invalid configuration is specified (ie a lane specific item is set on wrong lane), depending on setup this function can panic.

Parameters

- interp Interpolator instance, interp0 or interp1.
- lane The lane to set

- config Pointer to interpolation config

3.29. sm_config

PIO state machine configuration. More...

3.29.1. Data Structures

- struct **pio_sm_config**
  PIO Configuration structure.

3.29.2. Functions

- static void **sm_config_set_out_pins** (pio_sm_config *c, uint out_base, uint out_count)
  Set the 'out' pins in a state machine configuration. More...
- static void **sm_config_set_set_pins** (pio_sm_config *c, uint set_base, uint set_count)
  Set the 'set' pins in a state machine configuration. More...
- static void **sm_config_set_in_pins** (pio_sm_config *c, uint in_base)
  Set the 'in' pins in a state machine configuration. More...
- static void **sm_config_set_sideset_pins** (pio_sm_config *c, uint sideset_base)
  Set the 'sideset' pins in a state machine configuration. More...
- static void **sm_config_set_sideset** (pio_sm_config *c, uint bit_count, bool optional, bool pindirs)
  Set the 'sideset' options in a state machine configuration. More...
- static void **sm_config_set_clkdiv** (pio_sm_config *c, float div)
  Set the state machine clock divider (from a floating point value) in a state machine configuration. More...
- static void **sm_config_set_clkdiv_int_frac** (pio_sm_config *c, uint16_t div_int, uint8_t div_frac)
  Set the state machine clock divider (from integer and fractional parts - 16:8) in a state machine configuration. More...
- static void **sm_config_set_wrap** (pio_sm_config *c, uint wrap_target, uint wrap)
  Set the wrap addresses in a state machine configuration. More...
- static void **sm_config_set_jmp_pin** (pio_sm_config *c, uint pin)
  Set the 'jmp' pin in a state machine configuration. More...
- static void **sm_config_set_in_shift** (pio_sm_config *c, bool shift_right, bool autopush, uint push_threshold)
  Setup 'in' shifting parameters in a state machine configuration. More...
- static void **sm_config_set_out_shift** (pio_sm_config *c, bool shift_right, bool autopull, uint pull_threshold)
  Setup 'out' shifting parameters in a state machine configuration. More...
- static void **sm_config_set_fifo_join** (pio_sm_config *c, enum pio_fifo_join join)
  Setup the FIFO joining in a state machine configuration. More...
- static void **sm_config_set_out_special** (pio_sm_config *c, bool sticky, bool has_enable_pin, int enable_pin_index)
  Set special 'out' operations in a state machine configuration. More...
- static void **sm_config_set_mov_status** (pio_sm_config *c, enum pio_mov_status_type status_sel, uint status_n)
  Set source for 'mov status' in a state machine configuration. More...
- static pio_sm_config **pio_get_default_sm_config** ()
  Get the default state machine configuration. More...
- static void **pio_sm_set_out_pins** (PIO pio, uint sm, uint out_base, uint out_count)
  Set the current 'out' pins for a state machine. More...

- static void pio_sm_set_set_pins (PIO pio, uint sm, uint set_base, uint set_count)
  Set the current 'set' pins for a state machine. More...
- static void pio_sm_set_in_pins (PIO pio, uint sm, uint in_base)
  Set the current 'in' pins for a state machine. More...
- static void pio_sm_set_sideset_pins (PIO pio, uint sm, uint sideset_base)
  Set the current 'sideset' pins for a state machine. More...

3.29.3. Detailed Description

PIO state machine configuration.

A PIO block needs to be configured, these functions provide helpers to set up configuration structures. See pio_sm_set_config

3.29.4. Function Documentation

3.29.4.1. pio_get_default_sm_config

```
static pio_sm_config pio_get_default_sm_config ()
```

Get the default state machine configuration.

Setting	Default
Out Pins	32 starting at 0
Set Pins	0 starting at 0
In Pins (base)	0
Side Set Pins (base)	0
Side Set	disabled
Wrap	wrap=31, wrap_to=0
In Shift	shift_direction=right, autopush=false, push_thrshold=32
Out Shift	shift_direction=right, autopull=false, pull_thrshold=32
Jmp Pin	0
Out Special	sticky=false, has_enable_pin=false, enable_pin_index=0
Mov Status	status_sel=STATUS_TX_LESSTHAN, n=0

Returns

- the default state machine configuration which can then be modified.

3.29.4.2. pio_sm_set_in_pins

```
static void pio_sm_set_in_pins (PIO pio,
        uint sm,
        uint in_base)
```

Set the current 'in' pins for a state machine.

Can overlap with the 'out', ''set' and 'sideset' pins

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)
- `in_base` 0-31 First pin to set as input

3.29.4.3. pio_sm_set_out_pins

```
static void pio_sm_set_out_pins (PIO pio,
       uint sm,
       uint out_base,
       uint out_count)
```

Set the current 'out' pins for a state machine.

Can overlap with the 'in', 'set' and 'sideset' pins

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)
- `out_base` 0-31 First pin to set as output
- `out_count` 0-32 Number of pins to set.

3.29.4.4. pio_sm_set_set_pins

```
static void pio_sm_set_set_pins (PIO pio,
       uint sm,
       uint set_base,
       uint set_count)
```

Set the current 'set' pins for a state machine.

Can overlap with the 'in', 'out' and 'sideset' pins

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)
- `set_base` 0-31 First pin to set as
- `set_count` 0-5 Number of pins to set.

3.29.4.5. pio_sm_set_sideset_pins

```
static void pio_sm_set_sideset_pins (PIO pio,
       uint sm,
       uint sideset_base)
```

Set the current 'sideset' pins for a state machine.

Can overlap with the 'in', 'out' and 'set' pins

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)
- `sideset_base` base pin for 'side set'

3.29.4.6. sm_config_set_clkdiv

```
static void sm_config_set_clkdiv (pio_sm_config *c,
       float div)
```

Set the state machine clock divider (from a floating point value) in a state machine configuration.

The clock divider acts on the system clock to provide a clock for the state machine. See the databook for more details.

Parameters

- `c` Pointer to the configuration structure to modify
- `div` The fractional divisor to be set. 1 for full speed. An integer clock divisor of n will cause the state machine to run 1 cycle in every n. Note that for small n, the jitter introduced by a fractional divider (e.g. 2.5) may be unacceptable although it will depend on the use case.

3.29.4.7. sm_config_set_clkdiv_int_frac

```
static void sm_config_set_clkdiv_int_frac (pio_sm_config *c,
       uint16_t div_int,
       uint8_t div_frac)
```

Set the state machine clock divider (from integer and fractional parts - 16:8) in a state machine configuration.

The clock divider acts on the system clock to provide a clock for the state machine. See the databook for more details.

Parameters

- `c` Pointer to the configuration structure to modify
- `div_int` Integer part of the divisor
- `div_frac` Fractional part in 1/256ths

See also

- sm_config_set_clkdiv

3.29.4.8. sm_config_set_fifo_join

```
static void sm_config_set_fifo_join (pio_sm_config *c,
       enum pio_fifo_join join)
```

Setup the FIFO joining in a state machine configuration.

Parameters

- `c` Pointer to the configuration structure to modify
- `join` Specifies the join type.

See also

- enum pio_fifo_join

3.29.4.9. sm_config_set_in_pins

```
static void sm_config_set_in_pins (pio_sm_config *c,
       uint in_base)
```

Set the 'in' pins in a state machine configuration.

Can overlap with the 'out', ''set' and 'sideset' pins

Parameters

- `c` Pointer to the configuration structure to modify
- `in_base` 0-31 First pin to set as input

3.29.4.10. sm_config_set_in_shift

```
static void sm_config_set_in_shift (pio_sm_config *c,
       bool shift_right,
       bool autopush,
       uint push_threshold)
```

Setup 'in' shifting parameters in a state machine configuration.

Parameters

- `c` Pointer to the configuration structure to modify
- `shift_right` true to shift ISR to right, false to shift ISR to left
- `autopush` whether autopush is enabled
- `push_threshold` threshold in bits to shift in before auto/conditional re-pushing of the ISR

3.29.4.11. sm_config_set_jmp_pin

```
static void sm_config_set_jmp_pin (pio_sm_config *c,
       uint pin)
```

Set the 'jmp' pin in a state machine configuration.

Parameters

- `c` Pointer to the configuration structure to modify
- `pin` The raw GPIO pin number to use as the source for a jmp pin instruction

3.29.4.12. sm_config_set_mov_status

```
static void sm_config_set_mov_status (pio_sm_config *c,
       enum pio_mov_status_type status_sel,
       uint status_n)
```

Set source for 'mov status' in a state machine configuration.

Parameters

- `c` Pointer to the configuration structure to modify
- `status_sel` the status operation selector
- `status_n` parameter for the mov status operation (currently a bit count)

3.29.4.13. sm_config_set_out_pins

```
static void sm_config_set_out_pins (pio_sm_config *c,
       uint out_base,
       uint out_count)
```

Set the 'out' pins in a state machine configuration.

Can overlap with the 'in', 'set' and 'sideset' pins

Parameters

- c Pointer to the configuration structure to modify
- out_base 0-31 First pin to set as output
- out_count 0-32 Number of pins to set.

3.29.4.14. sm_config_set_out_shift

```
static void sm_config_set_out_shift (pio_sm_config *c,
       bool shift_right,
       bool autopull,
       uint pull_threshold)
```

Setup 'out' shifting parameters in a state machine configuration.

Parameters

- c Pointer to the configuration structure to modify
- shift_right true to shift OSR to right, false to shift OSR to left
- autopull whether autopull is enabled
- pull_threshold threshold in bits to shift out before auto/conditional re-pulling of the OSR

3.29.4.15. sm_config_set_out_special

```
static void sm_config_set_out_special (pio_sm_config *c,
       bool sticky,
       bool has_enable_pin,
       int enable_pin_index)
```

Set special 'out' operations in a state machine configuration.

Parameters

- c Pointer to the configuration structure to modify
- sticky to enable 'sticky' output (i.e. re-asserting most recent OUT/SET pin values on subsequent cycles)
- has_enable_pin true to enable auxiliary OUT enable pin
- enable_pin_index pin index for auxiliary OUT enable

3.29.4.16. sm_config_set_set_pins

```
static void sm_config_set_set_pins (pio_sm_config *c,
       uint set_base,
       uint set_count)
```

Set the 'set' pins in a state machine configuration.

Can overlap with the 'in', 'out' and 'sideset' pins

Parameters

- c Pointer to the configuration structure to modify
- set_base 0-31 First pin to set as
- set_count 0-5 Number of pins to set.

3.29.4.17. sm_config_set_sideset

```
static void sm_config_set_sideset (pio_sm_config *c,
       uint bit_count,
       bool optional,
       bool pindirs)
```

Set the 'sideset' options in a state machine configuration.

Parameters

- c Pointer to the configuration structure to modify
- bit count Number of bits to steal from delay field in the instruction for use of side set
- optional True if the topmost side set bit is used as a flag for whether to apply side set on that instruction
- pindirs True if the side set affects pin directions rather than values

3.29.4.18. sm_config_set_sideset_pins

```
static void sm_config_set_sideset_pins (pio_sm_config *c,
       uint sideset_base)
```

Set the 'sideset' pins in a state machine configuration.

Can overlap with the 'in', 'out' and 'set' pins

Parameters

- c Pointer to the configuration structure to modify
- sideset_base base pin for 'side set'

3.29.4.19. sm_config_set_wrap

```
static void sm_config_set_wrap (pio_sm_config *c,
       uint wrap_target,
       uint wrap)
```

Set the wrap addresses in a state machine configuration.

Parameters

- c Pointer to the configuration structure to modify
- wrap_target the instruction memory address to wrap to
- wrap the instruction memory address after which to set the program counter to wrap_target if the instruction does not itself update the program_counter

3.30. High Level APIs

3.30.1. Modules

- pico_audio
- pico_audio_i2s
- pico_audio_pwm
- pico_multicore
  Adds support for running code on the second processor core (core1)

- `pico_stdlib`
  Aggregation of a core subset of Pico SDK libraries used by most executables along with some additional utility methods. Including pico_stdlib gives you everything you need to get a basic program running which prints to stdout or flashes a LED.
- `pico_sync`
  Synchronization primitives and mutual exclusion.
- `pico_scanvideo`
- `pico_scanvideo_dpi`
- `pico_time`
  API for accurate timestamps, sleeping, and time based callbacks.
- `pico_util`

3.31. pico_audio

3.32. pico_audio_i2s

3.33. pico_audio_pwm

3.34. pico_multicore

Adds support for running code on the second processor core (core1) More…

3.34.1. Modules

- `fifo`
  Functions for inter-core FIFO.

3.34.2. Functions

- `void multicore_reset_core1 ()`
  Reset Core 1.
- `void multicore_launch_core1 (void(*entry)(void))`
  Run code on core 1. More…
- `void multicore_launch_core1_with_stack (void(*entry)(void), uint32_t *stack_bottom, size_t stack_size_bytes)`
  Launch code on core 1 with stack. More…
- `void multicore_sleep_core1 ()`
  Send core 1 to sleep.
- `void multicore_launch_core1_raw (void(*entry)(void), uint32_t *sp, uint32_t vector_table)`
  Launch code on core 1 with no stack protection. More…

3.34.3. Detailed Description

Adds support for running code on the second processor core (core1)

Example

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "pico/multicore.h"

#define FLAG_VALUE 123

void core1_entry() {

    multicore_fifo_push_blocking(FLAG_VALUE);

    uint32_t g = multicore_fifo_pop_blocking();

    if (g != FLAG_VALUE)
        printf("Hmm, that's not right on core 1!\n");
    else
        printf("Its all gone well on core 1!");

    while (1)
        tight_loop_contents();
}

int main() {
    stdio_init_all();
    printf("Hello, multicore!\n");


    multicore_launch_core1(core1_entry);

    // Wait for it to start up

    uint32_t g = multicore_fifo_pop_blocking();

    if (g != FLAG_VALUE)
        printf("Hmm, that's not right on core 0!\n");
    else {
        multicore_fifo_push_blocking(FLAG_VALUE);
        printf("It's all gone well on core 0!");
    }

}
```

3.34.4. Function Documentation

3.34.4.1. multicore_launch_core1

`void multicore_launch_core1 (void(*entry)(void))`

Run code on core 1.

Reset core1 and enter the given function on core 1 using the default core 1 stack (below core 0 stack)

Parameters

- `entry` Function entry point, this function should not return.

3.34.4.2. multicore_launch_core1_raw

```
void multicore_launch_core1_raw (void(*entry)(void),
       uint32_t *sp,
       uint32_t vector_table)
```

Launch code on core 1 with no stack protection.

Reset core1 and enter the given function using the passed sp as the initial stack pointer. This is a bare bones functions that does not provide a stack guard even if USE_STACK_GUARDS is defined

3.34.4.3. multicore_launch_core1_with_stack

```
void multicore_launch_core1_with_stack (void(*entry)(void),
       uint32_t *stack_bottom,
       size_t stack_size_bytes)
```

Launch code on core 1 with stack.

Reset core1 and enter the given function on core 1 using the passed stack for core 1

3.34.4.4. multicore_reset_core1

```
void multicore_reset_core1 ()
```

Reset Core 1.

3.34.4.5. multicore_sleep_core1

```
void multicore_sleep_core1 ()
```

Send core 1 to sleep.

3.35. pico_stdlib

Aggregation of a core subset of Pico SDK libraries used by most executables along with some additional utility methods. Including pico_stdlib gives you everything you need to get a basic program running which prints to stdout or flashes a LED. More…

3.35.1. Functions

- `void setup_default_uart ()`
  Set up the default UART and assign it to the default GPIO's. More…
- `void set_sys_clock_48mhz ()`
  Initialise the system clock to 48MHz. More…
- `void set_sys_clock_pll (uint32_t vco_freq, uint post_div1, uint post_div2)`
  Initialise the system clock. More…
- `bool check_sys_clock_khz (uint32_t freq_khz, uint *vco_freq_out, uint *post_div1_out, uint *post_div2_out)`
  Check if a given system clock frequency is valid/attainable. More…
- `static bool set_sys_clock_khz (uint32_t freq_khz, bool required)`
  Attempt to set a system clock frequency in khz. More…

3.35.2. Detailed Description

Aggregation of a core subset of Pico SDK libraries used by most executables along with some additional utility methods. Including pico_stdlib gives you everything you need to get a basic program running which prints to stdout or flashes a LED.

This library aggregates:

- hardware_uart
- hardware_gpio
- pico_binary_info
- pico_runtime
- pico_platform
- pico_printf
- pico_stdio
- pico_standard_link
- pico_util

There are some basic default values used by these functions that will default to usable values, however, they can be customised in a board definition header via config.h or similar

3.35.3. Function Documentation

3.35.3.1. check_sys_clock_khz

```
bool check_sys_clock_khz (uint32_t freq_khz,
       uint *vco_freq_out,
       uint *post_div1_out,
       uint *post_div2_out)
```

Check if a given system clock frequency is valid/attainable.

Parameters

- `freq_khz` Requested frequency
- `vco_freq_put` On success, the voltage controller oscillator frequeucny to be used by the SYS PLL
- `post_div1_out` On success, The first post divider for the SYS PLL
- `post_div2_out` On success, The second post divider for the SYS PLL.

Returns

- true if the frequency is possible and the output parameters have been written.

3.35.3.2. set_sys_clock_48mhz

```
void set_sys_clock_48mhz ()
```

Initialise the system clock to 48MHz.

Set the system clock to 48MHz, and set the peripheral clock to match.

3.35.3.3. set_sys_clock_khz

```
static bool set_sys_clock_khz (uint32_t freq_khz,
       bool required)
```

Attempt to set a system clock frequency in khz.

Note that not all clock frequencies are possible; it is preferred that you use src/rp2_common/hardware_clocks/scripts/vcocalc.py to calculate the parameters for use with sey_sys_clock_pll

Parameters

- `freq_khz` Requested frequency
- `requiredif` true then this function will assert if the frequency is not attainable.

Returns

- true if the clock was configured

3.35.3.4. set_sys_clock_pll

```
void set_sys_clock_pll (uint32_t vco_freq,
       uint post_div1,
       uint post_div2)
```

Initialise the system clock.

See the PLL documentation in the datasheet for details of driving the PLLs.

Parameters

- `vco_freq` The voltage controller oscillator frequeucny to be used by the SYS PLL
- `post_div1` The first post divider for the SYS PLL
- `post_div2` The second post divider for the SYS PLL.

3.35.3.5. setup_default_uart

