Raspberry Pi Pico-series C/C++ SDK

Libraries and tools for C/C++ development on Raspberry Pi microcontrollers.

Raspberry Pi Ltd

Colophon

Throughout the text "the SDK" refers to our Raspberry Pi Pico SDK. More details about the SDK can be found throughout this book. Source code included in the documentation is Copyright © 2020-2024 Raspberry Pi Ltd (formerly Raspberry Pi (Trading) Ltd.) and licensed under the 3-Clause BSD license.

build-date: 2025-02-20
build-version: 3184e62-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 LTD ("RPL") "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 RPL 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.

RPL 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 RPL harmless against all liabilities, costs, damages or other losses arising out of their use of the RESOURCES.

RPL 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 RPL 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"). RPL 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 RPL's Standard Terms. RPL's provision of the RESOURCES does not expand or otherwise modify RPL's Standard Terms including but not limited to the disclaimers and warranties expressed in them.

Table of contents

Chapter 1. About the SDK

1.1. Introduction

The SDK (Software Development Kit) provides the headers, libraries and build system necessary to write programs for RP-series microcontroller-based devices such as Raspberry Pi Pico or Raspberry Pi Pico 2 in C, C++ or Arm assembly language.

The 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 RP-series microcontroller's hardware, including DMA, IRQs, and the wide variety fixed function peripherals and PIO (Programmable IO).

Additionally, the SDK provides higher level libraries for dealing with timers, synchronization, Wi-Fi and Bluetooth networking, USB and multicore programming. These libraries should be comprehensive enough that your application code rarely, if at all, needs to access hardware registers directly. However, if you do need or prefer to access the raw hardware registers, you will also find complete and fully-commented register definition headers in the SDK. There's no need to look up addresses in the datasheet.

The SDK can be used to build anything from simple applications, fully-fledged runtime environments such as MicroPython, to low level software such as the RP-series microcontroller's on-chip bootrom itself.

The design goal for entire SDK is to be simple but powerful.

> **Looking to get started?**
>
> This book documents the SDK APIs, explains the internals and overall design of the SDK, and explores some deeper topics like using the PIO assembler to build new interfaces to external hardware. For a quick start with setting up the SDK and writing SDK programs, **Getting started with Raspberry Pi Pico-series** is the best place to start.

1.2. Anatomy of a SDK Application

Before going completely depth-first in our traversal of the SDK, it's worth getting a little breadth by looking at one of the SDK examples covered in **Getting started with Raspberry Pi Pico-series**, in more detail.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/blink_simple/blink_simple.c

```c
/**
 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */

#include "pico/stdlib.h"

#ifndef LED_DELAY_MS
#define LED_DELAY_MS 250
#endif

#ifndef PICO_DEFAULT_LED_PIN
#warning blink_simple example requires a board with a regular LED
#endif

```

```c
// Initialize the GPIO for the LED
void pico_led_init(void) {
#ifdef PICO_DEFAULT_LED_PIN
    // A device like Pico that uses a GPIO for the LED will define PICO_DEFAULT_LED_PIN
    // so we can use normal GPIO functionality to turn the led on and off
    gpio_init(PICO_DEFAULT_LED_PIN);
    gpio_set_dir(PICO_DEFAULT_LED_PIN, GPIO_OUT);
#endif
}

// Turn the LED on or off
void pico_set_led(bool led_on) {
#if defined(PICO_DEFAULT_LED_PIN)
    // Just set the GPIO on or off
    gpio_put(PICO_DEFAULT_LED_PIN, led_on);
#endif
}

int main() {
    pico_led_init();
    while (true) {
        pico_set_led(true);
        sleep_ms(LED_DELAY_MS);
        pico_set_led(false);
        sleep_ms(LED_DELAY_MS);
    }
}
```

This program consists only of a single C file, with three functions. As with almost any C programming environment, the function called `main()` is special, and is the point where the language runtime first hands over control to your program. In the SDK the `main()` function does not take any arguments. It's quite common for the `main()` function not to return, as is shown here.

NOTE

The return code of `main()` is ignored by the SDK runtime, and the default behaviour is to hang the processor on exit.

At the top of the C file, we include a header called `pico/stdlib.h`. This is an umbrella header that pulls in some other commonly used headers. In particular, the ones needed here are `hardware/gpio.h`, which is used for accessing the general purpose IOs on RP-series microcontrollers (the `gpio_xxx` functions here), and `pico/time.h` which contains, among other things, the `sleep_ms` function. Broadly speaking, a library whose name starts with `pico` provides high level APIs and concepts, or aggregates smaller interfaces; a name beginning with `hardware` indicates a thinner abstraction between your code and the RP-series microcontroller on-chip hardware.

So, using mainly the `hardware_gpio` and `pico_time` libraries, this C program will blink an LED connected to the default LED GPIO (which exact pin varies from one RP-series microcontroller board to another) on and off, twice per second, forever (or at least until unplugged). In the directory containing the C file (you can click the link above the source listing to go there), there is one other file which lives alongside it.

Directory listing of pico-examples/blink_simple

```
blink_simple
├── blink_simple.c
└── CMakeLists.txt

0 directories, 2 files
```

The second file is a *CMake* file, which tells the SDK how to turn the C file into a binary application for an RP-series microcontroller-based board. Later sections will detail exactly what CMake is, and why it is used, but we can look at the contents of this file without getting mired in those details.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/blink_simple/CMakeLists.txt

```
add_executable(blink_simple
        blink_simple.c
)

# pull in common dependencies
target_link_libraries(blink_simple pico_stdlib)

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

# call pico_set_program_url to set path to example on github, so users can find the source
for an example via picotool
example_auto_set_url(blink_simple)
```

The standard CMake function `add_executable` in this file declares that a program called `blink_simple` should be built from the C file shown earlier. This is also the "target" name in CMake, and is also used when building the program individually. For example, in the `pico-examples` repository you can say `make blink_simple` in your build directory, and that name comes from *this* line. You can have multiple executables in a single project, and the `pico-examples` repository is one such project.

The `target_link_libraries` is pulling in the SDK functionality that our program needs. If you don't ask for a library, it doesn't appear in your program binary. Just like `pico/stdlib.h` is an umbrella header that includes things like `pico/time.h` and `hardware/gpio.h`, `pico_stdlib` is an umbrella *library* that makes libraries like `pico_time` and `hardware_gpio` available to your build, so that those headers can be included in the first place, and the extra C source files are compiled and linked. If you need less common functionality, not included with `pico_stdlib`, like accessing the DMA hardware, you should add those dependencies here (e.g. listing `hardware_dma` before or after `pico_stdlib`).

We could end the CMake file here, and that would be enough to build the `blink_simple` program. By default, the build will produce an ELF file (executable linkable format), containing all of your code and the SDK libraries it uses. You can load an ELF into the RP-series microcontroller's RAM or external flash through the Serial Wire Debug port, with a debugger setup like `gdb` and `openocd`, or via `picotool`. It's often easier to program your Pico-series device or other RP-series microcontroller board directly over USB with BOOTSEL mode, and this requires a different type of file, called UF2, which serves the same purpose here as an ELF file, but is constructed to survive the rigours of USB mass storage transfer more easily. The `pico_add_extra_outputs` function declares that you want a UF2 file to be created, as well as some useful extra build output like disassembly and map files.

NOTE

The ELF file is converted to a UF2 using `picotool`.

The final `example_auto_set_url` function is used to embed a link back to the example soource code on github into the output binary such that it can be displayed via `picotool info blink_simple.elf`. You'll see this on the `pico-examples` applications, but it's not applicable to your own programs.

Finally, a brief note on the `pico_stdlib` library. Besides common hardware and high-level libraries like `hardware_gpio` and `pico_time`, it also pulls in system components like `pico-runtime`, which is needed to set up the hardware and runtime environment that lets you just implement`main()` and `pico_standard_link` which configures the linking of your executable whilst using a simple `CMakeLists.txt`. These are incredibly low-level components that most users will not need to worry about. The reason they are mentioned is to point out that they are ultimately *implicit dependencies* of your program because of your dependence on `pico_stdlib`; if you choose not depend on `pico_stdlib` and then you can pick just the exact SDK libraries you want explcitly.

Chapter 2. SDK architecture

RP-series microcontrollers are powerful chips, and in particular were designed with a disproportionate amount of system RAM for their point in the microcontroller design space. However it *is* an embedded environment, so RAM, CPU cycles and program space are still at a premium. As a result 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 next few sections try to highlight some of the design decisions behind the SDK: the how and the why, as much as the what.