```
void setup_default_uart ()
```

Set up the default UART and assign it to the default GPIO's.

By default this will use UART 0, with TX to pin GPIO 0, RX to pin GPIO 1, and the baudrate to 115200

Calling this method also intializes stdin/stdout over UART if the pico_stdio_uart library is linked.

Defaults can be changed using configuration defines, PICO_DEFAULT_UART_INSTANCE, PICO_DEFAULT_UART_BAUD_RATE PICO_DEFAULT_UART_TX_PIN PICO_DEFAULT_UART_RX_PIN

3.36. pico_sync

Synchronization primitives and mutual exclusion. More…

3.36.1. Modules

- **`critical_section`**
  Critical Section API for short-lived mutual exclusion safe for IRQ and multi-core.
- **`mutex`**
  Mutex API for non IRQ mutual exclusion between cores.

- `sem`
  Semaphore API for restricting access to a source.

3.36.2. Files

- file `lock_core.h`

3.36.3. Detailed Description

Synchronization primitives and mutual exclusion.

3.37. pico_scanvideo

3.38. pico_scanvideo_dpi

3.39. pico_time

API for accurate timestamps, sleeping, and time based callbacks. More…

3.39.1. Modules

- `timestamp`
  Timestamp functions relating to points in time (including the current time)
- `sleep`
  Sleep functions for delaying execution in a lower power state.
- `alarm`
  Alarm functions for scheduling future execution.
- `repeating_timer`
  Repeating Timer functions for simple scheduling of repeated execution.

3.39.2. Detailed Description

API for accurate timestamps, sleeping, and time based callbacks.

The functions defined here provide a much more powerful and user friendly wrapping around the low level hardware timer functionality. For these functions (and any other Pico SDK functionality e.g. timeouts, that relies on them) to work correctly, the hardware timer should not be modified. i.e. it is expected to be monotonically increasing once per microsecond. Fortunately there is no need to modify the hardware timer as any functionality you can think of that isn't already covered here can easily be modeled by adding or subtracting a constant value from the unmodified hardware timer.

- hardware_timer

3.40. pico_util

3.40.1. Modules

- `datetime`
  Date/Time formatting.
- `pheap`
  Pairing Heap Implementation.
- `queue`
  Multi-core and IRQ safe queue implementation.

3.41. fifo

Functions for inter-core FIFO. More…

3.41.1. Functions

- `static bool multicore_fifo_rvalid ()`
  Check the read FIFO to see if there is data waiting. More…
- `static bool multicore_fifo_wready ()`
  Check the FIFO to see if the write FIFO is full. More…
- `void multicore_fifo_push_blocking (uint32_t data)`
  Push data on to the FIFO. More…
- `uint32_t multicore_fifo_pop_blocking ()`
  Pop data from the FIFO. More…
- `static void multicore_fifo_drain ()`
  Flush any data in the outgoing FIFO.
- `static void multicore_fifo_clear_irq ()`
  Clear FIFO interrupt.
- `static int32_t multicore_fifo_get_status ()`
  Get FIFO status. More…

3.41.2. Detailed Description

Functions for inter-core FIFO.

The RP2040 contains two FIFOs for passing data, messages or ordered events between the two cores. Each FIFO is 32 bits wide, and 8 entries deep. One of the FIFOs can only be written by core 0, and read by core 1. The other can only be written by core 1, and read by core 0.

3.41.3. Function Documentation

3.41.3.1. multicore_fifo_clear_irq

```
static void multicore_fifo_clear_irq ()
```

Clear FIFO interrupt.

3.41.3.2. multicore_fifo_drain

```
static void multicore_fifo_drain ()
```

Flush any data in the outgoing FIFO.

3.41.3.3. multicore_fifo_get_status

```
static int32_t multicore_fifo_get_status ()
```

Get FIFO status.

Bit	Description
3	Sticky flag indicating the RX FIFO was read when empty. This read was ignored by the FIFO.
2	Sticky flag indicating the TX FIFO was written when full. This write was ignored by the FIFO.
1	Value is 1 if this core's TX FIFO is not full (i.e. if FIFO_WR is ready for more data)
0	Value is 1 if this core's RX FIFO is not empty (i.e. if FIFO_RD is valid)

Returns

- The status as a bitfield

3.41.3.4. multicore_fifo_pop_blocking

```
uint32_t multicore_fifo_pop_blocking ()
```

Pop data from the FIFO.

This function will block until there is data ready to be read Use `multicore_fifo_rvalid()` to check if data is ready to be read if you don't want to block.

Returns

- 32 bit unsigned data from the FIFO.

3.41.3.5. multicore_fifo_push_blocking

```
void multicore_fifo_push_blocking (uint32_t data)
```

Push data on to the FIFO.

This function will block until there is space for the data to be sent. Use `multicore_fifo_wready()` to check if it is possible to write to the FIFO if you don't want to block.

Parameters

- `data` A 32 bit value to push on to the FIFO

3.41.3.6. multicore_fifo_rvalid

```
static bool multicore_fifo_rvalid ()
```

Check the read FIFO to see if there is data waiting.

Returns

- true if the FIFO has data in it, false otherwise

3.41.3.7. multicore_fifo_wready

`static bool multicore_fifo_wready ()`

Check the FIFO to see if the write FIFO is full.

Returns

- true if the FIFO is full, false otherwise

3.42. critical_section

Critical Section API for short-lived mutual exclusion safe for IRQ and multi-core. More…

3.42.1. Functions

- `void critical_section_init (critical_section_t *critsec)`
  Initialise a critical_section structure allowing the system to assign a spin lock number. More…
- `void critical_section_init_with_lock_num (critical_section_t *critsec, uint lock_num)`
  Initialise a critical_section structure assigning a specific spin lock number. More…
- `static void critical_section_enter_blocking (critical_section_t *critsec)`
  Enter a critical_section. More…
- `static void critical_section_exit (critical_section_t *critsec)`
  Release a critical_section. More…

3.42.2. Detailed Description

Critical Section API for short-lived mutual exclusion safe for IRQ and multi-core.

A critical section is non-reentrant, and provides mutual exclusion using a spin-lock to prevent access from the other core, and from (higher priority) interrupts on the same core. It does the former using a spin lock and the latter by disabling interrupts on the calling core.

Because interrupts are disabled by this function, uses of the critical_section should be as short as possible.

3.42.3. Function Documentation

3.42.3.1. critical_section_enter_blocking

`static void critical_section_enter_blocking (critical_section_t *critsec)`

Enter a critical_section.

If the spin lock associated with this critical section is in use, then this method will block until it is released.

Parameters

- `critsec` Pointer to critical_section structure

3.42.3.2. critical_section_exit

```
static void critical_section_exit (critical_section_t *critsec)
```

Release a critical_section.

Parameters

- `critsec` Pointer to critical_section structure

3.42.3.3. critical_section_init

```
void critical_section_init (critical_section_t *critsec)
```

Initialise a critical_section structure allowing the system to assign a spin lock number.

The critical section is initialized ready for use, and will use a (possibly shared) spin lock number assigned by the system. Note that in general it is unlikely that you would be nesting crtical sections, however if you do so you use `critical_section_init_with_lock_num` to ensure that the spin lock's used are different.

Parameters

- `critsec` Pointer to critical_section structure

3.42.3.4. critical_section_init_with_lock_num

```
void critical_section_init_with_lock_num (critical_section_t *critsec,
       uint lock_num)
```

Initialise a critical_section structure assigning a specific spin lock number.

Parameters

- `critsec` Pointer to critical_section structure

3.43. mutex

Mutex API for non IRQ mutual exclusion between cores. More...

3.43.1. Macros

- `#define auto_init_mutex(name) static __attribute__((section(".mutex_array"))) mutex_t name`
  Helper macro for static definition of mutexes. More...

3.43.2. Functions

- `void mutex_init (mutex_t *mtx)`
  Initialise a mutex structure. More...
- `void mutex_enter_blocking (mutex_t *mtx)`
  Take ownership of a mutex. More...
- `bool mutex_try_enter (mutex_t *mtx, uint32_t *owner_out)`
  Check to see if a mutex is available. More...
- `bool mutex_enter_timeout_ms (mutex_t *mtx, uint32_t timeout_ms)`
  Wait for mutex with timeout. More...
- `bool mutex_enter_block_until (mutex_t *mtx, absolute_time_t until)`
  Wait for mutex until a specific time. More...

- `void mutex_exit (mutex_t *mtx)`
  Release ownership of a mutex. More...
- `static bool mutex_is_initialzed (mutex_t *mtx)`
  Test for mutex initialised state. More...

3.43.3. Detailed Description

Mutex API for non IRQ mutual exclusion between cores.

Mutexes are application level locks usually used protecting data structures that might be used by multiple cores. Unlike critical sections, the mutex protected code is not necessarily required/expected to complete quickl, as no other sytemwide locks are held on account of a locked mutex.

Because they are not re-entrant on the same core, blocking on a mutex should never be done in an IRQ handler. It is valid to call mutex_try_enter from within an IRQ handler, if the operation that would be conducted under lock can be skipped if the mutex is locked (at least by the same core).

See critical_section.h for protecting access between multiple cores AND IRQ handlers

3.43.4. Function Documentation

3.43.4.1. mutex_enter_block_until

```
bool mutex_enter_block_until (mutex_t *mtx,
       absolute_time_t until)
```

Wait for mutex until a specific time.

Wait until the specific time to take ownership of the mutex. If the calling core can take ownership of the mutex before the timeout expires, then true will be returned and the calling core will own the mutex, otherwise false will be returned and the calling core will own the mutex.

Parameters

- `mtx` Pointer to mutex structure
- `until` The time after which to return if the core cannot take owner ship of the mutex

Returns

- true if mutex now owned, false if timeout occurred before mutex became available

3.43.4.2. mutex_enter_blocking

```
void mutex_enter_blocking (mutex_t *mtx)
```

Take ownership of a mutex.

This function will block until the calling core can claim ownership of the mutex. On return the caller core owns the mutex

Parameters

- `mtx` Pointer to mutex structure

3.43.4.3. mutex_enter_timeout_ms

```
bool mutex_enter_timeout_ms (mutex_t *mtx,
       uint32_t timeout_ms)
```

Wait for mutex with timeout.

Wait for up to the specific time to take ownership of the mutex. If the calling core can take ownership of the mutex before the timeout expires, then true will be returned and the calling core will own the mutex, otherwise false will be returned and the calling core will own the mutex.

Parameters

- `mtx` Pointer to mutex structure
- `timeout_ms` The timeout in milliseconds.

Returns

- true if mutex now owned, false if timeout occurred before mutex became available

3.43.4.4. mutex_exit

```
void mutex_exit (mutex_t *mtx)
```

Release ownership of a mutex.

Parameters

- `mtx` Pointer to mutex structure

3.43.4.5. mutex_init

```
void mutex_init (mutex_t *mtx)
```

Initialise a mutex structure.

Parameters

- `mtx` Pointer to mutex structure

3.43.4.6. mutex_is_initialzed

```
static bool mutex_is_initialzed (mutex_t *mtx)
```

Test for mutex initialised state.

Parameters

- `mtx` Pointer to mutex structure

Returns

- true if the mutex is initialised, false otherwise

3.43.4.7. mutex_try_enter

```
bool mutex_try_enter (mutex_t *mtx,
       uint32_t *owner_out)
```

Check to see if a mutex is available.

Will return true if the mutex is unowned, false otherwise

Parameters

- `mtx` Pointer to mutex structure
- `owner_out` If mutex is owned, and this pointer is non-zero, it will be filled in with the core number of the current owner of the mutex

3.44. sem

Semaphore API for restricting access to a source. More…

3.44.1. Functions

- `void sem_init (semaphore_t *sem, int16_t initial_permits, int16_t max_permits)`
  Initialise a semaphore structure. More…
- `int sem_available (semaphore_t *sem)`
  Return number of available permits on the semaphore. More…
- `bool sem_release (semaphore_t *sem)`
  Release a permit on a semaphore. More…
- `void sem_reset (semaphore_t *sem, int16_t permits)`
  Reset semaphore to a specific number of available permits. More…
- `void sem_acquire_blocking (semaphore_t *sem)`
  Acquire a permit from the semaphore. More…
- `bool sem_acquire_timeout_ms (semaphore_t *sem, uint32_t timeout_ms)`
  Acquire a permit from a semaphore, with timeout. More…

3.44.2. Detailed Description

Semaphore API for restricting access to a source.

A semaphore holds a number of available permits. sem_acquire methods will acquire a permit if available (reducing the available count by 1) or block if the number of available permits is 0. sem_release() increases the number of available permits by one potentially unblocking a sem_acquire method.

Note that sem_release() may be called an arbitrary number of times, however the number of available permits is capped to the max_permit value specified during semaphore initialization.

Although these semaphore related functions can be used from IRQ handlers, it is obviously preferable to only release semaphores from within an IRQ handler (i.e. avoid blocking)

3.44.3. Function Documentation

3.44.3.1. sem_acquire_blocking

```
void sem_acquire_blocking (semaphore_t *sem)
```

Acquire a permit from the semaphore.

This function will block and wait if no permits are available.

Parameters

- `sem` Pointer to semaphore structure

3.44.3.2. sem_acquire_timeout_ms

```
bool sem_acquire_timeout_ms (semaphore_t *sem,
       uint32_t timeout_ms)
```

Acquire a permit from a semaphore, with timeout.

This function will block and wait if no permits are available, until the defined timeout has been reached. If the timeout is reached the function will return false, otherwise it will return true.

Parameters

- `sem` Pointer to semaphore structure
- `timeout_ms` Time to wait to aquire the semaphore, in ms.

Returns

- false if timeout reached, true if permit was acquired.

3.44.3.3. sem_available

```
int sem_available (semaphore_t *sem)
```

Return number of available permits on the semaphore.

Parameters

- `sem` Pointer to semaphore structure

Returns

- The number of permits available on the semaphore.

3.44.3.4. sem_init

```
void sem_init (semaphore_t *sem,
       int16_t initial_permits,
       int16_t max_permits)
```

Initialise a semaphore structure.

Parameters

- `sem` Pointer to semaphore structure
- `initial_permits` How many permits are initially acquired
- `max_permits` Total number of permits allowed for this semaphore

3.44.3.5. sem_release

```
bool sem_release (semaphore_t *sem)
```

Release a permit on a semaphore.

Increases the number of permits by one (unless the number of permits is already at the maximum). A blocked `sem_acquire` will be released if the number of permits is increased.

Parameters

- `sem` Pointer to semaphore structure

Returns

- true if the number of permits avialable was increased.

3.44.3.6. sem_reset

```
void sem_reset (semaphore_t *sem,
       int16_t permits)
```

Reset semaphore to a specific number of available permits.

Reset value should be from 0 to the max_permits specified in the init function

Parameters

- `sem` Pointer to semaphore structure
- `permits` the new number of available permits

3.45. timestamp

Timestamp functions relating to points in time (including the current time) More...

3.45.1. Functions

- `static absolute_time_t get_absolute_time ()`
  Return a representation of the current time. More...
- `static uint32_t to_ms_since_boot (absolute_time_t t)`
  Convert a timestamp into a number of milliseconds since boot. More...
- `static absolute_time_t delayed_by_us (const absolute_time_t t, uint64_t us)`
  Return a timestamp value obtained by adding a number of microseconds to another timestamp. More...
- `static absolute_time_t delayed_by_ms (const absolute_time_t t, uint32_t ms)`
  Return a timestamp value obtained by adding a number of milliseconds to another timestamp. More...
- `static absolute_time_t make_timeout_time_us (uint64_t us)`
  Convenience method to get the timestamp a number of microseconds from the current time. More...
- `static absolute_time_t make_timeout_time_ms (uint32_t ms)`
  Convenience method to get the timestamp a number of milliseconds from the current time. More...
- `static int64_t absolute_time_diff_us (absolute_time_t from, absolute_time_t to)`
  Return the difference in microseconds between two timestamps. More...
- `static bool is_nil_time (absolute_time_t t)`
  Determine if the given timestamp is nil. More...

3.45.2. Detailed Description

Timestamp functions relating to points in time (including the current time)

These are functions for dealing with timestamps (i.e. instants in time) represented by the type absolute_time_t. This opaque type is provided to help prevent accidental mixing of timestamps and relative time values.

3.45.3. Function Documentation

3.45.3.1. absolute_time_diff_us

```
static int64_t absolute_time_diff_us (absolute_time_t from,
        absolute_time_t to)
```

Return the difference in microseconds between two timestamps.

Parameters

- `from` the first timestamp

- to the second timestamp

Returns

- the number of microseconds between the two timestamps (positive if to is after from)

3.45.3.2. delayed_by_ms

```
static absolute_time_t delayed_by_ms (const absolute_time_t t,
       uint32_t ms)
```

Return a timestamp value obtained by adding a number of milliseconds to another timestamp.

Parameters

- t the base timestamp
- ms the number of milliseconds to add

Returns

- the timestamp representing the resulting time

3.45.3.3. delayed_by_us

```
static absolute_time_t delayed_by_us (const absolute_time_t t,
       uint64_t us)
```

Return a timestamp value obtained by adding a number of microseconds to another timestamp.

Parameters

- t the base timestamp
- us the number of microseconds to add

Returns

- the timestamp representing the resulting time

3.45.3.4. get_absolute_time

```
static absolute_time_t get_absolute_time ()
```

Return a representation of the current time.

Returns an opaque high fidelity representation of the current time sampled during the call.

Returns

- the absolute time (now) of the hardware timer

See also

- absolute_time_t
- sleep_until()
- time_us_64()

3.45.3.5. is_nil_time

```
static bool is_nil_time (absolute_time_t t)
```

Determine if the given timestamp is nil.

Parameters

- `t` the timestamp

Returns

- true if the timestamp is nil

See also

- nil_time()

3.45.3.6. make_timeout_time_ms

```
static absolute_time_t make_timeout_time_ms (uint32_t ms)
```

Convenience method to get the timestamp a number of milliseconds from the current time.

Parameters

- `ms` the number of milliseconds to add to the current timestamp

Returns

- the future timestamp

3.45.3.7. make_timeout_time_us

```
static absolute_time_t make_timeout_time_us (uint64_t us)
```

Convenience method to get the timestamp a number of microseconds from the current time.

Parameters

- `us` the number of microseconds to add to the current timestamp

Returns

- the future timestamp

3.45.3.8. to_ms_since_boot

```
static uint32_t to_ms_since_boot (absolute_time_t t)
```

Convert a timestamp into a number of milliseconds since boot.

fn to_ms_since_boot

Parameters

- `t` an absolute_time_t value to convert

Returns

- the number of microseconds since boot represented by t

See also

- to_us_since_boot

3.46. sleep

Sleep functions for delaying execution in a lower power state. More…

3.46.1. Functions

- `void sleep_until (absolute_time_t target)`
  Wait until after the given timestamp to return. More…
- `void sleep_us (uint64_t us)`
  Wait for the given number of microseconds before returning. More…
- `void sleep_ms (uint32_t ms)`
  Wait for the given number of milliseconds before returning. More…
- `bool best_effort_wfe_or_timeout (absolute_time_t timeout_timestamp)`
  Helper method for blocking on a timeout. More…

3.46.2. Detailed Description

Sleep functions for delaying execution in a lower power state.

These functions allow the calling core to sleep. This is a lower powered sleep; waking and re-checking time on every processor event (WFE)

These functions should not be called from an IRQ handler.

Lower powered sleep requires use of the default alarm pool which may be disabled by the PICO_TIME_DEFAULT_ALARM_POOL_DISABLED define or currently full in which case these functions become busy waits instead.

Whilst *sleep_* functions are preferable to *busy_wait* functions from a power perspective, the *busy_wait* equivalent function may return slightly sooner after the target is reached.

- busy_wait_until()
- busy_wait_us()
- busy_wait_us_32()

3.46.3. Function Documentation

3.46.3.1. best_effort_wfe_or_timeout

`bool best_effort_wfe_or_timeout (absolute_time_t timeout_timestamp)`

Helper method for blocking on a timeout.

This method will return in response to a an event (as per __wfe) or when the target time is reached, or at any point before.

This method can be used to implement a lower power polling loop waiting on some condition signalled by an event (`__sev()`).

This is called because under certain circumstances (notably the default timer pool being disabled or full) the best effort is simply to return immediately without a __wfe, thus turning the calling code into a busy wait.

Example usage:

Parameters

- `timeout_timestamp` the timeout time

Returns

- true if the target time is reached, false otherwise

3.46.3.2. sleep_ms

`void sleep_ms (uint32_t ms)`

Wait for the given number of milliseconds before returning.

Parameters

- `ms` the number of milliseconds to sleep

3.46.3.3. sleep_until

`void sleep_until (absolute_time_t target)`

Wait until after the given timestamp to return.

Parameters

- `target` the time after which to return

See also

- sleep_us()
- busy_wait_until()

3.46.3.4. sleep_us

`void sleep_us (uint64_t us)`

Wait for the given number of microseconds before returning.

Parameters

- `us` the number of microseconds to sleep

See also

- busy_wait_us()

3.47. alarm

Alarm functions for scheduling future execution. More…

3.47.1. Typedefs

- `typedef int32_t alarm_id_t`
  The identifier for an alarm. More…
- `typedef int64_t(* alarm_callback_t )(alarm_id_t id, void *user_data)`
  User alarm callback. More…

3.47.2. Macros

- `#define PICO_TIME_DEFAULT_ALARM_POOL_DISABLED 0`
  If 1 then the default alarm pool is disabled (so no hardware alarm is claimed for the pool) More…
- `#define PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM 3`
  Selects which hardware alarm is used for the default alarm pool. More…

- `#define PICO_TIME_DEFAULT_ALARM_POOL_MAX_TIMERS 16`
  Selects the maximum number of concurrent timers in the default alarm pool. More…

3.47.3. Functions

- `void alarm_pool_init_default ()`
  Create the default alarm pool (if not already created or disabled)
- `alarm_pool_t * alarm_pool_default ()`
  The default alarm pool used when alarms are added without specifying an alarm pool, and also used by the Pico SDK to support lower power sleeps and timeouts. More…
- `alarm_pool_t * alarm_pool_create (uint hardware_alarm_num, uint max_timers)`
  Create an alarm pool. More…
- `uint alarm_pool_hardware_alarm_num (alarm_pool_t *pool)`
  Return the hardware alarm used by an alarm pool. More…
- `void alarm_pool_destroy (alarm_pool_t *pool)`
  Destroy the alarm pool, cancelling all alarms and freeing up the underlying hardware alarm. More…
- `alarm_id_t alarm_pool_add_alarm_at (alarm_pool_t *pool, absolute_time_t time, alarm_callback_t callback, void *user_data, bool fire_if_past)`
  Add an alarm callback to be called at a specific time. More…
- `static alarm_id_t alarm_pool_add_alarm_in_us (alarm_pool_t *pool, uint64_t us, alarm_callback_t callback, void *user_data, bool fire_if_past)`
  Add an alarm callback to be called after a delay specified in microseconds. More…
- `static alarm_id_t alarm_pool_add_alarm_in_ms (alarm_pool_t *pool, uint32_t ms, alarm_callback_t callback, void *user_data, bool fire_if_past)`
  Add an alarm callback to be called after a delay specified in milliseconds. More…
- `bool alarm_pool_cancel_alarm (alarm_pool_t *pool, alarm_id_t alarm_id)`
  Cancel an alarm. More…
- `static alarm_id_t add_alarm_at (absolute_time_t time, alarm_callback_t callback, void *user_data, bool fire_if_past)`
  Add an alarm callback to be called at a specific time. More…
- `static alarm_id_t add_alarm_in_us (uint64_t us, alarm_callback_t callback, void *user_data, bool fire_if_past)`
  Add an alarm callback to be called after a delay specified in microseconds. More…
- `static alarm_id_t add_alarm_in_ms (uint32_t ms, alarm_callback_t callback, void *user_data, bool fire_if_past)`
  Add an alarm callback to be called after a delay specified in milliseconds. More…
- `static bool cancel_alarm (alarm_id_t alarm_id)`
  Cancel an alarm from the default alarm pool. More…

3.47.4. Detailed Description

Alarm functions for scheduling future execution.

Alarms are added to alarm pools, which may hold a certain fixed number of active alarms. Each alarm pool utilizes one of four underlying hardware alarms, thus you may have up to four alarm pools. An alarm pool calls (except when the callback would happen before or during being set) the callback on the core from which the alarm pool was created. Callbacks are called from the hardware alarm IRQ handler, so care must be taken in their implementation.

A default pool is created (todo unless not) on the core specified by PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM on core 0, and may be used by the method variants that take no alarm pool parameter.

See also

- struct alarm_pool
- hardware_timer

3.47.5. Function Documentation

3.47.5.1. add_alarm_at

```
static alarm_id_t add_alarm_at (absolute_time_t time,
       alarm_callback_t callback,
       void *user_data,
       bool fire_if_past)
```

Add an alarm callback to be called at a specific time.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core of the default alarm pool (generally core 0). If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- `time` the timestamp when (after which) the callback should fire
- `callback` the callback function
- `user_data` user data to pass to the callback function
- `fire_if_past` if true, this method will call the callback itself before returning 0 if the timestamp happens before or during this method call

Returns

- >0 the alarm id
- 0 the target timestamp was during or before this method call (whether the callback was called depends on fire_if_past)
- -1 if there were no alarm slots available

3.47.5.2. add_alarm_in_ms

```
static alarm_id_t add_alarm_in_ms (uint32_t ms,
       alarm_callback_t callback,
       void *user_data,
       bool fire_if_past)
```

Add an alarm callback to be called after a delay specified in milliseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core of the default alarm pool (generally core 0). If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- `ms` the delay (from now) in milliseconds when (after which) the callback should fire
- `callback` the callback function
- `user_data` user data to pass to the callback function
- `fire_if_past` if true, this method will call the callback itself before returning 0 if the timestamp happens before or during this method call

Returns

- >0 the alarm id
- 0 the target timestamp was during or before this method call (whether the callback was called depends on fire_if_past)
- -1 if there were no alarm slots available

3.47.5.3. add_alarm_in_us

```
static alarm_id_t add_alarm_in_us (uint64_t us,
       alarm_callback_t callback,
       void *user_data,
       bool fire_if_past)
```

Add an alarm callback to be called after a delay specified in microseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core of the default alarm pool (generally core 0). If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- `us` the delay (from now) in microseconds when (after which) the callback should fire
- `callback` the callback function
- `user_data` user data to pass to the callback function
- `fire_if_past` if true, this method will call the callback itself before returning 0 if the timestamp happens before or during this method call

Returns

- >0 the alarm id
- 0 the target timestamp was during or before this method call (whether the callback was called depends on fire_if_past)
- -1 if there were no alarm slots available

3.47.5.4. alarm_pool_add_alarm_at

```
alarm_id_t alarm_pool_add_alarm_at (alarm_pool_t *pool,
       absolute_time_t time,
       alarm_callback_t callback,
       void *user_data,
       bool fire_if_past)
```

Add an alarm callback to be called at a specific time.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core the alarm pool was created on. If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- `pool` the alarm pool to use for scheduling the callback (this determines which hardware alarm is used, and which core calls the callback)
- `time` the timestamp when (after which) the callback should fire
- `callback` the callback function
- `user_data` user data to pass to the callback function
- `fire_if_past` if true, this method will call the callback itself before returning 0 if the timestamp happens before or

during this method call

Returns

- >0 the alarm id
- 0 the target timestamp was during or before this method call (whether the callback was called depends on fire_if_past)
- -1 if there were no alarm slots available

3.47.5.5. alarm_pool_add_alarm_in_ms

```
static alarm_id_t alarm_pool_add_alarm_in_ms (alarm_pool_t *pool,
       uint32_t ms,
       alarm_callback_t callback,
       void *user_data,
       bool fire_if_past)
```

Add an alarm callback to be called after a delay specified in milliseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core the alarm pool was created on. If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- `pool` the alarm pool to use for scheduling the callback (this determines which hardware alarm is used, and which core calls the callback)
- `ms` the delay (from now) in milliseconds when (after which) the callback should fire
- `callback` the callback function
- `user_data` user data to pass to the callback function
- `fire_if_past` if true, this method will call the callback itself before returning 0 if the timestamp happens before or during this method call

Returns

- >0 the alarm id
- 0 the target timestamp was during or before this method call (whether the callback was called depends on fire_if_past)
- -1 if there were no alarm slots available

3.47.5.6. alarm_pool_add_alarm_in_us

```
static alarm_id_t alarm_pool_add_alarm_in_us (alarm_pool_t *pool,
       uint64_t us,
       alarm_callback_t callback,
       void *user_data,
       bool fire_if_past)
```

Add an alarm callback to be called after a delay specified in microseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core the alarm pool was created on. If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- `pool` the alarm pool to use for scheduling the callback (this determines which hardware alarm is used, and which core calls the callback)

- `us` the delay (from now) in microseconds when (after which) the callback should fire
- `callback` the callback function
- `user_data` user data to pass to the callback function
- `fire_if_past` if true, this method will call the callback itself before returning 0 if the timestamp happens before or during this method call

Returns

- >0 the alarm id
- 0 the target timestamp was during or before this method call (whether the callback was called depends on fire_if_past)
- -1 if there were no alarm slots available

3.47.5.7. alarm_pool_cancel_alarm

```
bool alarm_pool_cancel_alarm (alarm_pool_t *pool,
       alarm_id_t alarm_id)
```

Cancel an alarm.

Parameters

- `pool` the alarm_pool containing the alarm
- `alarm_id` the alarm

Returns

- true if the alarm was cancelled, false if it didn't exist

See also

- alarm_id_t for a note on reuse of IDs

3.47.5.8. alarm_pool_create

```
alarm_pool_t* alarm_pool_create (uint hardware_alarm_num,
       uint max_timers)
```

Create an alarm pool.

The alarm pool will call callbacks from an alarm IRQ Handler on the core of this function is called from.

In many situations there is never any need for anything other than the default alarm pool, however you might want to create another if you want alarm callbacks on core 1 or require alarm pools of different priority (IRQ priority based preemption of callbacks)

Parameters

- `hardware_alarm_num` the hardware alarm to use to back this pool
- `max_timers` the maximum number of timers

See also

- alarm_pool_default()
- hardware_claiming

3.47.5.9. alarm_pool_default

`alarm_pool_t* alarm_pool_default ()`

The default alarm pool used when alarms are added without specifying an alarm pool, and also used by the Pico SDK to support lower power sleeps and timeouts.

See also

- PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM
- alarm_pool_default()

3.47.5.10. alarm_pool_destroy

`void alarm_pool_destroy (alarm_pool_t *pool)`

Destroy the alarm pool, cancelling all alarms and freeing up the underlying hardware alarm.

Parameters

- `pool` the pool

Returns

- the hardware alarm used by the pool

3.47.5.11. alarm_pool_hardware_alarm_num

`uint alarm_pool_hardware_alarm_num (alarm_pool_t *pool)`

Return the hardware alarm used by an alarm pool.

Parameters

- `pool` the pool

Returns

- the hardware alarm used by the pool

3.47.5.12. alarm_pool_init_default

`void alarm_pool_init_default ()`

Create the default alarm pool (if not already created or disabled)

3.47.5.13. cancel_alarm

`static bool cancel_alarm (alarm_id_t alarm_id)`

Cancel an alarm from the default alarm pool.

Parameters

- `alarm_id` the alarm

Returns

- true if the alarm was cancelled, false if it didn't exist

See also

- alarm_id_t for a note on reuse of IDs

3.48. repeating_timer

Repeating Timer functions for simple scheduling of repeated execution. More…

3.48.1. Data Structures

- `struct repeating_timer`
  Information about a repeating timer.

3.48.2. Typedefs

- `typedef bool(* repeating_timer_callback_t )(repeating_timer_t *rt)`
  Callback for a repeating timer. More…

3.48.3. Functions

- `bool alarm_pool_add_repeating_timer_us (alarm_pool_t *pool, int64_t delay_us, repeating_timer_callback_t callback, void *user_data, repeating_timer_t *out)`
  Add a repeating timer that is called repeatedly at the specified interval in microseconds. More…
- `static bool alarm_pool_add_repeating_timer_ms (alarm_pool_t *pool, int32_t delay_ms, repeating_timer_callback_t callback, void *user_data, repeating_timer_t *out)`
  Add a repeating timer that is called repeatedly at the specified interval in milliseconds. More…
- `static bool add_repeating_timer_us (int64_t delay_us, repeating_timer_callback_t callback, void *user_data, repeating_timer_t *out)`
  Add a repeating timer that is called repeatedly at the specified interval in microseconds. More…
- `static bool add_repeating_timer_ms (int32_t delay_ms, repeating_timer_callback_t callback, void *user_data, repeating_timer_t *out)`
  Add a repeating timer that is called repeatedly at the specified interval in milliseconds. More…
- `bool cancel_repeating_timer (repeating_timer_t *timer)`
  Cancel a repeating timer. More…

3.48.4. Detailed Description

Repeating Timer functions for simple scheduling of repeated execution.

The regular *alarm_* functionality can be used to make repeating alarms (by return non zero from the callback), however these methods abstract that further (at the cost of a user structure to store the repeat delay in (which the alarm framework does not have space for).

3.48.5. Function Documentation

3.48.5.1. add_repeating_timer_ms

```
static bool add_repeating_timer_ms (int32_t delay_ms,
       repeating_timer_callback_t callback,
       void *user_data,
       repeating_timer_t *out)
```

Add a repeating timer that is called repeatedly at the specified interval in milliseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core of the default alarm pool (generally core 0). If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- `delay_ms` the repeat delay in milliseconds; if >0 then this is the delay between one callback ending and the next starting; if <0 then this is the negative of the time between the starts of the callbacks. The value of 0 is treated as 1 microsecond
- `callback` the repeating timer callback function
- `user_data` user data to pass to store in the repeating_timer structure for use by the callback.
- `out` the pointer to the user owned structure to store the repeating timer info in. BEWARE this storage location must outlive the repeating timer, so be careful of using stack space

Returns

- false if there were no alarm slots available to create the timer, true otherwise.

3.48.5.2. add_repeating_timer_us

```
static bool add_repeating_timer_us (int64_t delay_us,
        repeating_timer_callback_t callback,
        void *user_data,
        repeating_timer_t *out)
```

Add a repeating timer that is called repeatedly at the specified interval in microseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core of the default alarm pool (generally core 0). If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- `delay_us` the repeat delay in microseconds; if >0 then this is the delay between one callback ending and the next starting; if <0 then this is the negative of the time between the starts of the callbacks. The value of 0 is treated as 1
- `callback` the repeating timer callback function
- `user_data` user data to pass to store in the repeating_timer structure for use by the callback.
- `out` the pointer to the user owned structure to store the repeating timer info in. BEWARE this storage location must outlive the repeating timer, so be careful of using stack space

Returns

- false if there were no alarm slots available to create the timer, true otherwise.

3.48.5.3. alarm_pool_add_repeating_timer_ms

```
static bool alarm_pool_add_repeating_timer_ms (alarm_pool_t *pool,
        int32_t delay_ms,
        repeating_timer_callback_t callback,
        void *user_data,
        repeating_timer_t *out)
```

Add a repeating timer that is called repeatedly at the specified interval in milliseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core the alarm pool was created on. If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- pool the alarm pool to use for scheduling the repeating timer (this determines which hardware alarm is used, and which core calls the callback)
- delay_ms the repeat delay in milliseconds; if >0 then this is the delay between one callback ending and the next starting; if <0 then this is the negative of the time between the starts of the callbacks. The value of 0 is treated as 1 microsecond
- callback the repeating timer callback function
- user_data user data to pass to store in the repeating_timer structure for use by the callback.
- out the pointer to the user owned structure to store the repeating timer info in. BEWARE this storage location must outlive the repeating timer, so be careful of using stack space

Returns

- false if there were no alarm slots available to create the timer, true otherwise.

3.48.5.4. alarm_pool_add_repeating_timer_us

```
bool alarm_pool_add_repeating_timer_us (alarm_pool_t *pool,
       int64_t delay_us,
       repeating_timer_callback_t callback,
       void *user_data,
       repeating_timer_t *out)
```

Add a repeating timer that is called repeatedly at the specified interval in microseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core the alarm pool was created on. If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- pool the alarm pool to use for scheduling the repeating timer (this determines which hardware alarm is used, and which core calls the callback)
- delay_us the repeat delay in microseconds; if >0 then this is the delay between one callback ending and the next starting; if <0 then this is the negative of the time between the starts of the callbacks. The value of 0 is treated as 1
- callback the repeating timer callback function
- user_data user data to pass to store in the repeating_timer structure for use by the callback.
- out the pointer to the user owned structure to store the repeating timer info in. BEWARE this storage location must outlive the repeating timer, so be careful of using stack space

Returns

- false if there were no alarm slots available to create the timer, true otherwise.

3.48.5.5. cancel_repeating_timer

```
bool cancel_repeating_timer (repeating_timer_t *timer)
```

Cancel a repeating timer.

Parameters

- timer the repeating timer to cancel

Returns

- true if the repeating timer was cancelled, false if it didn't exist

See also

- alarm_id_t for a note on reuse of IDs

3.49. datetime

Date/Time formatting. More...

3.49.1. Data Structures

- `struct datetime_t`
  Structure containing date and time information.

3.49.2. Functions

- `void datetime_to_str (char *buf, uint buf_size, const datetime_t *t)`
  Convert a datetime_t structure to a string. More...

3.49.3. Detailed Description

Date/Time formatting.

3.49.4. Function Documentation

3.49.4.1. datetime_to_str

```
void datetime_to_str (char *buf,
       uint buf_size,
       const datetime_t *t)
```

Convert a datetime_t structure to a string.

Parameters

- `buf` character buffer to accept generated string
- `buf_size` The size of the passed in buffer
- `t` The datetime to be converted.

3.50. pheap

Pairing Heap Implementation. More...

3.50.1. Detailed Description

Pairing Heap Implementation.

pheap defines a simple pairing heap. the implementation simply tracks array indexes, it is up to the user to provide storage for heap entries and a comparison function.

NOTE

this class is not safe for concurrent usage. It should be externally protected. Furthermore if used concurrently, the caller needs to protect around their use of the returned id. for example, ph_remove_head returns the id of an element that is no longer in the heap.

The user can still use this to look at the data in their companion array, however obviously further operations on the heap may cause them to overwrite that data as the id may be reused on subsequent operations

3.51. queue

Multi-core and IRQ safe queue implementation. More…

3.51.1. Functions

- `void queue_init_with_spinlock (queue_t *q, uint element_size, uint element_count, uint spinlock_num)`
  Initialise a queue with a specific spinlock for concurrency protection. More…
- `static void queue_init (queue_t *q, uint element_size, uint element_count)`
  Initialise a queue, allocating a (possibly shared) spinlock. More…
- `void queue_free (queue_t *q)`
  Destroy the specified queue. More…
- `static uint queue_get_level_unsafe (queue_t *q)`
  Unsafe check of level of the specified queue. More…
- `static uint queue_get_level (queue_t *q)`
  Check of level of the specified queue. More…
- `static bool queue_is_empty (queue_t *q)`
  Check if queue is empty. More…
- `static bool queue_is_full (queue_t *q)`
  Check if queue is full. More…
- `bool queue_try_add (queue_t *q, void *data)`
  Non-blocking add value queue if not full. More…
- `bool queue_try_remove (queue_t *q, void *data)`
  Non-blocking removal of entry from the queue if non empty. More…
- `bool queue_try_peek (queue_t *q, void *data)`
  Non-blocking peek at the next item to be removed from the queue. More…
- `void queue_add_blocking (queue_t *q, void *data)`
  Blocking add of value to queue. More…
- `void queue_remove_blocking (queue_t *q, void *data)`
  Blocking remove entry from queue. More…
- `void queue_peek_blocking (queue_t *q, void *data)`
  Blocking peek at next value to be removed from queue. More…

3.51.2. Detailed Description

Multi-core and IRQ safe queue implementation.

Note that this queue stores values of a specified size, and pushed values are copied into the queue

3.51.3. Function Documentation

3.51.3.1. queue_add_blocking