NOTE

Some parts of this overview are quite technical or deal with very low-level parts of the SDK and build system. You might prefer to skim this section at first and then read it thoroughly at a later time, after writing a few SDK applications.

2.1. The Build System

The SDK uses CMake to manage the build. CMake is widely supported by IDEs (Integrated Development Environments), which can use a `CMakeLists.txt` file to discover source files and generate code autocomplete suggestions. The same `CMakeLists.txt` file provides a terse specification of how your application (or your project with many distinct applications) should be built, which CMake uses to generate a robust build system used by `make`, `ninja` or other build tools. The build system produced is customised for the platform (e.g. Windows, or a Linux distribution) and by any configuration variables the developer chooses.

Section 2.6 shows how CMake can set configuration defines for a particular program, or based on which RP-series microcontroller *board* you are building for, to configure things like default pin mappings and features of SDK libraries. These defines are listed in Chapter 5, and Board Configuration files are covered in more detail in Chapter 8. Additionally Chapter 6 describes CMake variables you can use to control the functionality of the build itself.

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

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/blink_simple/CMakeLists.txt

```
add_executable(blink_simple
        blink_simple.c
)

# pull in common dependencies
target_link_libraries(blink_simple pico_stdlib)

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

# call pico_set_program_url to set path to example on github, so users can find the source for an example via picotool
example_auto_set_url(blink_simple)
```

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

The 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 certain code contained in the bootrom within the RP-series microcontroller).

pico_stdlib is an INTERFACE library and provides all the rest of the code and configuration needed to compile and link the blink application. You will notice if you watch a build of blink_simple (https://github.com/raspberrypi/pico-examples/blob/master/blink_simple/blink_simple.c) that in addition to the single blink_simple.c file, the inclusion of pico_stdlib causes dozens of other source files to be compiled to flesh out the blink_simple application such that it can be run on a RP-series microcontroller.

2.2. Every Library is an INTERFACE Library

All libraries within the SDK are CMake INTERFACE libraries. (Note this does not include the C/C++ standard libraries provided by the compiler). Conceptually, a CMake INTERFACE library is a collection of:

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

The INTERFACE libraries form a tree of dependencies, with each contributing source files, include paths, compiler definitions and compile/link options to the build. These are collected based on the libraries you have listed in your CMakeLists.txt file, and the libraries depended on by *those* libraries, and so on recursively. 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.

When building an executable with the SDK, all of the code for one executable, including the SDK libraries, is (re)compiled for *that* executable from source. Building in this way allows your build configuration to specify customised settings for those libraries (e.g. enabling/disabling assertions, setting the sizes of static buffers), on a *per-application* basis, at compile time. This allows for faster and smaller binaries, in addition of course to the ability to remove support for unwanted features from your executable entirely.

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, and the linker will garbage collect any functions you don't call, so this doesn't bloat your binary. We can take a quick peek into the directory structure of the hardware_gpio library, which our blink_simple example uses to turn the LED on and off:

```
hardware_gpio
├── CMakeLists.txt
├── gpio.c
└── include
    └── hardware
        └── gpio.h
```

Depending on the hardware_gpio INTERFACE library in your application causes gpio.c to be compiled and linked into your executable, and adds the include directory shown here to your search path, so that a #include "hardware/gpio.h" will pull in the correct header in your code.

INTERFACE libraries also make it easy to aggregate functionality into readily consumable chunks (such as pico_stdlib), which don't directly contribute any code, but depend on a handful of lower-level libraries that do. Like a metapackage, this lets you pull in a group of libraries related to a particular goal without listing them all by name.

IMPORTANT

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 through 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.

2.3. SDK Library Structure

The full API listings are given in Chapter 4; this chapter gives an overview of how SDK libraries are organised, and the relationships between them.

There are a number of layers of libraries within the SDK. This section starts with the highest-level libraries, which can be used in C or C++ applications, and navigates all the way down to the `hardware_regs` library, which is a comprehensive set of hardware definitions suitable for use in Arm assembly as well as C and C++, before concluding with a brief note on how the TinyUSB stack can be used from within the SDK.

2.3.1. Higher-level Libraries

These libraries (`pico_xxx`) provide higher-level APIs, concepts and abstractions that are common to most RP-series microcontroller-based applications. The APIs are listed in High Level APIs. These may be libraries that have cross-cutting concerns between multiple pieces of hardware (for example the `sleep_` functions in `pico_time` need to concern themselves both with the RP-series microcontrollers' timer hardware and with how processors enter and exit low power states), or they may be pure software infrastructure required for your program to run smoothly. This includes libraries for things like:

- Alarms, timers and time functions
- Multi-core support and synchronization primitives
- 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 are added over time. Certain additional libraries that are not fully supported/stable/documented (e.g. - Audio support (via PIO), DPI/VGA/MIPI Video support (via PIO), file system support, SDIO support via (PIO)) are included in the Pico Extras GitHub repository.

2.3.2. Runtime Support Libraries

These libraries provide basic application features required for a basic program.

- Runtime startup and initialization functions, e.g. performing minimal hardware initialisation (e.g. default PLL and clock configuration), and calling functions with `constructor` attributes before entering `main()`
- Low level interfacing with the C/C++ runtime library
- Hardaware/bootrom accelerated single and double-precision floating point support.

- Compact `printf` support, and `stdio` support via `UART`, `USB`, `semihosting` and `Segger RTT`
- On RP2040, language level `/` and `%` support for fast division using RP2040 hardware dividers
- Standard runtime linking setup with default linker scripts

> **NOTE**
>
> There is more high-level discussion of the aggregate library `pico_runtime` in Section 2.7

2.3.3. Hardware Support Libraries

These are individual libraries (`hardware_xxx`) providing actual APIs for interacting with each piece of physical hardware/peripheral. They are lightweight and provide only thin abstractions. The APIs are listed in Hardware APIs.

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);
```

The `hardware_` libraries are intended to have a very minimal runtime cost. They generally do not require any or much RAM, and rarely rely on other runtime infrastructure. In general their only dependencies are the `hardware_structs` and `hardware_regs` libraries that contain definitions of memory-mapped register layout on the RP-series microcontroller. As such they can be used by low-level or other specialized applications that don't want to use the rest of the SDK libraries and runtime.

> **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/blob/master/src/rp2_common/hardware_pio/include/hardware/pio.h directly as shown above.
>
> Using `static inline` functions is common in SDK header files because such methods are often called with parameters that have fixed known values at compile time. In such cases, the compiler is often able to fold the code down to a single register write (or in this case a read, AND with a constant value, OR with a constant value, and a write) with no function call overhead. This tends to produce much smaller and faster binaries.

2.3.3.1. 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.

There are individual claiming/unclaiming methods in the respective `hardware_` libraries.

2.3.4. Hardware Structs Library

The `hardware_structs` library provides a set of C structures which represent the memory mapped layout of the RP-series microcontroller registers in the system address space. This allows you to replace something like the following (which you'd write in C with the defines from the lower-level `hardware_regs`)

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

with something like this (where `pio0` is a pointer 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-prone-ness of dealing with individual memory locations, pointer casts and volatile access. As a bonus, the structs tend to produce better code with older compilers, as they encourage the reuse of a base pointer with offset load/stores, instead of producing a 32 bit literal for every register accessed.

The struct headers are named consistently with both the `hardware_` libraries and the `hardware_regs` register headers. For example, if you access the `hardware_pio` library's functionality through `hardware/pio.h`, the `hardware_structs` library (a dependee of `hardware_pio`) contains a header you can include as `hardware/structs/pio.h` if you need to access a register directly, and this itself will pull in `hardware/regs/pio.h` for register field definitions. The PIO header is a bit lengthy to include here. `hardware/structs/pll.h` is a shorter example to give a feel for what these headers actually contain:

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2350/hardware_structs/include/hardware/structs/pll.h Lines 27 - 74