```
void queue_add_blocking (queue_t *q,
        void *data)
```

Blocking add of value to queue.

If the queue is full this function will block, until a removal happens on the queue

Parameters

- `q` Pointer to a queue_t structure, used as a handle
- `data` Pointer to value to be copied into the queue

3.51.3.2. queue_free

```
void queue_free (queue_t *q)
```

Destroy the specified queue.

Does not deallocate the `queue_t` structure itself.

Parameters

- `q` Pointer to a queue_t structure, used as a handle

3.51.3.3. queue_get_level

```
static uint queue_get_level (queue_t *q)
```

Check of level of the specified queue.

Parameters

- `q` Pointer to a queue_t structure, used as a handle

Returns

- Number of entries in the queue

3.51.3.4. queue_get_level_unsafe

```
static uint queue_get_level_unsafe (queue_t *q)
```

Unsafe check of level of the specified queue.

This does not use the spinlock, so may return incorrect results if the spin lock is not externally locked

Parameters

- `q` Pointer to a queue_t structure, used as a handle

Returns

- Number of entries in the queue

3.51.3.5. queue_init

```
static void queue_init (queue_t *q,
        uint element_size,
       uint element_count)
```

Initialise a queue, allocating a (possibly shared) spinlock.

Parameters

- `q` Pointer to a queue_t structure, used as a handle
- `element_size` Size of each value in the queue
- `element_count` Maximum number of entries in the queue

3.51.3.6. queue_init_with_spinlock

```
void queue_init_with_spinlock (queue_t *q,
       uint element_size,
       uint element_count,
       uint spinlock_num)
```

Initialise a queue with a specific spinlock for concurrency protection.

Parameters

- `q` Pointer to a queue_t structure, used as a handle
- `element_size` Size of each value in the queue
- `element_count` Maximum number of entries in the queue
- `spinlock_num` The spin ID used to protect the queue

3.51.3.7. queue_is_empty

```
static bool queue_is_empty (queue_t *q)
```

Check if queue is empty.

This function is interrupt and multicore safe.

Parameters

- `q` Pointer to a queue_t structure, used as a handle

Returns

- true if queue is empty, false otherwise

3.51.3.8. queue_is_full

```
static bool queue_is_full (queue_t *q)
```

Check if queue is full.

This function is interrupt and multicore safe.

Parameters

- `q` Pointer to a queue_t structure, used as a handle

Returns

- true if queue is full, false otherwise

3.51.3.9. queue_peek_blocking

```
void queue_peek_blocking (queue_t *q,
        void *data)
```

Blocking peek at next value to be removed from queue.

If the queue is empty function will block until a value is added

Parameters

- `q` Pointer to a queue_t structure, used as a handle
- `data` Pointer to the location to receive the peeked value

3.51.3.10. queue_remove_blocking

```
void queue_remove_blocking (queue_t *q,
        void *data)
```

Blocking remove entry from queue.

If the queue is empty this function will block until a value is added.

Parameters

- `q` Pointer to a queue_t structure, used as a handle
- `data` Pointer to the location to receive the removed value

3.51.3.11. queue_try_add

```
bool queue_try_add (queue_t *q,
        void *data)
```

Non-blocking add value queue if not full.

If the queue is full this function will return immediately with false, otherwise the data is copied into a new value added to the queue, and this function will return true.

Parameters

- `q` Pointer to a queue_t structure, used as a handle
- `data` Pointer to value to be copied into the queue

Returns

- true if the value was added

3.51.3.12. queue_try_peek

```
bool queue_try_peek (queue_t *q,
        void *data)
```

Non-blocking peek at the next item to be removed from the queue.

If the queue is not empty this function will return immediately with true with the peeked entry copied into the location specified by the data parameter, otherwise the function will return false.

Parameters

- `q` Pointer to a queue_t structure, used as a handle
- `data` Pointer to the location to receive the peeked value

Returns

- true if there was a value to peek

3.51.3.13. queue_try_remove

```
bool queue_try_remove (queue_t *q,
       void *data)
```

Non-blocking removal of entry from the queue if non empty.

If the queue is not empty function will copy the removed value into the location provided and return immediately with true, otherwise the function will return immediately with false.

Parameters

- `q` Pointer to a queue_t structure, used as a handle
- `data` Pointer to the location to receive the removed value

Returns

- true if a value was removed

3.52. Third-party Libraries

3.52.1. Modules

- **`tinyusb_device`**
  TinyUSB Device-mode support for the RP2040
- **`tinyusb_host`**
  TinyUSB Host-mode support for the RP2040

3.53. tinyusb_device

TinyUSB Device-mode support for the RP2040 More…

3.53.1. Detailed Description

TinyUSB Device-mode support for the RP2040

3.54. tinyusb_host

TinyUSB Host-mode support for the RP2040 More…

3.54.1. Detailed Description

TinyUSB Host-mode support for the RP2040

3.55. Runtime Infrastructure

3.55.1. Modules

- `pico_base`
  Core types and macros for the Pico SDK. This header is intended to be included by all source code.
- `pico_bit_ops`
  Optimized bit manipulation functions. Additionally provides replacement implementations of the compiler built-ins *builtin_popcount,* builtin_clz and __bulitin_ctz.
- `pico_bootrom`
  Access to functions in the RP2040 bootrom.
- `pico_divider`
  Optimized 32 and 64 bit division functions accelerated by the RP2040 hardware divider. Additionally provides integration with the C / and % operators.
- `pico_double`
  Optimized double-precision floating point functions including replacement implementations of the compiler built-ins.
- `pico_float`
  Optimized single-precision floating point functions including replacement implementations of the compiler built-ins.
- `pico_int64`
  Optimized replacement implementations of the compiler built-in 64 bit multiplication.
- `pico_malloc`
  Multi-core safety for malloc, calloc and free.
- `pico_memory`
  Optimized replacement implementations of the compiler built-in memcpy and memset.
- `pico_platform`
  Compiler definitions for the selected PICO_PLATFORM.
- `pico_runtime`
  Aggregate runtime support including pico_bit_ops, pico_divider, pico_double, pico_int64, pico_float, pico_malloc, pico_memory and pico_standard_link.
- `pico_standard_link`

3.56. pico_base

Core types and macros for the Pico SDK. This header is intended to be included by all source code. More…

3.56.1. Detailed Description

Core types and macros for the Pico SDK. This header is intended to be included by all source code.

3.57. pico_bit_ops

Optimized bit manipulation functions. Additionally provides replacement implementations of the compiler built-ins *builtin_popcount,* builtin_clz and __bulitin_ctz. More…

3.57.1. Functions

- `uint32_t __rev (uint32_t bits)`
  Reverse the bits in a 32 bit word. More…

- `uint64_t __revll (uint64_t bits)`
  Reverse the bits in a 64 bit double word. More…

3.57.2. Detailed Description

Optimized bit manipulation functions. Additionally provides replacement implementations of the compiler built-ins *builtin_popcount,* builtin_clz and __bulitin_ctz.

3.57.3. Function Documentation

3.57.3.1. __rev

`uint32_t __rev (uint32_t bits)`

Reverse the bits in a 32 bit word.

Parameters

- `bits` 32 bit input

Returns

- the 32 input bits reversed

3.57.3.2. __revll

`uint64_t __revll (uint64_t bits)`

Reverse the bits in a 64 bit double word.

Parameters

- `bits` 64 bit input

Returns

- the 64 input bits reversed

3.58. pico_bootrom

Access to functions in the RP2040 bootrom. More…

3.58.1. Detailed Description

Access to functions in the RP2040 bootrom.

3.59. pico_divider

Optimized 32 and 64 bit division functions accelerated by the RP2040 hardware divider. Additionally provides integration with the C / and % operators. More…

3.59.1. Files

- file **divider.h**
  High level APIs including combined quotient and remainder functions for 32 and 64 bit accelerated by the hardware divider.

3.59.2. Functions

- `int32_t div_s32s32 (int32_t a, int32_t b)`
  Integer divide of two signed 32-bit values. More…
- `static int32_t divmod_s32s32_rem (int32_t a, int32_t b, int32_t *rem)`
  Integer divide of two signed 32-bit values, with remainder. More…
- `divmod_result_t divmod_s32s32 (int32_t a, int32_t b)`
  Integer divide of two signed 32-bit values. More…
- `uint32_t div_u32u32 (uint32_t a, uint32_t b)`
  Integer divide of two unsigned 32-bit values. More…
- `static uint32_t divmod_u32u32_rem (uint32_t a, uint32_t b, uint32_t *rem)`
  Integer divide of two unsigned 32-bit values, with remainder. More…
- `divmod_result_t divmod_u32u32 (uint32_t a, uint32_t b)`
  Integer divide of two unsigned 32-bit values. More…
- `int64_t div_s64s64 (int64_t a, int64_t b)`
  Integer divide of two signed 64-bit values. More…
- `int64_t divmod_s64s64_rem (int64_t a, int64_t b, int64_t *rem)`
  Integer divide of two signed 64-bit values, with remainder. More…
- `int64_t divmod_s64s64 (int64_t a, int64_t b)`
  Integer divide of two signed 64-bit values. More…
- `uint64_t div_u64u64 (uint64_t a, uint64_t b)`
  Integer divide of two unsigned 64-bit values. More…
- `uint64_t divmod_u64u64_rem (uint64_t a, uint64_t b, uint64_t *rem)`
  Integer divide of two unsigned 64-bit values, with remainder. More…
- `uint64_t divmod_u64u64 (uint64_t a, uint64_t b)`
  Integer divide of two signed 64-bit values. More…
- `int32_t div_s32s32_unsafe (int32_t a, int32_t b)`
  Unsafe integer divide of two signed 32-bit values. More…
- `int32_t divmod_s32s32_rem_unsafe (int32_t a, int32_t b, int32_t *rem)`
  Unsafe integer divide of two signed 32-bit values, with remainder. More…
- `int64_t divmod_s32s32_unsafe (int32_t a, int32_t b)`
  Unsafe integer divide of two unsigned 32-bit values. More…
- `uint32_t div_u32u32_unsafe (uint32_t a, uint32_t b)`
  Unsafe integer divide of two unsigned 32-bit values. More…
- `uint32_t divmod_u32u32_rem_unsafe (uint32_t a, uint32_t b, uint32_t *rem)`
  Unsafe integer divide of two unsigned 32-bit values, with remainder. More…
- `uint64_t divmod_u32u32_unsafe (uint32_t a, uint32_t b)`
  Unsafe integer divide of two unsigned 32-bit values. More…
- `int64_t div_s64s64_unsafe (int64_t a, int64_t b)`
  Unsafe integer divide of two signed 64-bit values. More…

- `int64_t divmod_s64s64_rem_unsafe (int64_t a, int64_t b, int64_t *rem)`
  Unsafe integer divide of two signed 64-bit values, with remainder. More…
- `int64_t divmod_s64s64_unsafe (int64_t a, int64_t b)`
  Unsafe integer divide of two signed 64-bit values. More…
- `uint64_t div_u64u64_unsafe (uint64_t a, uint64_t b)`
  Unsafe integer divide of two unsigned 64-bit values. More…
- `uint64_t divmod_u64u64_rem_unsafe (uint64_t a, uint64_t b, uint64_t *rem)`
  Unsafe integer divide of two unsigned 64-bit values, with remainder. More…
- `uint64_t divmod_u64u64_unsafe (uint64_t a, uint64_t b)`
  Unsafe integer divide of two signed 64-bit values. More…

3.59.3. Detailed Description

Optimized 32 and 64 bit division functions accelerated by the RP2040 hardware divider. Additionally provides integration with the C / and % operators.

3.59.4. Function Documentation

3.59.4.1. div_s32s32

```
int32_t div_s32s32 (int32_t a,
        int32_t b)
```

Integer divide of two signed 32-bit values.

Parameters

- `a` Dividend
- `b` Divisor

Returns

- quotient

3.59.4.2. div_s32s32_unsafe

```
int32_t div_s32s32_unsafe (int32_t a,
        int32_t b)
```

Unsafe integer divide of two signed 32-bit values.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor

Returns

- quotient

3.59.4.3. div_s64s64

```
int64_t div_s64s64 (int64_t a,
       int64_t b)
```

Integer divide of two signed 64-bit values.

Parameters

- `a` Dividend
- `b` Divisor

Returns

- Quotient

3.59.4.4. div_s64s64_unsafe

```
int64_t div_s64s64_unsafe (int64_t a,
       int64_t b)
```

Unsafe integer divide of two signed 64-bit values.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor

Returns

- Quotient

3.59.4.5. div_u32u32

```
uint32_t div_u32u32 (uint32_t a,
       uint32_t b)
```

Integer divide of two unsigned 32-bit values.

Parameters

- `a` Dividend
- `b` Divisor

Returns

- Quotient

3.59.4.6. div_u32u32_unsafe

```
uint32_t div_u32u32_unsafe (uint32_t a,
       uint32_t b)
```

Unsafe integer divide of two unsigned 32-bit values.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor

Returns

- Quotient

3.59.4.7. div_u64u64

```
uint64_t div_u64u64 (uint64_t a,
        uint64_t b)
```

Integer divide of two unsigned 64-bit values.

Parameters

- `a` Dividend
- `b` Divisor

Returns

- Quotient

3.59.4.8. div_u64u64_unsafe

```
uint64_t div_u64u64_unsafe (uint64_t a,
        uint64_t b)
```

Unsafe integer divide of two unsigned 64-bit values.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor

Returns

- Quotient

3.59.4.9. divmod_s32s32

```
divmod_result_t divmod_s32s32 (int32_t a,
        int32_t b)
```

Integer divide of two signed 32-bit values.

Parameters

- `a` Dividend
- `b` Divisor

Returns

- quotient in low word/r0, remainder in high word/r1

3.59.4.10. divmod_s32s32_rem

```
static int32_t divmod_s32s32_rem (int32_t a,
        int32_t b,
        int32_t *rem)
```

Integer divide of two signed 32-bit values, with remainder.

Parameters

- `a` Dividend
- `b` Divisor

- `rem` The remainder of dividend/divisor

Returns

- Quotient result of dividend/divisor

3.59.4.11. divmod_s32s32_rem_unsafe

```
int32_t divmod_s32s32_rem_unsafe (int32_t a,
        int32_t b,
        int32_t *rem)
```

Unsafe integer divide of two signed 32-bit values, with remainder.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor
- `rem` The remainder of dividend/divisor

Returns

- Quotient result of dividend/divisor

3.59.4.12. divmod_s32s32_unsafe

```
int64_t divmod_s32s32_unsafe (int32_t a,
        int32_t b)
```

Unsafe integer divide of two unsigned 32-bit values.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor

Returns

- quotient in low word/r0, remainder in high word/r1

3.59.4.13. divmod_s64s64

```
int64_t divmod_s64s64 (int64_t a,
        int64_t b)
```

Integer divide of two signed 64-bit values.

Parameters

- `a` Dividend
- `b` Divisor

Returns

- quotient in result (r0,r1), remainder in regs (r2, r3)

3.59.4.14. divmod_s64s64_rem

```
int64_t divmod_s64s64_rem (int64_t a,
       int64_t b,
       int64_t *rem)
```

Integer divide of two signed 64-bit values, with remainder.

Parameters

- `a` Dividend
- `b` Divisor
- `rem` The remainder of dividend/divisor

Returns

- Quotient result of dividend/divisor

3.59.4.15. divmod_s64s64_rem_unsafe

```
int64_t divmod_s64s64_rem_unsafe (int64_t a,
       int64_t b,
       int64_t *rem)
```

Unsafe integer divide of two signed 64-bit values, with remainder.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor
- `rem` The remainder of dividend/divisor

Returns

- Quotient result of dividend/divisor

3.59.4.16. divmod_s64s64_unsafe

```
int64_t divmod_s64s64_unsafe (int64_t a,
       int64_t b)
```

Unsafe integer divide of two signed 64-bit values.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor

Returns

- quotient in result (r0,r1), remainder in regs (r2, r3)

3.59.4.17. divmod_u32u32

```
divmod_result_t divmod_u32u32 (uint32_t a,
       uint32_t b)
```

Integer divide of two unsigned 32-bit values.

Parameters

- a Dividend
- b Divisor

Returns

- quotient in low word/r0, remainder in high word/r1

3.59.4.18. divmod_u32u32_rem

```
static uint32_t divmod_u32u32_rem (uint32_t a,
        uint32_t b,
        uint32_t *rem)
```

Integer divide of two unsigned 32-bit values, with remainder.

Parameters

- a Dividend
- b Divisor
- rem The remainder of dividend/divisor

Returns

- Quotient result of dividend/divisor

3.59.4.19. divmod_u32u32_rem_unsafe

```
uint32_t divmod_u32u32_rem_unsafe (uint32_t a,
        uint32_t b,
        uint32_t *rem)
```

Unsafe integer divide of two unsigned 32-bit values, with remainder.

Do not use in interrupts

Parameters

- a Dividend
- b Divisor
- rem The remainder of dividend/divisor

Returns

- Quotient result of dividend/divisor

3.59.4.20. divmod_u32u32_unsafe

```
uint64_t divmod_u32u32_unsafe (uint32_t a,
        uint32_t b)
```

Unsafe integer divide of two unsigned 32-bit values.

Do not use in interrupts

Parameters

- a Dividend
- b Divisor

Returns

- quotient in low word/r0, remainder in high word/r1

3.59.4.21. divmod_u64u64

```
uint64_t divmod_u64u64 (uint64_t a,
       uint64_t b)
```

Integer divide of two signed 64-bit values.

Parameters

- a Dividend
- b Divisor

Returns

- quotient in result (r0,r1), remainder in regs (r2, r3)

3.59.4.22. divmod_u64u64_rem

```
uint64_t divmod_u64u64_rem (uint64_t a,
       uint64_t b,
       uint64_t *rem)
```

Integer divide of two unsigned 64-bit values, with remainder.

Parameters

- a Dividend
- b Divisor
- rem The remainder of dividend/divisor

Returns

- Quotient result of dividend/divisor

3.59.4.23. divmod_u64u64_rem_unsafe

```
uint64_t divmod_u64u64_rem_unsafe (uint64_t a,
       uint64_t b,
       uint64_t *rem)
```

Unsafe integer divide of two unsigned 64-bit values, with remainder.

Do not use in interrupts

Parameters

- a Dividend
- b Divisor
- rem The remainder of dividend/divisor

Returns

- Quotient result of dividend/divisor

3.59.4.24. divmod_u64u64_unsafe

```
uint64_t divmod_u64u64_unsafe (uint64_t a,
       uint64_t b)
```

Unsafe integer divide of two signed 64-bit values.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor

Returns

- quotient in result (r0,r1), remainder in regs (r2, r3)

3.60. pico_double

Optimized double-precision floating point functions including replacement implementations of the compiler built-ins. More…

3.60.1. Detailed Description

Optimized double-precision floating point functions including replacement implementations of the compiler built-ins.

3.61. pico_float

Optimized single-precision floating point functions including replacement implementations of the compiler built-ins. More…

3.61.1. Detailed Description

Optimized single-precision floating point functions including replacement implementations of the compiler built-ins.

3.62. pico_int64

Optimized replacement implementations of the compiler built-in 64 bit multiplication. More…

3.62.1. Detailed Description

Optimized replacement implementations of the compiler built-in 64 bit multiplication.

This library does not provide any additional functions

3.63. pico_malloc

Multi-core safety for malloc, calloc and free. More…

3.63.1. Detailed Description

Multi-core safety for malloc, calloc and free.

This library does not provide any additional functions

3.64. pico_memory

Optimized replacement implementations of the compiler built-in memcpy and memset. More…

3.64.1. Detailed Description

Optimized replacement implementations of the compiler built-in memcpy and memset.

This library does not provide any additional functions TO DO: expose memcpy4 etc

3.65. pico_platform

Compiler definitions for the selected PICO_PLATFORM. More…

3.65.1. Detailed Description

Compiler definitions for the selected PICO_PLATFORM.

3.66. pico_runtime

Aggregate runtime support including pico_bit_ops, pico_divider, pico_double, pico_int64, pico_float, pico_malloc, pico_memory and pico_standard_link. More…

3.66.1. Detailed Description

Aggregate runtime support including pico_bit_ops, pico_divider, pico_double, pico_int64, pico_float, pico_malloc, pico_memory and pico_standard_link.

3.67. pico_standard_link

3.68. Miscellaneous

3.68.1. Modules

- `boot_picoboot`
  Header file for the PICOBOOT USB interface exposed by an RP2040 in boot mode.
- `boot_uf2`
  Header file for the UF2 format supported by an RP2040 in boot mode.

3.69. boot_picoboot

Header file for the PICOBOOT USB interface exposed by an RP2040 in boot mode. More…

3.69.1. Detailed Description

Header file for the PICOBOOT USB interface exposed by an RP2040 in boot mode.

3.70. boot_uf2

Header file for the UF2 format supported by an RP2040 in boot mode. More…

3.70.1. Detailed Description

Header file for the UF2 format supported by an RP2040 in boot mode.

Chapter 4. Using the PIO (Programmable IO)

4.1. What is Programmable I/O (PIO)?

Programmable IO (PIO) is a unique todo:? feature of the RP2040. It allows you to create new types of (or additional) hardware interfaces on your RP2040 based device. (e.g. I want to add 4 more UARTs, or I'd like to output DPI video)

4.1.1. Background

Interfacing with other digital hardware components is hard. It often happens at very high frequencies (due to amounts of data that need to be transferred), and has very exact timing requirements.

Traditionally on your desktop computer you have one option for hardware interfacing:

4.1.2. I/O Using dedicated hardware on your PC

This is a no brainer; your computer has high speed USB ports, HDMI outputs SD card slots, SATA drive controllers etc. to take care of the tricky and time sensitive business of sending and receiving ones and zeros and responding with minimal latency or interruption to the device, SD card, hard driver etc. on the other end of the hardware interface.

The custom hardware components take care of specific tasks that the more general multi-tasking CPU is not designed for. The operating system drivers perform higher level management of what the hardware components do, and coordinate data transfers via DMA to/from memory from the controller and receive IRQs when high level tasks need attention.

4.1.3. I/O Using dedicated hardware on your Raspberry Pi or microcontroller

Not so common on PCs, your Raspberry Pi or microcontroller is likely to have dedicated hardware on chip for managing UART, I2C, SPI, PWM todo more over GPIO pins. Like USB controllers (which many microcontroller including the Raspberry Pi Pico have), I2C and SPI are general purpose connection protocols allowing connection to a wide variety of different types of external hardware using the same piece of on chip hardware.

This is all very well, but the area taken up by these individual components and the associate cost, often leaves you with a limited menu, or you end up paying for a bunch of stuff you don't need, and find yourself without enough of what you really want, and of course you are out of luck if your microcontroller does not have dedicated hardware for the type of hardware device you want to attach (although in some cases you may be able to bridge over USB, I2C or SPI at the cost of buying external hardware).

4.1.4. I/O Using software control of GPIOs (*"bit-banging"*)

The third option on your Raspberry Pi or microcontroller - i.e. something with GPIOs - is to use the CPU to wiggle (and listen to) the GPIOs at dizzyingly high speeds, and hope to do so with sufficiently correct timing that the external hardware still understands the signals.

As a bit of background it is worth thinking about types of hardware that you might want to interface:

todo if this is a useful table, then verify and flesh it out a bit

Table 6. Types of hardware

Interface Speed	Interface
1-1000Hz	Push buttons, temperature sensors
?-?	UART
22-100+ kHz	PCM audio
300+ kHz	PWM audio
400-1200 kHz	WS2812 LED string
10-4000 MHz	LAN
20-1000 MHz	DPI/VGA/HDMI/DVI video
12-4000 MHz	SD card
48-40000 MHz	USB

"Bit-Banging" (i.e using the processor to hammer out the protocol via the GPIOs) is very hard. The processor isn't really designed for this. It has other work to do…. for slower protocols you might be able to use an IRQ to wake up the processor from what it was doing fast enough (though latency here is a concern) to send the next bit(s). Indeed back in the early days of PC sound it was not uncommon to set a hardware timer interrupt at 11kHz and write out one 8 bit PCM sample every interrupt for some rather primitive sounding audio!

Doing that on a PC nowadays is laughed at, even though they are many order of magnitudes faster than they were pack then.

The alternative when *"bit-banging"* is to sit the processor in a carefully timed loop, written exactly in assembly, trying to make sure the GPIO reading and writing happens on the exact cycle required. This is really really hard work if indeed possible at all. Many heroic hours and likely hundreds of github repositories are dedicated to the task of doing such things (perhaps for LED strings). Additionally of course, your processor is now busy doing the *"bit-banging"*, and cannot be used for other tasks.

Whilst dealing with something like an LED string is possible using *"bit-banging"*, once your hardware protocol gets faster to the point that it is of similar order of magnitude to your system clock speed, there is really nothing you can hope to do.

Therefore you're back to custom hardware for the protocols you know up front you are going to want (or more accurately, the chip designer thinks you might need).

4.1.5. Programmable I/O Hardware using PIO

The unique PIO subsystem of the RP2040 allows you to write small simple programs for what are called *PIO state machines* (of which the RP2040 has eight split across two PIO *instances*) which are each responsible for setting and reading bits of on one or more GPIOs, buffering data to or from the processor (or RP2040's ultra-fast DMA subsystem), and notifying the processor (via IRQ or polling) when more data or attention is needed.

These programs can perform operations with cycle accuracy at up to system clock speed (or the program clock's can be divided down to run at slower speeds for less frisky protocols).

For simple hardware protocols - such as PWM or duplex SPI - a single PIO state machine can handle the task of implementing the hardware interface all on its own, or for more involved protocols such as SDIO or DPI video you may end up using two or three.

TIP

If you are ever tempted to *"bit-bang"* a protocol on the RP2040, don't! Use the PIO instead. Frankly this is true for anything that repeatedly reads or writes from GPIOs, but certainly anything which aims to transfer data.

4.2. Getting started with PIO

It is possible to write PIO programs with both within the C/C++ SDK and directly from MicroPython.

Additionally the intention is for both to have simple APIs to trivially have new UARTs, PWM channels etc created for you, picking from a menu of pre-written PIO programs, but for now you'll have to follow along with example code and do that yourself.

TO DO: a mini handbook for the PIO assembly language. talk through the language constructs, show a couple of simple examples

4.3. Something simpler than WS2812 TO DO: Working title!

TO DO: Whilst WS2812 is actually only a few instructions in PIO assembly, it is probably not the simplest first program, so pre-amble with something else first

4.4. Using PIOASM the PIO Assembler

4.5. Managing WS2812 LEDs

The WS2812 LED, or NeoPixel, is…

TO DO: Write introduction to this, and tidy it all up. Direct dump from Graham's email at this time

TO DO: This is an example "app note" for PIO but probably not the **first** app note we want to show

```
.program ws2812
.side_set 1

.define public T1 2
.define public T2 5
.define public T3 3

.wrap_target
bitloop:
    out x, 1       [T3 - 1] set 0 ; Side-set still takes place when instruction stalls
    jmp !x do_zero [T1 - 1] set 1 ; Branch on the bit we shifted out. Positive pulse
do_one:
    jmp  bitloop   [T2 - 1] set 1 ; Continue driving high, for a long pulse
do_zero:
    nop            [T2 - 1] set 0 ; Or drive low, for a short pulse
.wrap

% c-sdk {
#include "hardware/clocks.h"

static inline void ws2812_program_init(PIO pio, uint sm, uint offset, uint pin, float freq,
  bool rgbw) {

    pio_gpio_select(pio, pin);
    pio_set_consecutive_pindirs(pio, sm, pin, 1, true);

    pio_sm_config c = ws2812_program_default_config(offset);
    sm_config_sideset_pins(&c, pin);
    sm_config_out_shift(&c, false, true, rgbw ? 32 : 24);

    int cycles_per_bit = ws2812_T1 + ws2812_T2 + ws2812_T3;
    float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit);
    sm_config_clkdiv(&c, div);

    pio_sm_init(pio, sm, offset, &c);
    pio_sm_enable(pio, sm, true);
}
%}
```

Starting to disect this snippet. We are defining a program named ws2812

```
.program ws2812
```

Each instruction in the PIO is 16 bits wide. 5 of those are used for the “delay” which is usually 0 to 31 cycles (after the instruction completes before moving to the next up instruction).. (FYI the instruction delay appears as the value in [] to the right of the instruction in the assembly)

```
.side_set 1
```

This directive `.side_set 1` says we’re stealing one of those bits to use for "side set" which allows us to assert pin values with each instruction (in addition to what the instructions are themselves doing).This is very useful for high frequency use cases (e.g. pixel clocks for DPI panels), but also for. Shrinking program size (each PIO only has a 32 instruction memory - shared between 4 SMs).

Note that stealing one bit has left our delay range from 0-15, but that is quite natural because you rarely want to mix side set with lower frequency stuff. Because we didn’t say `.side_set 1` opt which mens the side set is optional (at the cost of another delay bit to say whether the instruction does a side set) we now have to specify a side set value for each instruction in the program (this is the `set N` after the delay)

```
.define public T1 2
.define public T2 5
.define public T3 3
```

`.define` lets you declare constants. The public modifier means that the code generator will spit out a `#define` for it, in the Pico SDK generator case.

```
.wrap_target
```

We’ll ignore this for now, and come back to it later.

```
bitloop:
```

This is a label, while,

```
    out x, 1       [T3 - 1] set 0 ; Side-set still takes place when instruction stalls
```

- `out` takes bits from the OSR (output shift register) which contains data coming from the CPU (or DMA) side. Basically this is the data we are trying to write to the LED string. We are dealing with 1 bit (the ", 1”)… this takes a 1 bit value from the OSR and shifts the remainder. Whether it takes the top bit(s) SHIFT_TO_LEFT or the bottom bits SHIFT_TO_RIGHT is configurable on the PIO state machine (SM).
- `x` (one of two scratch registers; the other imaginatively called y) is the target
- `[T3-1]` is the delay we talked about (T3 minus 1 cycles), set 0 its the side set

So, we can read the instruction as:

1. Set 0 on the side set pin (since side set happens from the beginning of the instruction)
2. Shift one bit out of the OSR into the x register. Result is that x register is 0 or 1
3. Wait `T3 - 1` cycles after the instruction (I.e. the whole thing takes `T3` cycles since the instruction itself took a cycle). Note also here we're talking bout cycles from the point of view of the state machine. That is confutable to a user defined frequency by (fractional) clk divider from the system clock.

```
    jmp !x do_zero [T1 - 1] set 1 ; Branch on the bit we shifted out. Positive pulse
```

1. `set 1` on the side set pin (this is the leading edge of our pulse)
2. if `x == 0` then the next instruction will be at table do_zero, otherwise it will be the next instruction
3. we delay `T1 - 1` after the instruction (whether the branch is taken or not)

Ok, at this point we have output

```
      ----------
-----|
  T3       T1
```

```
do_one:
    jmp  bitloop   [T2 - 1] set 1 ; Continue driving high, for a long pulse
```

In this branch we do

1. `set 1` on the side set pin (this is the leading edge of our pulse)
2. `jmp` unconditionally to bitloop
3. we delay `T1 - 1` after the instruction, so we have

```
      ----------.------
-----|
  T3      T1         T2
```

4. we end up back where we started

```
do_zero:
    nop            [T2 - 1] set 0 ; Or drive low, for a short pulse
```

1. `set 0` on the side set pin (this is the leading edge of our pulse)
2. twiddle our thumbs for a cycle.
3. we delay `T1 - 1` after the instruction, so we have

```
      ----------
-----|          |------
  T3     T1        T2
```

Why did we twiddle our thumbs when we could have `jmp`-ed, good question, actually in this case there is no good reason, however it does show you .wrap and .wrap_target

`.wrap` → `.wrap_target` is basically a free jmp (which often is important to either save the instruction space of the jump, or the 1 cycle cost)). In this case we just fall off the end of the program and start back at the beginning.

Anway, now it should be clear why our timings are numbered this way, because what the LED string sees really is

```
 ----------.------
|                 |-----
    T1        T2    T3
```

For a one, and

```
 ----------
|          |------.-----
    T1        T2.   T3
```

For a zero

I say sees as it is looking for the start of a pulse and the wire is at 0 until we start sending bits (which are encoded as long pulses for 1 and short pulses for zero

So it looks like we just do this indefinitely, but where is the data coming from

This is more fully explained in the **RP2040 Datasheet**, but the data that goes into the OSR comes from the SM's write FIFO (which is filled by directly by poking from the CPU, or via DMA)

The `OUT` instruction by default will shift zeros into the other end of the (32 bit) OSR as it shifts data out, so you'll end up getting zeros for sure after 32 bits. There is a `PULL` instruction to explicitly take data from the FIFO and put it in the OSR (it also blocks if the FIFO is empty)

However, in the majority of cases it is simpler to configured the SM for auto-pull (in which case the OSR is automatically refilled (PULLed) when a configured number of bits have been shifted out of the ISR) You'll see this configured number of bits in the switch I had between 24 and 32 for (BGR or WGBR for number of data bits in 3 color or 4 color LEDs)

When we run `pioasm` and ask it to spit out Pico SDK code (which is the default), it will create some static variables describing the program, and a method ws2812_default_program_config which configures a PIO SM based on the information in the actual .pio file (namely the .side_set in this case).

```
% c-sdk {
```

Of course how you configure the PIO SM when using the program is very much related to the program you have written. Rather than try to store a data representation off all that information, and parse it at runtime, for the use cases where you'd like to encapsulate setup or other API functions with your Pio program, you can embed code withing the .PIO file

In this case we are passing thru code for the Pico SDK (it will end up in the generated header file)

We have here a function `ws2812_program_init` which is provided to help the user to instantiate an instance of the LED driver program

PIO
: (which of the two PIOs we are dealing with)

SM
: which state machine on that pio

Offset
: where the PIO program was loaded in the 5 bit address space

Pin
: which GPIO pin we'd like to output on

Freq
: the bit frequency to output at

Rgbw
: true if we have 4 color not 3

Such that,

- `pio_gpio_select(pio, pin);` Configures the func sea for the pin to the right PIO
- `pio_sm_config c = ws2812_program_default_config(offset);` Get the default configuration using the generated function for this program
- `pio_set_consecutive_pindirs(pio, sm, pin, 1, true);` Sets the PIO pin direction of 1 pin starting at pin number "pin" to out
- `sm_config_sideset_pins(&c, pin);` Sets the side set to set values starting at pin "pin" (I say starting at because if you had .side_set 3, then it would be outputting values on numbers pin, pin+1, pin+2)
- `sm_config_out_shift(&c, false, true, rgbw ? 32 : 24);` False for shift_to_right (i.e. we want to shift out MSB first) True for auto-pull 32 or 24 for the number of bits to shift before we refill the OSR
- `int cycles_per_bit = ws2812_T1 + ws2812_T2 + ws2812_T3;` Here we see the benefit of .define public; we can use the T1 - T3 values in our code

This is the total number of PIO SM. Cycles to output a single bit

- `float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit); sm_config_clkdiv(&c, div);` Using that we can configure how much to divide the system clock by
- `pio_sm_init(pio, sm, offset, &c);` Now initialize the SM using the configuration we have built
- `pio_sm_enable(pio, sm, true);`

And make it go now!

At this point the program will be stuck on OUT waiting for data (the OUT becomes blocking because the OSR was empty, and with auto-pull it effectively did a PULL which blocked because the FIFO is empty)… note I misspoke slightly earlier when I said you are configuring how many bits are shifted out before re-filling. You are configuring what to set the "count of valid bits" value for the OSR to when it is PULLed. i.e. the OSR is empty when the count of valid bits decrements to zero ℹ NOTE

If you pokes 32 bit values one at time (one per pixel) directly to the SM FIFO

`pio_put_blocking(pio0, 0, pixel_grb << 8);` You'll notice the << 8 remember we were shifting out starting with the MSB, so we want the 24 bit color values at the top… this works fine for WGBR too, just that the W is always 0)

`pio_put_blocking` is a helper method that waits until there is room in the FIFO before trying to write another value

Alternative you can make a DMA transfer that reads sequential words from memory and writes each of them to the same address (the SM FIFO address). Fortunately the PIO SM provides a DREQ signal that can be configured onto your transfer, so that data flows at the correct rate

This is all very snazzy and efficient, and the PIO SM can actually consume data thru the FIFO via DMA at up to 32 bits every 2 system clocks cycles (although we don't have anything - including video- that goes quite that fast, although of course this is shared bandwidth).

TODO: show code to send to start the PIO transferring a whole string of pixels without further CPU intervention via DMA

Appendix A: App Notes

Attaching a 7 segment LED via GPIO

This example code shows how to interface the Raspberry Pi Pico to a generic 7 segment LED device. It uses the LED to count from 0 to 9 and then repeat. If the button is pressed, then the numbers will count down instead of up.

Wiring information

Our 7 Segment display has pins as follows.

```
--A--
F   B
--G--
E   C
--D--
```

By default we are allocating GPIO 2 to A, 3 to B etc. So, connect GPIO 2 to pin A on the 7 segment LED display and so on. You will need the appropriate resistors (68 ohm should be fine) for each segment. The LED device used here is common anode, so the anode pin is connected to the 3.3v supply, and the GPIO's need to pull low (to ground) to complete the circuit. The pull direction of the GPIO's is specified in the code itself.

Connect the switch to connect on pressing. One side should be connected to ground, the other to GPIO 9.

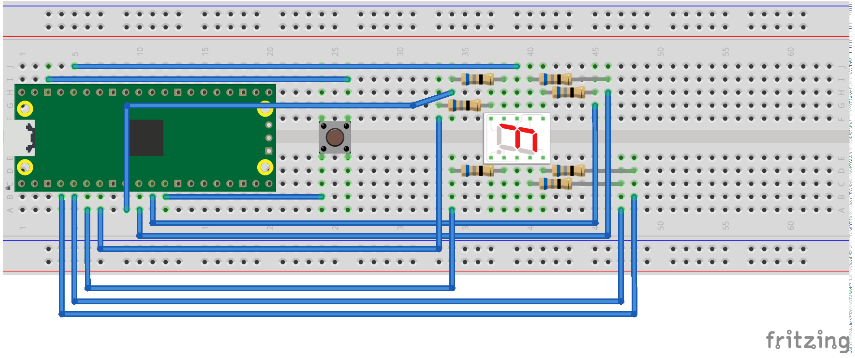


Figure 1. Wiring Diagram for 7 segment LED.

List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/pre_release/gpio/hello_7segment/CMakeLists.txt Lines 1 - 9

```
add_executable(hello_7segment
        hello_7segment.c
        )

# Pull in our pico_stdlib which pulls in commonly used features
target_link_libraries(hello_7segment pico_stdlib)

# create map/bin/hex file etc.
```

```
pico_add_extra_outputs(hello_7segment)
```

hello_7segment.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/pre_release/gpio/hello_7segment/hello_7segment.c Lines 1 - 95

```
/**
 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */

#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/gpio.h"

/*
  Our 7 Segment display has pins as follows:

  --A--
  F   B
  --G--
  E   C
  --D--

  By default we are allocating GPIO 2 to A, 3 to B etc.
  So, connect GOIP 2 to pin A on the 7 segment LED display etc. Don't forget
  the appropriate resistors, best to use one for each segment!