```
27 typedef struct {
28     _REG_(PLL_CS_OFFSET) // PLL_CS
29     // Control and Status
30     // 0x80000000 [31]    LOCK         (0) PLL is locked
31     // 0x40000000 [30]    LOCK_N       (0) PLL is not locked +
32     // 0x00000100 [8]     BYPASS       (0) Passes the reference clock to the output instead of the...
33     // 0x0000003f [5:0]   REFDIV       (0x01) Divides the PLL input reference clock
34     io_rw_32 cs;
35 
36     _REG_(PLL_PWR_OFFSET) // PLL_PWR
37     // Controls the PLL power modes
38     // 0x00000020 [5]     VCOPD        (1) PLL VCO powerdown +
39     // 0x00000008 [3]     POSTDIVPD    (1) PLL post divider powerdown +
40     // 0x00000004 [2]     DSMPD        (1) PLL DSM powerdown +
41     // 0x00000001 [0]     PD           (1) PLL powerdown +
42     io_rw_32 pwr;
43 
44     _REG_(PLL_FBDIV_INT_OFFSET) // PLL_FBDIV_INT
45     // Feedback divisor
46     // 0x00000fff [11:0]  FBDIV_INT    (0x000) see ctrl reg description for constraints
47     io_rw_32 fbdiv_int;
48 
49     _REG_(PLL_PRIM_OFFSET) // PLL_PRIM
50     // Controls the PLL post dividers for the primary output
51     // 0x00070000 [18:16] POSTDIV1     (0x7) divide by 1-7
52     // 0x00007000 [14:12] POSTDIV2     (0x7) divide by 1-7
53     io_rw_32 prim;
54 
```

```
    _REG_(PLL_INTR_OFFSET) // PLL_INTR
    // Raw Interrupts
    // 0x00000001 [0]     LOCK_N_STICKY (0)
    io_rw_32 intr;

    _REG_(PLL_INTE_OFFSET) // PLL_INTE
    // Interrupt Enable
    // 0x00000001 [0]     LOCK_N_STICKY (0)
    io_rw_32 inte;

    _REG_(PLL_INTF_OFFSET) // PLL_INTF
    // Interrupt Force
    // 0x00000001 [0]     LOCK_N_STICKY (0)
    io_rw_32 intf;

    _REG_(PLL_INTS_OFFSET) // PLL_INTS
    // Interrupt status after masking & forcing
    // 0x00000001 [0]     LOCK_N_STICKY (0)
    io_ro_32 ints;
} pll_hw_t;
```

The structure contains the layout of the hardware registers in a block, and some defines bind that layout to the base addresses of the *instances* of that peripheral in the RP-series microcontroller global address map.

Additionally, you can use one of the atomic `set`, `clear`, or `xor` address aliases of a piece of hardware to *set*, *clear* or *toggle* respectively the specified 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;
```

Or, equivalently:

```
hw_set_bits(&pio0->sm[1].shiftctrl, PIO_SM1_SHIFTCTRL_AUTOPULL_BITS);
```

NOTE

The hardware atomic set/clear/XOR IO aliases are used extensively in the SDK libraries, to avoid certain classes of data race when two cores, or an IRQ and foreground code, are accessing registers concurrently.

NOTE

On RP-series microcontrollers, the atomic register aliases are a native part of the *peripheral*, not a CPU function, so the system DMA can also perform atomic set/clear/XOR operation on registers.

2.3.5. Hardware Registers Library

The `hardware_regs` library is a complete set of include files for all RP-series microcontroller 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++.

For example, here is a snippet from `hardware/regs/sio.h`:

```
// Description    : Single-cycle IO block
//                  Provides core-local and inter-core hardware for the two
//                  processors, with single-cycle access.
// =============================================================================
#ifndef HARDWARE_REGS_SIO_DEFINED
#define HARDWARE_REGS_SIO_DEFINED
// =============================================================================
// Register    : SIO_CPUID
// Description : Processor core identifier
//               Value is 0 when read from processor core 0, and 1 when read
//               from processor core 1.
#define SIO_CPUID_OFFSET 0x00000000
#define SIO_CPUID_BITS   0xffffffff
#define SIO_CPUID_RESET  "-"
#define SIO_CPUID_MSB    31
#define SIO_CPUID_LSB    0
#define SIO_CPUID_ACCESS "RO"

#endif
```

These header files are fairly heavily commented (the same information as is present in the datasheet register listings, or the SVD files). They define the offset of every register, and the layout of the fields in those registers, as well as the access type of the field, e.g. "RO" for read-only.

TIP

The headers in `hardware_regs` contain *only* comments and `#define` statements. This means they can be included from assembly files (`.S`, so the C preprocessor can be used), as well as C and C++ files.

2.3.6. TinyUSB Port

In addition to the core SDK libraries, we provide a RP-series microcontroller port of TinyUSB as the standard device and host USB support library within the SDK, and the SDK contains some build infrastructure for easily pulling this into your application.