  Connect button so that pressing the switch connects the GPIO 9 (default) to
  ground (pull down)
*/

#define FIRST_GPIO 2
#define BUTTON_GPIO (FIRST_GPIO+7)

// This array converts a number 0-9 to a bit pattern to send to the GPIO's
int bits[10] = {
        0x3f,  // 0
        0x06,  // 1
        0x5b,  // 2
        0x4f,  // 3
        0x66,  // 4
        0x6d,  // 5
        0x7d,  // 6
        0x07,  // 7
        0x7f,  // 8
        0x67   // 9
};

// tag::hello_gpio[]
int main() {
    stdio_init_all();
    printf("Hello, 7segment - press button to count down!\n");

    // We could use gpio_set_dir_out_masked() here
    for (int gpio = FIRST_GPIO; gpio < FIRST_GPIO + 7; gpio++) {
        gpio_init(gpio);
        gpio_set_dir(gpio, GPIO_OUT);
        // Our bitmap above has a bit set where we need an LED on, BUT, we are pulling low to
light
        // so invert our output
        gpio_set_outover(gpio, GPIO_OVERRIDE_INVERT);
    }

    gpio_init(BUTTON_GPIO);
    gpio_set_dir(BUTTON_GPIO, GPIO_IN);
    // We are using the button to pull down to 0v when pressed, so ensure that when
    // unpressed, it uses internal pull ups. Otherwise when unpressed, the input will
    // be floating.
    gpio_pull_up(BUTTON_GPIO);

    int val = 0;
    while (true) {
        // Count upwards or downwards depending on button input
        // We are pulling down on switch active, so invert the get to make
        // a press count downwards
        if (!gpio_get(BUTTON_GPIO)) {
            if (val == 9) {
                val = 0;
            } else {
                val++;
            }
        } else if (val == 0) {
            val = 9;
        } else {
            val--;
        }

        // We are starting with GPIO 2, our bitmap starts at bit 0 so shift to start at 2.
        int32_t mask = bits[val] << FIRST_GPIO;

        // Set all our GPIO's in one go!
        // If something else is using GPIO, we might want to use gpio_put_masked()
        gpio_set_mask(mask);
        sleep_ms(250);
        gpio_clr_mask(mask);
    }

    return 0;
}
// end::hello_gpio[]
```

Bill of Materials

Table 7. A list of materials required for the example

Item	**Quantity**	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	http://raspberrypi.org/
7 segment LED module	1	generic part
68 ohm resistor	7	generic part
DIL push to make switch	1	generic switch
M/M Jumper wires	10	generic part

DHT-11, DHT-22, and AM2302 Sensors

The DHT sensors are fairly well known hobbyist sensors for measuring relative humidity and temperature using a capacitive humidity sensor, and a thermistor. While they are slow, one reading every ~2 seconds, they are reliable and good for basic data logging. Communication is based on a custom protocol which uses a single wire for data.

NOTE

> The DHT-11 and DHT-22 sensors are the most common. They use the same protocol but have different characteristics, the DHT-22 has better accuracy, and has a larger sensor range than the DHT-11. The sensor is available from a number of retailers.

Wiring information

See Figure 2 for wiring instructions.

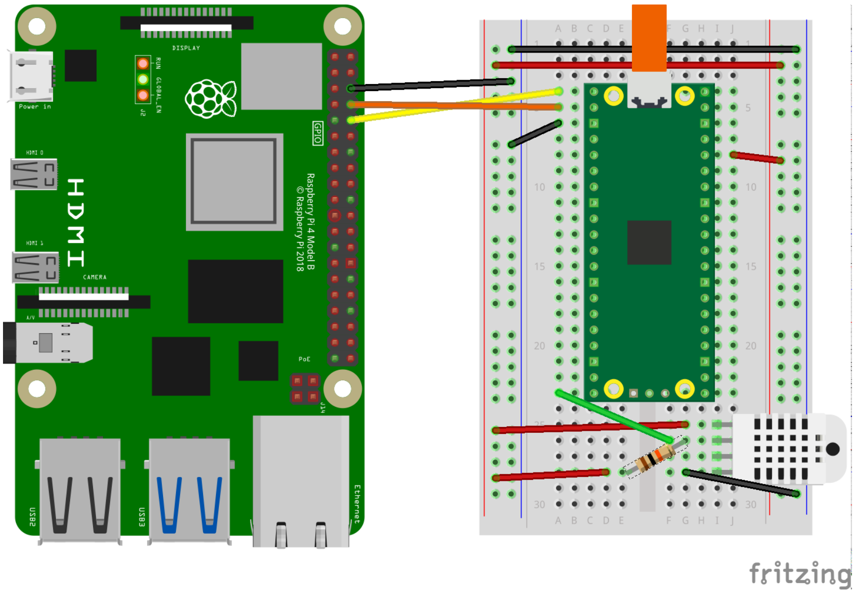


Figure 2. Wiring the DHT-22 temperature sensor to Raspberry Pi Pico, and connecting Pico's UART0 to the Raspberry Pi 4.

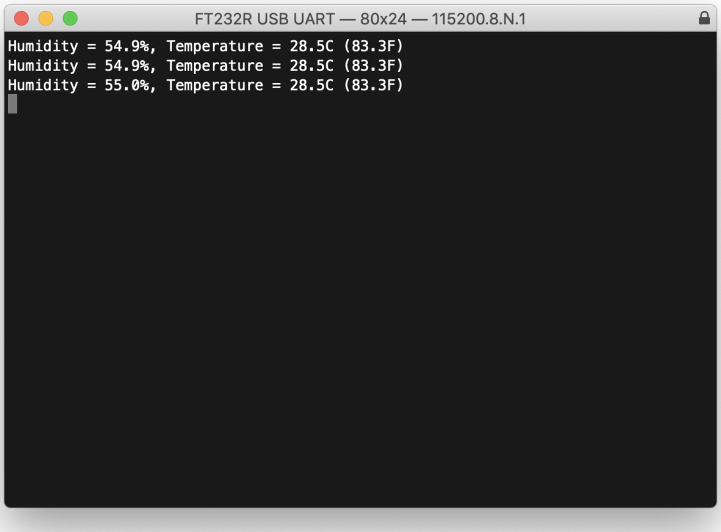
NOTE

> One of the pins (pin 3) on the DHT sensor will not be connected, it is not used.

You will want to place a 10 kΩ resistor between VCC and the data pin, to act as a medium-strength pull up on the data line.

Connecting UART0 of Pico to Raspberry Pi as in Figure 2 and you should see something similar to Figure 3 in `minicom` when connected to `/dev/serial0` on the Raspberry Pi.

Figure 3. Serial output over Pico's UART0 in a terminal window.



```
Humidity = 54.9%, Temperature = 28.5C (83.3F)
Humidity = 54.9%, Temperature = 28.5C (83.3F)
Humidity = 55.0%, Temperature = 28.5C (83.3F)
```

Connect to `/dev/serial0` by typing,

```
$ minicom -b 115200 -o -D /dev/serial0
```

at the command line.

List of Files

A list of files with descriptions of their function;

CMakeLists.txt

Make file to incorporate the example in to the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/pre_release/gpio/dht_sensor/CMakeLists.txt Lines 1 - 7

```
add_executable(dht
        dht.c
        )

target_link_libraries(dht pico_stdlib)

pico_add_extra_outputs(dht)
```

dht.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/pre_release/gpio/dht_sensor/dht.c Lines 1 - 83

```
/**
 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 **/

```

```c
#include <stdio.h>
#include <math.h>
#include "pico/stdlib.h"
#include "hardware/gpio.h"

const uint LED_PIN = PICO_DEFAULT_LED_PIN;
const uint DHT_PIN = 15;
const uint MAX_TIMINGS = 85;

typedef struct {
    float humidity;
    float temp_celsius;
} dht_reading;

void read_from_dht(dht_reading *result);

int main() {
    stdio_init_all();
    gpio_init(LED_PIN);
    gpio_init(DHT_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);
    while (1) {
        dht_reading reading;
        read_from_dht(&reading);
        float fahrenheit = (reading.temp_celsius * 9 / 5) + 32;
        printf("Humidity = %.1f%%, Temperature = %.1fC (%.1fF)\n",
               reading.humidity, reading.temp_celsius, fahrenheit);

        sleep_ms(2000);
    }
}

void read_from_dht(dht_reading *result) {
    int data[5] = {0, 0, 0, 0, 0};
    uint last = 1;
    uint j = 0;

    gpio_set_dir(DHT_PIN, GPIO_OUT);
    gpio_put(DHT_PIN, 0);
    sleep_ms(20);
    gpio_set_dir(DHT_PIN, GPIO_IN);

    gpio_put(LED_PIN, 1);
    for (uint i = 0; i < MAX_TIMINGS; i++) {
        uint count = 0;
        while (gpio_get(DHT_PIN) == last) {
            count++;
            sleep_us(1);
            if (count == 255) break;
        }
        last = gpio_get(DHT_PIN);
        if (count == 255) break;

        if ((i >= 4) && (i % 2 == 0)) {
            data[j / 8] <<= 1;
            if (count > 16) data[j / 8] |= 1;
            j++;
        }
    }
    gpio_put(LED_PIN, 0);

    if ((j >= 40) && (data[4] == ((data[0] + data[1] + data[2] + data[3]) & 0xFF))) {
        result->humidity = (float) ((data[0] << 8) + data[1]) / 10;
        if (result->humidity > 100) {
            result->humidity = data[0];
        }
        result->temp_celsius = (float) (((data[2] & 0x7F) << 8) + data[3]) / 10;
        if (result->temp_celsius > 125) {
            result->temp_celsius = data[2];
        }
        if (data[2] & 0x80) {
            result->temp_celsius = -result->temp_celsius;
        }
    } else {
        printf("Bad data\n");
    }
}
```

Bill of Materials

Table 8. A list of materials required for the example

Item	**Quantity**	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	http://raspberrypi.org/
10 kΩ resistor	1	generic part
M/M Jumper wires	4	generic part
DHT-22 sensor	1	generic part

Attaching a BME280 temperature/humidity/pressure sensor via SPI

This example code shows how to interface the Raspberry Pi Pico to a BME280 temperature/humidity/pressure. The particular device used can be interfaced via I2C or SPI, we are using SPI, and interfacing at 3.3v.

This examples reads the data from the sensor, and runs it through the appropraite conpensation routines (see the chip datasheet for details https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme280-ds002.pdf). At startup the compensation parameters required by the compensation routines are read from the chip.)

Wiring information

Wiring up the device requires 6 jumpers as follows:

- GPIO 16 (pin 21) MISO/spi0_rx→ SDO/SDO on bme280 board
- GPIO 17 (pin 22) Chip select → CSB/!CS on bme280 board
- GPIO 18 (pin 24) SCK/spi0_sclk → SCL/SCK on bme280 board
- GPIO 19 (pin 25) MOSI/spi0_tx → SDA/SDI on bme280 board
- 3.3v (pin 3;6) → VCC on bme280 board
- GND (pin 38) → GND on bme280 board

The example here uses SPI port 0. Power is supplied from the 3.3V pin.

NOTE

There are many different manufacturers who sell boards with the BME280. Whilst they all appear slightly different, they all have, at least, the same 6 pins required to power and communicate. When wiring up a board that is different to the one in the diagram, ensure you connect up as described in the previous paragraph.

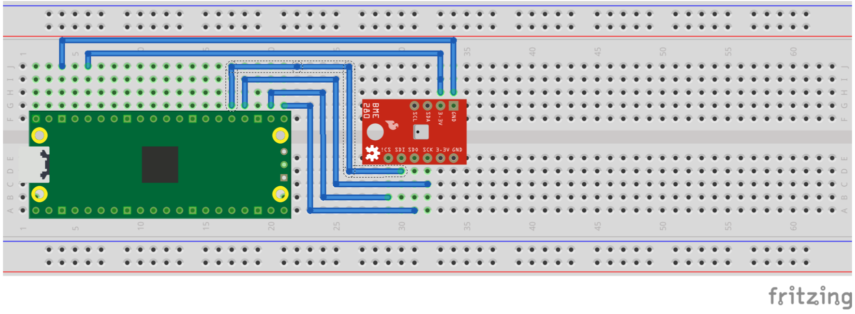


Figure 4. Wiring Diagram for bme280.

List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/pre_release/spi/bme280_spi/CMakeLists.txt Lines 1 - 9

```
add_executable(bme280_spi
        bme280_spi.c
        )

# Pull in our (to be renamed) simple get you started dependencies
target_link_libraries(bme280_spi pico_stdlib hardware_spi)

# create map/bin/hex file etc.
pico_add_extra_outputs(bme280_spi)
```

bme280_spi.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/pre_release/spi/bme280_spi/bme280_spi.c Lines 1 - 233

```
/**
 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */

#include <stdio.h>
#include <string.h>
#include "pico/stdlib.h"
#include "hardware/spi.h"

/* Example code to talk to a bme280 humidity/temperature/pressure sensor.

   NOTE: Ensure the device is capable of being driven at 3.3v NOT 5v. The Pico
   GPIO (and therefor SPI) cannot be used at 5v.

   You will need to use a level shifter on the SPI lines if you want to run the
   board at 5v.

   Connections on Raspberry Pi Pico board and a generic bme280 board, other
   boards may vary.

   GPIO 16 (pin 21) MISO/spi0_rx-> SDO/SDO on bme280 board
   GPIO 17 (pin 22) Chip select -> CSB/!CS on bme280 board
   GPIO 18 (pin 24) SCK/spi0_sclk -> SCL/SCK on bme280 board
   GPIO 19 (pin 25) MOSI/spi0_tx -> SDA/SDI on bme280 board
   3.3v (pin 3;6) -> VCC on bme280 board
   GND (pin 38)  -> GND on bme280 board

   Note: SPI devices can have a number of different naming schemes for pins. See
   the Wikipedia page at https://en.wikipedia.org/wiki/Serial_Peripheral_Interface
   for variations.

   This code uses a bunch of register definitions, and some compensation code derived
   from the Bosch datasheet which can be found here.
   https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme280-ds002.pdf
*/

#define PIN_MISO 16
#define PIN_CS   17
#define PIN_SCK  18
#define PIN_MOSI 19

#define SPI_PORT spi0
#define READ_BIT 0x80

int32_t t_fine;

uint16_t dig_T1;
int16_t dig_T2, dig_T3;
uint16_t dig_P1;
int16_t dig_P2, dig_P3, dig_P4, dig_P5, dig_P6, dig_P7, dig_P8, dig_P9;
uint8_t dig_H1, dig_H3;
int8_t dig_H6;
int16_t dig_H2, dig_H4, dig_H5;

/* The following compensation functions are required to convert from the raw ADC
data from the chip to something usable. Each chip has a different set of
compensation parameters stored on the chip at point of manufacture, which are
read from the chip at startup and used inthese routines.
*/
int32_t compensate_temp(int32_t adc_T) {
    int32_t var1, var2, T;
    var1 = ((((adc_T >> 3) - ((int32_t) dig_T1 << 1))) * ((int32_t) dig_T2)) >> 11;
    var2 = (((((adc_T >> 4) - ((int32_t) dig_T1)) * ((adc_T >> 4) - ((int32_t) dig_T1))) >> 12) * ((int32_t) dig_T3))
            >> 14;

    t_fine = var1 + var2;
    T = (t_fine * 5 + 128) >> 8;
    return T;
}

uint32_t compensate_pressure(int32_t adc_P) {
    int32_t var1, var2;
    uint32_t p;
    var1 = (((int32_t) t_fine) >> 1) - (int32_t) 64000;
    var2 = (((var1 >> 2) * (var1 >> 2)) >> 11) * ((int32_t) dig_P6);
    var2 = var2 + ((var1 * ((int32_t) dig_P5)) << 1);
    var2 = (var2 >> 2) + (((int32_t) dig_P4) << 16);
    var1 = (((dig_P3 * (((var1 >> 2) * (var1 >> 2)) >> 13)) >> 3) + ((((int32_t) dig_P2) * var1) >> 1)) >> 18;
    var1 = ((((32768 + var1)) * ((int32_t) dig_P1)) >> 15);
    if (var1 == 0)
        return 0;

    p = (((uint32_t) (((int32_t) 1048576) - adc_P) - (var2 >> 12))) * 3125;
    if (p < 0x80000000)
        p = (p << 1) / ((uint32_t) var1);
    else
        p = (p / (uint32_t) var1) * 2;

    var1 = (((int32_t) dig_P9) * ((int32_t) (((p >> 3) * (p >> 3)) >> 13))) >> 12;
    var2 = (((int32_t) (p >> 2)) * ((int32_t) dig_P8)) >> 13;
    p = (uint32_t) ((int32_t) p + ((var1 + var2 + dig_P7) >> 4));

    return p;
}

uint32_t compensate_humidity(int32_t adc_H) {
    int32_t v_x1_u32r;
    v_x1_u32r = (t_fine - ((int32_t) 76800));
    v_x1_u32r = (((((adc_H << 14) - (((int32_t) dig_H4) << 20) - (((int32_t) dig_H5) * v_x1_u32r)) +
                   ((int32_t) 16384)) >> 15) * (((((((v_x1_u32r * ((int32_t) dig_H6)) >> 10) * (((v_x1_u32r *
                                                                                                  ((int32_t) dig_H3))
            >> 11) + ((int32_t) 32768))) >> 10) + ((int32_t) 2097152)) *
                                                 ((int32_t) dig_H2) + 8192) >> 14));
    v_x1_u32r = (v_x1_u32r - (((((v_x1_u32r >> 15) * (v_x1_u32r >> 15)) >> 7) * ((int32_t) dig_H1)) >> 4));
    v_x1_u32r = (v_x1_u32r < 0 ? 0 : v_x1_u32r);
    v_x1_u32r = (v_x1_u32r > 419430400 ? 419430400 : v_x1_u32r);

    return (uint32_t) (v_x1_u32r >> 12);
}

static inline void cs_select() {
    asm volatile("nop \n nop \n nop");
    gpio_put(PIN_CS, 0);  // Active low
    asm volatile("nop \n nop \n nop");
}

static inline void cs_deselect() {
    asm volatile("nop \n nop \n nop");
    gpio_put(PIN_CS, 1);
    asm volatile("nop \n nop \n nop");
}

static void write_register(uint8_t reg, uint8_t data) {
    uint8_t buf[2];
    buf[0] = reg & 0x7f;  // remove read bit as this is a write
    buf[1] = data;
    cs_select();
    spi_write_blocking(SPI_PORT, buf, 2);
    cs_deselect();
    sleep_ms(10);
}

static void read_registers(uint8_t reg, uint8_t *buf, uint16_t len) {
    // For this particular device, we send the device the register we want to read
    // first, then subsequently read from the device. The register is auto incrementing
    // so we don't need to keep sending the register we want, just the first.
    reg |= READ_BIT;
    cs_select();
    spi_write_blocking(SPI_PORT, &reg, 1);
    sleep_ms(10);
    spi_read_blocking(SPI_PORT, 0, buf, len);
    cs_deselect();
    sleep_ms(10);
}

/* This function reads the manufacturing assigned compensation parameters from the device */
void read_compensation_parameters() {
    uint8_t buffer[26];

    read_registers(0x88, buffer, 24);

    dig_T1 = buffer[0] | (buffer[1] << 8);
    dig_T2 = buffer[2] | (buffer[3] << 8);
    dig_T3 = buffer[4] | (buffer[5] << 8);

    dig_P1 = buffer[6] | (buffer[7] << 8);
    dig_P2 = buffer[8] | (buffer[9] << 8);
    dig_P3 = buffer[10] | (buffer[11] << 8);
    dig_P4 = buffer[12] | (buffer[13] << 8);
    dig_P5 = buffer[14] | (buffer[15] << 8);
    dig_P6 = buffer[16] | (buffer[17] << 8);
    dig_P7 = buffer[18] | (buffer[19] << 8);
    dig_P8 = buffer[20] | (buffer[21] << 8);
    dig_P9 = buffer[22] | (buffer[23] << 8);

    dig_H1 = buffer[25];

    read_registers(0xE1, buffer, 8);

    dig_H2 = buffer[0] | (buffer[1] << 8);
    dig_H3 = (int8_t) buffer[2];
    dig_H4 = buffer[3] << 4 | (buffer[4] & 0xf);
    dig_H5 = (buffer[5] >> 4) | (buffer[6] << 4);
    dig_H6 = (int8_t) buffer[7];
}

static void bme280_read_raw(int32_t *humidity, int32_t *pressure, int32_t *temperature) {
    uint8_t buffer[8];

    read_registers(0xF7, buffer, 8);
    *pressure = ((uint32_t) buffer[0] << 12) | ((uint32_t) buffer[1] << 4) | (buffer[2] >> 4);
    *temperature = ((uint32_t) buffer[3] << 12) | ((uint32_t) buffer[4] << 4) | (buffer[5] >> 4);
    *humidity = (uint32_t) buffer[6] << 8 | buffer[7];
}

int main() {
    stdio_init_all();

    printf("Hello, bme280! Reading raw data from registers via SPI...\n");

    // This example will use SPI0 at 0.5MHz.
    spi_init(SPI_PORT, 500 * 1000);
    gpio_set_function(PIN_MISO, GPIO_FUNC_SPI);
    gpio_set_function(PIN_SCK, GPIO_FUNC_SPI);
    gpio_set_function(PIN_MOSI, GPIO_FUNC_SPI);

    // Chip select is active-low, so we'll initialise it to a driven-high state
    gpio_init(PIN_CS);
    gpio_set_dir(PIN_CS, GPIO_OUT);
    gpio_put(PIN_CS, 1);

    // See if SPI is working - interrograte the device for its I2C ID number, should be 0x60
    uint8_t id;
    read_registers(0xD0, &id, 1);
    printf("Chip ID is 0x%x\n", id);

    read_compensation_parameters();

    write_register(0xF2, 0x1); // Humidity oversampling register - going for x1
    write_register(0xF4, 0x27);// Set rest of oversampling modes and run mode to normal

    int32_t humidity, pressure, temperature;

    while (1) {
        bme280_read_raw(&humidity, &pressure, &temperature);

        // These are the raw numbers from the chip, so we need to run through the
        // compensations to get human understandable numbers
        pressure = compensate_pressure(pressure);
        temperature = compensate_temp(temperature);
        humidity = compensate_humidity(humidity);

        printf("Humidity = %.2f%%\n", humidity / 1024.0);
        printf("Pressure = %dPa\n", pressure);
        printf("Temp. = %.2fC\n", temperature / 100.0);

        sleep_ms(1000);
    }

    return 0;
}
```

Bill of Materials

Table 9. A list of materials required for the example

Item	**Quantity**	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	http://raspberrypi.org/
BME280 board	1	generic part
M/M Jumper wires	6	generic part

Attaching a MPU9250 acceleromter/gyoscope via SPI

This example code shows how to interface the Raspberry Pi Pico to the MPU9250 accelerometer/gyroscope board. The particular device used can be interfaced via I2C or SPI, we are using SPI, and interfacing at 3.3v.

NOTE

This is a very basic example, and only recovers raw data from the sensor. There are various calibration options available that should be used to ensure that the final results are accurate. It is also possible to wire up the interrupt pin to a GPIO and read data only when it is ready, rather than using the polling approach in the example.

Wiring information

Wiring up the device requires 6 jumpers as follows:

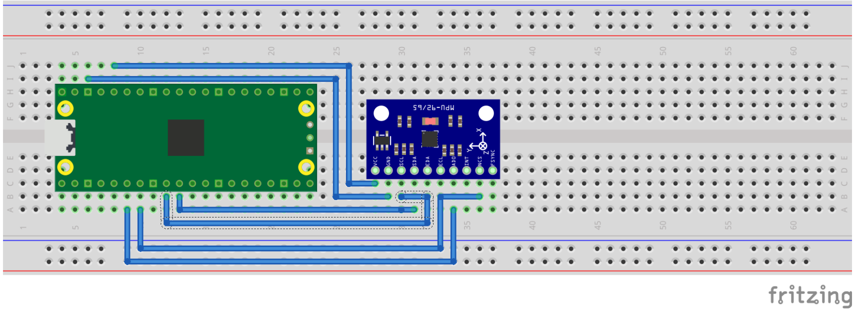
- GPIO 4 (pin 6) MISO/spi0_rx→ ADO on MPU9250 board
- GPIO 5 (pin 7) Chip select → NCS on MPU9250 board
- GPIO 6 (pin 9) SCK/spi0_sclk → SCL on MPU9250 board
- GPIO 7 (pin 10) MOSI/spi0_tx → SDA on MPU9250 board
- 3.3v (pin 36) → VCC on MPU9250 board
- GND (pin 38) → GND on MPU9250 board

The example here uses SPI port 0. Power is supplied from the 3.3V pin.

NOTE

There are many different manufacturers who sell boards with the MPU9250. Whilst they all appear slightly different, they all have, at least, the same 6 pins required to power and communicate. When wiring up a board that is different to the one in the diagram, ensure you connect up as described in the previous paragraph.

Figure 5. Wiring Diagram for MPU9250.



List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/pre_release/spi/mpu9250_spi/CMakeLists.txt Lines 1 - 9

```
add_executable(mpu9250_spi
        mpu9250_spi.c
        )

# Pull in our (to be renamed) simple get you started dependencies
target_link_libraries(mpu9250_spi pico_stdlib hardware_spi)

# create map/bin/hex file etc.
pico_add_extra_outputs(mpu9250_spi)
```

mpu9250_spi.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/pre_release/spi/mpu9250_spi/mpu9250_spi.c Lines 1 - 150

```c
/**
 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */

#include <stdio.h>
#include <string.h>
#include "pico/stdlib.h"
#include "hardware/spi.h"

/* Example code to talk to a MPU9250 MEMS accelerometer and gyroscope.
   Ignores the magnetometer, that is left as a exercise for the reader.

   This is taking to simple approach of simply reading registers. It's perfectly
   possible to link up an interrupt line and set things up to read from the
   inbuilt FIFO to make it more useful.

   NOTE: Ensure the device is capable of being driven at 3.3v NOT 5v. The Pico
   GPIO (and therefor SPI) cannot be used at 5v.

   You will need to use a level shifter on the I2C lines if you want to run the
   board at 5v.

   Connections on Raspberry Pi Pico board and a generic MPU9250 board, other
   boards may vary.

   GPIO 4 (pin 6) MISO/spi0_rx-> ADO on MPU9250 board
   GPIO 5 (pin 7) Chip select -> NCS on MPU9250 board
   GPIO 6 (pin 9) SCK/spi0_sclk -> SCL on MPU9250 board
   GPIO 7 (pin 10) MOSI/spi0_tx -> SDA on MPU9250 board
   3.3v (pin 36) -> VCC on MPU9250 board
   GND (pin 38)  -> GND on MPU9250 board

   Note: SPI devices can have a number of different naming schemes for pins. See
   the Wikipedia page at https://en.wikipedia.org/wiki/Serial_Peripheral_Interface
   for variations.
   The particular device used here uses the same pins for I2C and SPI, hence the
   using of I2C names
*/

#define PIN_MISO 4
#define PIN_CS   5
#define PIN_SCK  6
#define PIN_MOSI 7

#define SPI_PORT spi0
#define READ_BIT 0x80

static inline void cs_select() {
    asm volatile("nop \n nop \n nop");
    gpio_put(PIN_CS, 0);  // Active low
    asm volatile("nop \n nop \n nop");
}

static inline void cs_deselect() {
    asm volatile("nop \n nop \n nop");
    gpio_put(PIN_CS, 1);
    asm volatile("nop \n nop \n nop");
}

static void mpu9250_reset() {
    // Two byte reset. First byte register, second byte data
    // There are a load more options to set up the device in different ways that could be added here
    uint8_t buf[] = {0x6B, 0x00};
    cs_select();
    spi_write_blocking(SPI_PORT, buf, 2);
    cs_deselect();
}


static void read_registers(uint8_t reg, uint8_t *buf, uint16_t len) {
    // For this particular device, we send the device the register we want to read
    // first, then subsequently read from the device. The register is auto incrementing
    // so we don't need to keep sending the register we want, just the first.

    reg |= READ_BIT;
    cs_select();
    spi_write_blocking(SPI_PORT, &reg, 1);
    sleep_ms(10);
    spi_read_blocking(SPI_PORT, 0, buf, len);
    cs_deselect();
    sleep_ms(10);
}


static void mpu9250_read_raw(int16_t accel[3], int16_t gyro[3], int16_t *temp) {
    uint8_t buffer[6];

    // Start reading acceleration registers from register 0x3B for 6 bytes
    read_registers(0x3B, buffer, 6);

    for (int i = 0; i < 3; i++) {
        accel[i] = (buffer[i * 2] << 8 | buffer[(i * 2) + 1]);
    }

    // Now gyro data from reg 0x43 for 6 bytes
    read_registers(0x43, buffer, 6);

    for (int i = 0; i < 3; i++) {
        gyro[i] = (buffer[i * 2] << 8 | buffer[(i * 2) + 1]);;
    }

    // Now temperature from reg 0x41 for 2 bytes
    read_registers(0x41, buffer, 2);

    *temp = buffer[0] << 8 | buffer[1];
}

int main() {
    stdio_init_all();

    printf("Hello, MPU9250! Reading raw data from registers via SPI...\n");

    // This example will use SPI0 at 0.5MHz.
    spi_init(SPI_PORT, 500 * 1000);
    gpio_set_function(PIN_MISO, GPIO_FUNC_SPI);
    gpio_set_function(PIN_SCK, GPIO_FUNC_SPI);
    gpio_set_function(PIN_MOSI, GPIO_FUNC_SPI);

    // Chip select is active-low, so we'll initialise it to a driven-high state
    gpio_init(PIN_CS);
    gpio_set_dir(PIN_CS, GPIO_OUT);
    gpio_put(PIN_CS, 1);

    mpu9250_reset();

    // See if SPI is working - interrograte the device for its I2C ID number, should be 0x71
    uint8_t id;
    read_registers(0x75, &id, 1);
    printf("I2C address is 0x%x\n", id);

    int16_t acceleration[3], gyro[3], temp;

    while (1) {
        mpu9250_read_raw(acceleration, gyro, &temp);

        // These are the raw numbers from the chip, so will need tweaking to be really useful.
        // See the datasheet for more information
        printf("Acc. X = %d, Y = %d, Z = %d\n", acceleration[0], acceleration[1], acceleration[2]);
        printf("Gyro. X = %d, Y = %d, Z = %d\n", gyro[0], gyro[1], gyro[2]);
        // Temperature is simple so use the datasheet calculation to get deg C.
        // Note this is chip temperature.
        printf("Temp. = %f\n", (temp / 340.0) + 36.53);

        sleep_ms(100);
    }

    return 0;
}
```

Bill of Materials

Table 10. A list of materials required for the example

Item	**Quantity**	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	http://raspberrypi.org/
MPU9250 board	1	generic part
M/M Jumper wires	6	generic part

Attaching a MPU6050 acceleromter/gyoscope via I2C

This example code shows how to interface the Raspberry Pi Pico to the MPU6050 accelerometer/gyroscope board. This device uses I2C for communications, and most MPU6050 parts are happy running at either 3.3 or 5v. The Raspberry Pi RP2040 GPIO's work at 3.3v so that is what the example uses.

NOTE

This is a very basic example, and only recovers raw data from the sensor. There are various calibration options available that should be used to ensure that the final results are accurate. It is also possible to wire up the interrupt pin to a GPIO and read data only when it is ready, rather than using the polling approach in the example.

Wiring information

Wiring up the device requires 4 jumpers, to connect VCC (3.3v), GND, SDA and SCL. The example here uses I2C port 0, which is assigned to GPIO 4 (SDA) and 5 (SCL) in software. Power is supplied from the 3.3V pin.

NOTE

There are many different manufacturers who sell boards with the MPU6050. Whilst they all appear slightly different, they all have, at least, the same 4 pins required to power and communicate. When wiring up a board that is different to the one in the diagram, ensure you connect up as described in the previous paragraph.

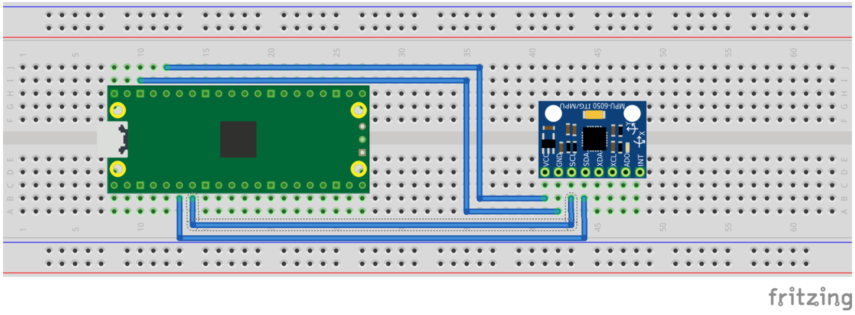


Figure 6. Wiring Diagram for MPU6050.

List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/pre_release/i2c/mpu6050_i2c/CMakeLists.txt Lines 1 - 9

```
add_executable(mpu6050_i2c
        mpu6050_i2c.c
        )

# Pull in our (to be renamed) simple get you started dependencies
target_link_libraries(mpu6050_i2c pico_stdlib hardware_i2c)

# create map/bin/hex file etc.
pico_add_extra_outputs(mpu6050_i2c)
```

mpu6050_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/pre_release/i2c/mpu6050_i2c/mpu6050_i2c.c Lines 1 - 110

```
/**
 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */

#include <stdio.h>
#include <string.h>
#include "pico/stdlib.h"
#include "hardware/i2c.h"

/* Example code to talk to a MPU6050 MEMS acceleromter and gyroscope

   This is taking to simple approach of simply reading registers. It's perfectly
   possible to link up an interrupt line and set things up to read from the
   inbuilt FIFO to make it more useful.

   NOTE: Ensure the device is capable of being driven at 3.3v NOT 5v. The Pico
   GPIO (and therefor I2C) cannot be used at 5v.

   You will need to use a level shifter on the I2C lines if you want to run the
   board at 5v.

   Connections on Raspberry Pi Pico board, other boards may vary.

   GPIO 4 (pin 6)-> SDA on MPU6050 board
   GPIO 5 (pin 7)-> SCL on MPU6050 board
   3.3v (pin 36) -> VCC on MPU6050 board
   GND (pin 38)  -> GND on MPU6050 board
*/

// By default these devices  are on bus address 0x68
static int addr = 0x68;

#define I2C_PORT i2c0

static void mpu6050_reset() {
    // Two byte reset. First byte register, second byte data
    // There are a load more options to set up the device in different ways that could be added here
    uint8_t buf[] = {0x6B, 0x00};
    i2c_write_blocking(I2C_PORT, addr, buf, 2, false);
}

static void mpu6050_read_raw(int16_t accel[3], int16_t gyro[3], int16_t *temp) {
    // For this particular device, we send the device the register we want to read
    // first, then subsequently read from the device. The register is auto incrementing
    // so we dont need to keep sending the register we want, just the first.

    uint8_t buffer[6];

    // Start reading acceleration registers from register 0x3B for 6 bytes
    uint8_t val = 0x3B;
    i2c_write_blocking(I2C_PORT, addr, &val, 1, true); // true to keep master control of bus
    i2c_read_blocking(I2C_PORT, addr, buffer, 6, false);

    for (int i = 0; i < 3; i++) {
        accel[i] = (buffer[i * 2] << 8 | buffer[(i * 2) + 1]);
    }

    // Now gyro data from reg 0x43 for 6 bytes
    // The register is auto incrementing on each read
    val = 0x43;
    i2c_write_blocking(I2C_PORT, addr, &val, 1, true);
    i2c_read_blocking(I2C_PORT, addr, buffer, 6, false);  // False - finished with bus

    for (int i = 0; i < 3; i++) {
        gyro[i] = (buffer[i * 2] << 8 | buffer[(i * 2) + 1]);;
    }

    // Now temperature from reg 0x41 for 2 bytes
    // The register is auto incrementing on each read
    val = 0x41;
    i2c_write_blocking(I2C_PORT, addr, &val, 1, true);
    i2c_read_blocking(I2C_PORT, addr, buffer, 2, false);  // False - finished with bus

    *temp = buffer[0] << 8 | buffer[1];
}

int main() {
    stdio_init_all();

    printf("Hello, MPU6050! Reading raw data from registers...\n");

    // This example will use I2C0 on GPIO4 (SDA) and GPIO5 (SCL) running at 400kHz.
    i2c_init(I2C_PORT, 400 * 1000);
    gpio_set_function(4, GPIO_FUNC_I2C);
    gpio_set_function(5, GPIO_FUNC_I2C);
    gpio_pull_up(4);
    gpio_pull_up(5);

    mpu6050_reset();

    int16_t acceleration[3], gyro[3], temp;

    while (1) {
        mpu6050_read_raw(acceleration, gyro, &temp);