The `tinyusb_dev` or `tinyusb_host` libraries within the SDK can be included in your application dependencies in `CMakeLists.txt` to add device or host support to your application respectively. Additionally, the `tinyusb_board` library is available to provide the additional "board support" code often used by TinyUSB demos. See the README in Pico Examples for more information and example code for setting up a fully functional application.

IMPORTANT

RP-series microcontroller USB hardware supports both Host and Device modes, but the two can not be used concurrently. TinyUSB can however also provide support for USB implemented via PIO, which is exposed, if available, via `tinyusb_pico_pio_usb`.

2.3.7. FreeRTOS Ports

FreeRTOS ports are available for RP2040 and RP2350 (both under Arm and RISC-V) either on a single core or in dual-core SMP mode.

The SDK does not directly depend on FreeRTOS, but does provide some libraries (particularly for networking) that are designed to be used with FreeRTOS. The pico-examples repository contains examples that use FreeRTOS, and when building you should set `FREERTOS_KERNEL_PATH`.

The ports are available in the FreeRTOS-Kernel repository.

NOTE

On RP2350, you must initialize the submodules of the FreeRTOS-Kernel repository.

To add FreeRTOS to your project:

1. Copy FreeRTOS_Kernel_import.cmake into your project.
2. Add a `FreeRTOSConfig.h` file to your project.
3. Add FreeRTOS libraries to `CMakeLists.txt`. The following code snippet shows a basic example:

```
# The following CMake or environment variables should be defined:
# PICO_SDK_PATH = path/to/sdk
# FREERTOS_KERNEL_PATH = path/to/FreeRTOS-Kernel
cmake_minimum_required(VERSION 3.14)

# Pull in SDK (must be before project)
include(pico_sdk_import.cmake)

project(freertos-sample C CXX ASM)

set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)

# Initialize the SDK
pico_sdk_init()

# FREERTOS: include FreeRTOS Kernel libraries
include(FreeRTOS_Kernel_import.cmake)

# assuming you have a sample.c!
add_executable(sample sample.c)

# FREERTOS: FreeRTOSConfig.h needs to be in the include path
target_include_directories(sample PRIVATE ${CMAKE_CURRENT_LIST_DIR})

# FREERTOS: Note, you should pick the FreeRTOS library that suits you best:
#
#           FreeRTOS-Kernel-Heap1 thru FreeRTOS-Kernel_Heap4
#           or
#           FreeRTOS-Kernel-Static
target_link_libraries(sample
        pico_stdlib
        FreeRTOS-Kernel-Heap4
)

pico_add_extra_outputs(sample)
```

2.3.8. Wi-Fi on Pico W

The IP support within the Pico SDK is provided by lwIP. The lwIP *raw* API is always supported: the full API, including blocking sockets, may be used under FreeRTOS.

There are a number of different library building blocks used within the IP and Wi-Fi support`: `pico_lwip` for lwIP, `pico_cyw43_driver` for the Wi-Fi chip driver, `pico_async_context` for accessing the non-thread-safe API (lwIP) in a consistent way whether polling, using multiple cores, or running FreeRTOS.

IMPORTANT

> By default `libcyw43` is licensed for non-commercial use, but users of Raspberry Pi Pico W, Pico WH, or anyone else who builds their product around RP2040 and CYW43439, benefit from a free commercial-use licence.

These libraries can be composed individually by advanced users, but in most common cases they are rolled into a few convenience libraries that you can add to your application's dependencies in `CMakeLists.txt`:

- **pico_cyw43_arch_lwip_poll** - For single-core, traditional polling-style access to lwIP on Pico W.
- **pico_cyw43_arch_threadsafe_background** - For single or multicore access to lwIP on Pico W, with lwIP callbacks handled in a low-priority interrupt, so no polling is required.
- **pico_cyw43_arch_lwip_sys_freertos** - For full access to the lwIP APIs (`NO_SYS=0`) under FreeRTOS.

For fuller details see the pico_cyw43_arch header file. Many examples of using Wi-Fi and lwIP with the Pico SDK may be found in the pico-examples repository.

2.3.9. Bluetooth on Pico W

The Bluetooth support within the Pico SDK is provided by BTstack. Documentation for BTstack can be found on BlueKitchen's website.

IMPORTANT

> In addition to the standard BTstack licensing terms, a supplemental licence which covers commercial use of BTstack with Raspberry Pi Pico W or Raspberry Pi Pico WH is provided.

See the pico-examples repository for Bluetooth examples including the examples from BTstack.

The Bluetooth support within the SDK is composed of multiple libraries:

The `pico_btstack_ble` library adds the support needed for Bluetooth Low Energy (BLE), and the `pico_btstack_classic` library adds the support needed for Bluetooth Classic. You can link to either library individually, or to both libraries enabling the dual-mode support provided by BTstack.

The `pico_btstack_cyw43` library is required for Bluetooth use. It adds support for the Bluetooth hardware on the Pico W, and integrates the BTstack run loop concept with the SDK's `pico_async_context` library allowing for running Bluetooth either via polling or in the background, along with multicore and/or FreeRTOS support.

The following additional libraries are optional:

- **pico_btstack_sbc_encoder** - Adds Bluetooth Sub Band Coding (SBC) encoder support.
- **pico_btstack_sbc_decoder** - Adds Bluetooth Sub Band Coding (SBC) decoder support.
- **pico_btstack_bnep_lwip** - Adds Bluetooth Network Encapsulation Protocol (BNEP) support using LwIP.
- **pico_btstack_bnep_lwip_sys_freertos** - Adds Bluetooth Network Encapsulation Protocol (BNEP) support using LwIP with FreeRTOS in `NO_SYS=0` mode.

To use BTstack you must add `pico_btstack_cyw43` and one or both of `pico_btsack_ble` and `pico_sbtstack_classic` to your application dependencies in your CMakeLists.txt. Additionally, you need to provide a `btstack_config.h` file in your source tree and add its location to your include path. For more details, see BlueKitchen's documentation on how to configure BTstack and the relevant Bluetooth example code in the pico-examples repository.

The CMake function `pico_btstack_make_gatt_header` can be used to run the BTstack `compile_gatt` tool to make a GATT header file from a BTstack GATT file.

2.4. Directory Structure

We have discussed libraries such as `pico_stdlib` and `hardware_gpio` above. Imagine you wanted to add some code using the RP-series microcontrollers DMA controller to the `hello_world` example in `pico-examples`. To do this you need to add a dependency on another library, `hardware_dma`, which is not included by default by `pico_stdlib` (unlike, say, `hardware_uart`).

You would change your `CMakeLists.txt` to list both `pico_stdlib` and `hardware_dma` as dependencies of the `hello_world` target (executable). (Note the line breaks are not required)

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

In your source code, you would include the DMA hardware library header as such:

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

Trying to include this header *without* listing `hardware_dma` as a dependency will fail, and this is due to how SDK files are organised into logical functional units on disk, to make it easier to add functionality in the future.

As an aside, this correspondence of `hardware_dma` → `hardware/dma.h` is the convention for *all* toplevel SDK library headers. The library is called `foo_bar` and the associated header is `foo/bar.h`. Some functions may be provided inline in the headers, others may be compiled and linked from additional `.c` files belonging to the library. Both of these require the relevant `hardware_` library to be listed as a dependency, either directly or through some higher-level bundle like `pico_stdlib`.

NOTE

Some libraries have additional headers which are located — for the above example — in `foo/bar/other.h`

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 top-level directories. This is described in the next section.

2.4.1. Locations of Files

Whilst you may be focused on building a binary to run specifically on Raspberry Pi Pico, which uses a RP2040, the 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 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 be too big or use too much RAM to fit on the device, and much of the software complexity may be non-hardware-specific.

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, along with a handful of other low-level platform libraries, all of which are specific to the RP2040.

Path	Description
src/rp2350/	This contains the hardware_regs and hardware_structs libraries mentioned earlier, along with a handful of other low-level platform libraries, all of which are specific to the RP2350.
src/rp2_common/	This contains the remaining hardware_ library implementations for individual hardware components, and pico_ libraries or library implementations that are intended specifically for RP-series microcontroller hardware. Libraries are included here even if they are RP2040 or RP2350 specific, if they are considered part of the RP-series microcontroller API proper.
src/common/	This is common code that is not specific to any hardware. This includes utilty code, headers providing hardware abstractions for functionality which are simulated in host mode (see below), along with some of the pico_ library implementations which, to the extent they use hardware, do so only through the hardware_ abstractions.
src/host/	This is a basic set of stub SDK library implementations sufficient to get simple Pico-series device applications running on your computer (Raspberry Pi OS, Linux, macOS or Windows using Cygwin or Windows Subsystem for Linux) for testing purposes. This is not intended to be a fully functional platform, 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 building for:

The value of PICO_PLATFORM determine which sets of library sources are compiled to build your program. When doing a PICO_PLATFORM=rp2040 build, you get code from common, rp2_common and rp2040; when doing a host build (PICO_PLATFROM=host), you get code from common and host.