        // These are the raw numbers from the chip, so will need tweaking to be really useful.
        // See the datasheet for more information
        printf("Acc. X = %d, Y = %d, Z = %d\n", acceleration[0], acceleration[1], acceleration[2]);
        printf("Gyro. X = %d, Y = %d, Z = %d\n", gyro[0], gyro[1], gyro[2]);
        // Temperature is simple so use the datasheet calculation to get deg C.
        // Note this is chip temperature.
        printf("Temp. = %f\n", (temp / 340.0) + 36.53);

        sleep_ms(100);
    }

    return 0;
}
```

Bill of Materials

Table 11. A list of materials required for the example

Item	**Quantity**	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	http://raspberrypi.org/
MPU6050 board	1	generic part
M/M Jumper wires	4	generic part

Attaching a 16x2 LCD via I2C

This example code shows how to interface the Raspberry Pi Pico to one of the very common 16x2 LCD character displays. The display will need a 3.3V I2C adapter board as this example uses I2C for communications.

NOTE

These LCD displays can also be driven directly using GPIO without the use of an adapter board. That is beyond the scope of this example.

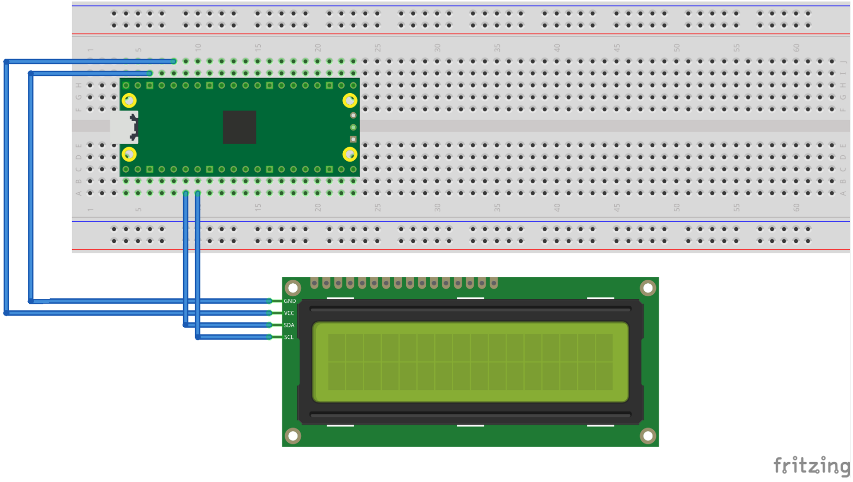
Wiring information

Wiring up the device requires 4 jumpers, to connect VCC (3.3v), GND, SDA and SCL. The example here uses I2C port 0, which is assigned to GPIO 4 (SDA) and 5 (SCL) in software. Power is supplied from the 3.3V pin.

WARNING

Many displays of this type are 5v. If you wish to use a 5v display you will need to use level shifters on the SDA and SCL lines to convert from the 3.3V used by the RP2040. Whilst a 5v display will just about work at 3.3v, the display will be dim.

Figure 7. Wiring Diagram for LCD1602A LCD with I2C bridge.



List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/pre_release/i2c/lcd_1602_i2c/CMakeLists.txt Lines 1 - 11

```
add_executable(lcd_1602_i2c
        lcd_1602_i2c.c
        )

# Pull in our (to be renamed) simple get you started dependencies
target_link_libraries(lcd_1602_i2c pico_stdlib hardware_i2c)

pico_set_program_url(lcd_1602_i2c "https://github.com/raspberrypi/pico-examples/tree/HEAD/i2c/lcd_1602_i2c")

# create map/bin/hex file etc.
```

```
pico_add_extra_outputs(lcd_1602_i2c)
```

lcd_1602_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/pre_release/i2c/lcd_1602_i2c/lcd_1602_i2c.c Lines 1 - 163

```
/**
 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */

#include <stdio.h>
#include <string.h>
#include "pico/stdlib.h"
#include "hardware/i2c.h"
#include "pico/binary_info.h"

/* Example code to drive a 16x2 LCD panel via a I2C bridge chip (e.g. PCF8574)

   NOTE: The panel must be capable of being driven  at 3.3v NOT 5v. The Pico
   GPIO (and therefor I2C) cannot be used at 5v.

   You will need to use a level shifter on the I2C lines if you want to run the
   board at 5v.

   Connections on Raspberry Pi Pico board, other boards may vary.

   GPIO 4 (pin 6)-> SDA on LCD bridge board
   GPIO 5 (pin 7)-> SCL on LCD bridge board
   3.3v (pin 36) -> VCC on LCD bridge board
   GND (pin 38)  -> GND on LCD bridge board
*/
// commands
const int LCD_CLEARDISPLAY = 0x01;
const int LCD_RETURNHOME = 0x02;
const int LCD_ENTRYMODESET = 0x04;
const int LCD_DISPLAYCONTROL = 0x08;
const int LCD_CURSORSHIFT = 0x10;
const int LCD_FUNCTIONSET = 0x20;
const int LCD_SETCGRAMADDR = 0x40;
const int LCD_SETDDRAMADDR = 0x80;

// flags for display entry mode
const int LCD_ENTRYSHIFTINCREMENT = 0x01;
const int LCD_ENTRYLEFT = 0x02;

// flags for display and cursor control
const int LCD_BLINKON = 0x01;
const int LCD_CURSORON = 0x02;
const int LCD_DISPLAYON = 0x04;

// flags for display and cursor shift
const int LCD_MOVERIGHT = 0x04;
const int LCD_DISPLAYMOVE = 0x08;

// flags for function set
const int LCD_5x10DOTS = 0x04;
const int LCD_2LINE = 0x08;
const int LCD_8BITMODE = 0x10;
```

```

// flag for backlight control
const int LCD_BACKLIGHT = 0x08;

const int LCD_ENABLE_BIT = 0x04;

#define I2C_PORT i2c0
// By default these LCD display drivers are on bus address 0x27
static int addr = 0x27;

// Modes for lcd_send_byte
#define LCD_CHARACTER  1
#define LCD_COMMAND    0

#define MAX_LINES      2
#define MAX_CHARS      16

/* Quick helper fucntion for single byte transfers */
void i2c_write_byte(uint8_t val) {
    i2c_write_blocking(I2C_PORT, addr, &val, 1, false);
}

void lcd_toggle_enable(uint8_t val) {
    // Toggle enable pin on LCD display
    // We cannot do this too quickly or things dont work
#define DELAY_US 600
    sleep_us(DELAY_US);
    i2c_write_byte(val | LCD_ENABLE_BIT);
    sleep_us(DELAY_US);
    i2c_write_byte(val & ~LCD_ENABLE_BIT);
    sleep_us(DELAY_US);
}

// The display is sent a byte as two separate nibble transfers
void lcd_send_byte(uint8_t val, int mode) {
    uint8_t high = mode | (val & 0xF0) | LCD_BACKLIGHT;
    uint8_t low = mode | ((val << 4) & 0xF0) | LCD_BACKLIGHT;

    i2c_write_byte(high);
    lcd_toggle_enable(high);
    i2c_write_byte(low);
    lcd_toggle_enable(low);
}

void lcd_clear(void) {
    lcd_send_byte(LCD_CLEARDISPLAY, LCD_COMMAND);
}

// go to location on LCD
void lcd_set_cursor(int line, int position) {
    int val = (line == 0) ? 0x80 + position : 0xC0 + position;
    lcd_send_byte(val, LCD_COMMAND);
}

void inline lcd_char(char val) {
    lcd_send_byte(val, LCD_CHARACTER);
}

void lcd_string(const char *s) {
    while (*s)
        lcd_char(*s++);
}

```

```
void lcd_init() {
    lcd_send_byte(0x03, LCD_COMMAND);
    lcd_send_byte(0x03, LCD_COMMAND);
    lcd_send_byte(0x03, LCD_COMMAND);
    lcd_send_byte(0x02, LCD_COMMAND);

    lcd_send_byte(LCD_ENTRYMODESET | LCD_ENTRYLEFT, LCD_COMMAND);
    lcd_send_byte(LCD_FUNCTIONSET | LCD_2LINE, LCD_COMMAND);
    lcd_send_byte(LCD_DISPLAYCONTROL | LCD_DISPLAYON, LCD_COMMAND);
    lcd_clear();
}

int main() {
    // This example will use I2C0 on GPIO4 (SDA) and GPIO5 (SCL)
    i2c_init(I2C_PORT, 100 * 1000);
    gpio_set_function(4, GPIO_FUNC_I2C);
    gpio_set_function(5, GPIO_FUNC_I2C);
    gpio_pull_up(4);
    gpio_pull_up(5);
    // Make the I2C pins available to picotool
    bi_decl( bi_2pins_with_func(4, 5, GPIO_FUNC_I2C));

    lcd_init();

    static uint8_t *message[] =
            {
                    "RP2040 by", "Raspberry Pi",
                    "A brand new", "microcontroller",
                    "Twin core M0", "Full C SDK",
                    "More power in", "your product",
                    "More beans", "than Heinz!"
            };

    while (1) {
        for (int m = 0; m < sizeof(message) / sizeof(message[0]); m += MAX_LINES) {
            for (int line = 0; line < MAX_LINES; line++) {
                lcd_set_cursor(line, (MAX_CHARS / 2) - strlen(message[m + line]) / 2);
                lcd_string(message[m + line]);
            }
            sleep_ms(2000);
            lcd_clear();
        }
    }

    return 0;
}
```

Bill of Materials

Table 12. A list of materials required for the example

Item	**Quantity**	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	http://raspberrypi.org/
1602A based LCD panel 3.3v	1	generic part
1602A to I2C bridge device 3.3v	1	generic part
M/M Jumper wires	4	generic part

Appendix B: SDK Configuration

SDK configuration is the process of customising the SDK differently to the defaults. In cases where you do need to make changes for specific circumstances, this chapter will show how that can be done, and what parameters can be changed.

Configuration is done by setting various predefined values in header files in your code. These will override the default values from the SDK itself.

So for example, if you wish to change the ID of the mutex used to protect access to spinlocks from its default value (8) to 9, you would add the following to your project header files, before any SDK includes.

```
#define PICO_SPINLOCK_ID_MUTEX 9`
```

Configuration Parameters

TO DO : this needs to be updated/corrected

Table 13. SDK Configuration Parameters

Parameter name	Used by	Default	Purpose
PICO_SPINLOCK_ID_MUTEX	sync.h	8	Mutex used when accessing Spinlocks
PICO_SPINLOCK_ID_IRQ	sync.h	9	
PICO_SPINLOCK_ID_TIMER	sync.h	10	
PICO_SPINLOCK_ID_HARDWARE_CLAIM	sync.h	11	Spinlock used when using `hw_claim_lock()`
PICO_SPINLOCK_ID_STRIPED_FIRST	sync.h	16	
PICO_SPINLOCK_ID_STRIPED_LAST	sync.h	23	
PICO_SPINLOCK_ID_USER0	sync.h	16	
PICO_STACK_SIZE			
PICO_NO_RAM_VECTOR_TABLE		0	
PICO_PHEAP_MAX_ENTRIES	pheap.h		
PICO_USBDEV_ENABLE_DEBUG_TRACE	usbdevice.h	0	
PICO_USBDEV_ASSUME_ZERO_INIT	usbdevice.h	0	
PICO_USBDEV_MAX_ENDPOINTS	usbdevice.h	USB_NUM_ENDPOINTS	
PICO_USBDEV_MAX_DESCRIPTOR_SIZE	usbdevice.h	64	
PICO_USBDEV_NO_DEVICE_SETUP_HANDLER	usbdevice.h	0	

Parameter name	Used by	Default	Purpose
PICO_USBDEV_NO_ENDPOINT_SETUP_HANDLER	usbdevice.h	0	
PICO_USBDEV_BULK_ONLY_EP1_THRU_16	usbdevice.h	0	
PICO_USBDEV_USE_ZERO_BASED_INTERFACES	usbdevice.h	0	
PICO_USBDEV_NO_TRANSFER_ON_INIT_METHOD	usbdevice.h	0	
PICO_USBDEV_NO_TRANSFER_ON_CANCEL_METHOD	usbdevice.h	0	
PICO_USBDEV_NO_INTERFACE_ALTERNATES	usbdevice.h	0	Disable alternative interfaces
PICO_USBDEV_ISOCHRONOUS_BUFFER_STRIDE_TYPE	usbdevice.h	0	Stride Type

Appendix C: Board Configuration

Board Configuration

Board configuration is the process of customising the SDK to run on a specific board design. The SDK comes some predefined configurations for boards produced by Raspberry Pi, the main (and default) example being the Raspberry Pi Pico.

Configurations specify a number of parameters that could vary between hardware designs. For example, default UART ports, on board LED locations and flash capacities etc.

This chapter will go though where these configurations files are, how to make changes and set parameters, and how to build your SDK using CMake with your customisations.

The Configuration files

Board specfic configuration files are stored in the SDK source tree, at `…/src/boards/include/boards/<boardname>.h`. The default configuration file is that for the Raspberry Pi Pico, and at the time of writing is:

`<sdk_path>/src/boards/include/boards/pico.h`

This relatively short file contains overrides from default of a small number of parameters used by the SDK when building code.

Pico SDK: https://github.com/raspberrypi/pico-sdk/tree/pre_release/src/boards/include/boards/pico.h Lines 1 - 52

```
/*
 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */

// -----------------------------------------------------
// NOTE: THIS HEADER IS ALSO INCLUDED BY ASSEMBLER SO
//       SHOULD ONLY CONSIST OF PREPROCESSOR DIRECTIVES
// -----------------------------------------------------

// This header may be included by other board headers as "boards/pico.h"

#ifndef _BOARDS_PICO_H
#define _BOARDS_PICO_H

#ifndef PICO_DEFAULT_UART
#define PICO_DEFAULT_UART 0
#endif

#ifndef PICO_DEFAULT_UART_TX_PIN
#define PICO_DEFAULT_UART_TX_PIN 0
#endif

#ifndef PICO_DEFAULT_UART_RX_PIN
#define PICO_DEFAULT_UART_RX_PIN 1
#endif

#ifndef PICO_DEFAULT_LED_PIN
#define PICO_DEFAULT_LED_PIN 25
#endif
```

```

#ifndef PICO_FLASH_SPI_CLKDIV
#define PICO_FLASH_SPI_CLKDIV 2
#endif

#ifndef PICO_FLASH_SIZE_BYTES
#define PICO_FLASH_SIZE_BYTES (2 * 1024 * 1024)
#endif

// Drive high to force power supply into PWM mode (lower ripple on 3V3 at light loads)
#define PICO_SMPS_MODE_PIN 23

#ifndef PICO_FLOAT_SUPPORT_ROM_V1
#define PICO_FLOAT_SUPPORT_ROM_V1 1
#endif

#ifndef PICO_DOUBLE_SUPPORT_ROM_V1
#define PICO_DOUBLE_SUPPORT_ROM_V1 1
#endif

#endif
```

As can be seen, it sets up the default UART to `uart0`, the GPIO pins to be used for that UART, the GPIO pin used for the onboard LED, and the flash size.

To create your own configuration file, create a file in the board `../source/folder` withthe name of your board, for example, `myboard.h`. Enter your board specific parameters in this file.

Building applications with a custom board configuration

The CMake system is what specifies which board configuration is going to be used.

To create a new build based on a new board configuration (we will use the `myboard` example from the previous section) first create a new build folder under your project folder. For our example we will use the pico-examples folder.

```
$ cd pico-examples
$ mkdir myboard_build
$ cd myboard_build
```

then run cmake as follows:

`cmake -D"PICO_BOARD=myboard" ..`

This will set up the system ready to build so you can simply type `make` in the `myboard_build` folder and the examples will be built for your new board configuration.

Available configuration parameters

TO DO : this needs to be updated/corrected

Table 14. Board Configuration Parameters

Parameter name	Used by	Default	Purpose
PICO_DEFAULT_UART	uart.h	0	Define which uart is the default. i.e. `0`, `1`
PICO_DEFAULT_UART_TX_PIN	uart.h	0	Define which GPIO pin is used for TX

Parameter name	Used by	Default	Purpose
PICO_DEFAULT_UART_RX_PIN	uart.h	1	Define which GPIO pin is used for RX
PICO_UART_ENABLE_CRLF_SUPPORT	uart.h	1	Enable/disable CRLF translation
PICO_DEFAULT_UART_BAUD_RATE	uart.h	115200	Set the baud rate of the default uart
PICO_DEFAULT_LED_PIN	user	25	Defines a default LED pin
PICO_FLASH_SIZE_BYTES	flash.c	None	Specify the flash memory size (in bytes)
PICO_SMPS_MODE_PIN			
PICO_FLASH_SPI_CLKDIV		None	Set the SPI Flash clock divider
PICO_SD_CLK_PIN	sdcard.h	23	
PICO_SD_CMD_PIN	sdcard.h	24	
PICO_SD_DAT0_PIN	sdcard.h	19	

Appendix D: Tuning for size and performance

TODO: wnyip

Appendix E: Building the Pico SDK API documentation

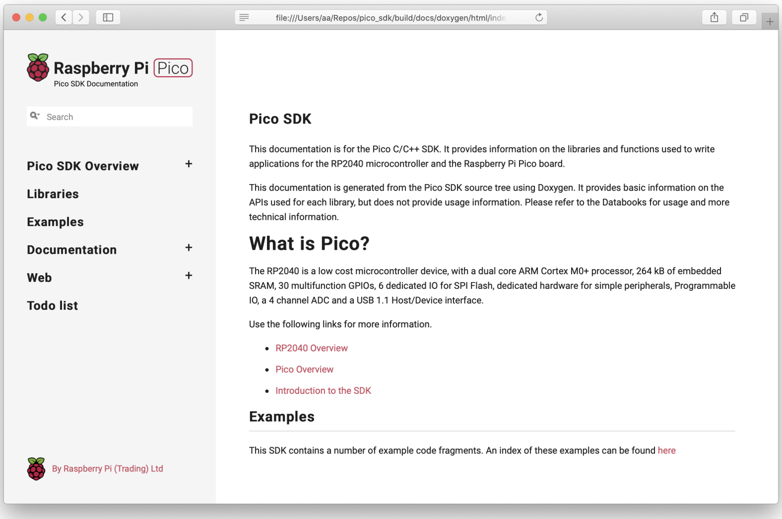
The Pico SDK documentation can also be found as part of Pico SDK itself and can be built directly from the command line. If you haven't already checked out the Pico SDK repository you should do so,

```
$ cd ~/
$ mkdir pico
$ cd pico
$ git clone -b pre_release git@github.com:raspberrypi/pico-sdk.git
$ cd pico-sdk
$ git submodule update --init --recursive
$ cd ..
$ git clone -b pre_release git@github.com:raspberrypi/pico-examples.git
```

and then afterwards you can go ahead and build the documentation.

```
$ cd pico-sdk
$ mkdir build
$ cd build
$ cmake ..
$ make docs
```

The API documentation will be build and can be found in the `pico-sdk/build/docs/doxygen/html` directory, see Figure 8.



Pico SDK

This documentation is for the Pico C/C++ SDK. It provides information on the libraries and functions used to write applications for the RP2040 microcontroller and the Raspberry Pi Pico board.

This documentation is generated from the Pico SDK source tree using Doxygen. It provides basic information on the APIs used for each library, but does not provide usage information. Please refer to the Databooks for usage and more technical information.

What is Pico?

The RP2040 is a low cost microcontroller device, with a dual core ARM Cortex M0+ processor, 264 kB of embedded SRAM, 30 multifunction GPIOs, 6 dedicated IO for SPI Flash, dedicated hardware for simple peripherals, Programmable IO, a 4 channel ADC and a USB 1.1 Host/Device interface.

Use the following links for more information.

- RP2040 Overview
- Pico Overview
- Introduction to the SDK

Examples

This SDK contains a number of example code fragments. An index of these examples can be found here

Figure 8. The Pico SDK API documentation

When building the documentation `cmake` will go ahead and clone the `pico-examples` repo into the `build` directory directly from Github. This can be over ridden if you have a local copy on the command line,

```
$ cmake -DPICO_EXAMPLES_PATH=../../pico-examples ..
```

or by using an environment variable.

```
$ export PICO_EXAMPLES_PATH=../../pico-examples
$ cmake ..
```

Raspberry Pi

Raspberry Pi Trading Ltd