With the advent of RP2350, there are two additional supported PICO_PLATFORM values, rp2350-arm-s for secure Arm code on RP2350, and rp2350-riscv for RISC-V on RP2350. rp2350 can also be used as a shorthand, but is expanded based on the value of PICO_DEFAULT_RP2350_PLATFORM.

TIP

Individual boards support only one of either RP2040 or RP2350. To avoid having to specify PICO_PLATFORM in addition to PICO_BOARD (see Section 2.6.1), specifying the latter can now automatically set the former.

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)
```

As a concrete example, we can list the hardware_uart directory under pico-sdk/rp2_common (you may also recall the hardware_gpio library we looked at earlier):

```
hardware_uart
├── CMakeLists.txt
├── include
│   └── hardware
│       └── uart.h
└── uart.c
```

uart.h contains function declarations and preprocessor defines for the hardware_uart library, as well as some inline functions that are expected to be particularly amenable to constant folding by the compiler. uart.c contains the implementations of more complex functions, such as calculating and setting up the divisors for a given UART baud rate.

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

2.5. Conventions for Library Functions

This section covers some common patterns you will see throughout the SDK libraries, such as conventions for function names, how errors are reported, and the approach used to efficiently configure hardware with many register fields without having unreadable numbers of function arguments.

2.5.1. Function Naming Conventions

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 SDK API is a C API and does not support function overloading.

2.5.1.1. Name prefix

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

2.5.1.2. 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 food_add_salt(), rather than oven_temperature() and food_salt().

2.5.1.3. Suffixes

2.5.1.3.1. Blocking/Non-Blocking Functions and Timeouts

Table 2. 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 2.5.2) 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 2.5.2) 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 2.5.2) after the specified number of microseconds

2.5.2. Return Codes and Error Handling

As mentioned earlier, there is a decision to be made as to whether/which functions return error codes that can be handled by the caller, and indeed whether the caller is likely to actually do something in response in an embedded environment. Also note that very often return codes are there to handle parameter checking, e.g. when asked to do something with the 27th DMA channel (when there are actually only 12).

In many cases checking for obviously invalid (likely program bug) parameters in (often inline) functions is prohibitively expensive in speed and code size terms, and therefore we need to be able to configure it on/off, which precludes return codes being returned for these exceptional cases.

The SDK follows two strategies:

1. Methods that can legitimately fail at runtime due to runtime conditions e.g. timeouts, dynamically allocated resource, can return a status which is either
   - A `bool` indicating success or not
   - An integer value which, if negative, is standard SDK negative integer return code from the `PICO_ERROR_` family (see `pico_error_code` values in pico_base) and if non-negative indicates a successful return. In the latter case the value is either `PICO_OK` (`0`) or any other positive value if the function actually needs to return something
2. Other issue like invalid parameters, or failure to allocate resources which are deemed program bugs (e.g. two libraries trying to use the same statically assigned piece of hardware) do not affect a return code (usually the functions return `void`) and must cause some sort of exceptional event.

   As of right now the exceptional event is a C `assert`, so these checks are always disabled in release builds by default. Additionally most of the calls to `assert` are disabled by default for code/size performance (even in debug builds); You can set `PARAM_ASSERTIONS_ENABLE_ALL=1` or `PARAM_ASSERTIONS_DISABLE_ALL=1` in your build to change the default across the entire SDK, or say `PARAM_ASSERTIONS_ENABLED_I2C=0/1` to explicitly specify the behaviour for the `hardware_i2c` module

   In the future we may support calling a custom function to throw an exception in C++ or other environments where stack unwinding is possible.
3. Obviously sometimes the calling code whether it be user code or another higher level function, may not want the called function to assert on bad input, in which case it is the responsibility of the caller to check the validity (there are a good number of API functions provided that help with this) of their arguments, and the caller can then choose to provide a more flexible runtime error experience.
4. Finally, some code may choose to "panic" directly if it detects an invalid state. A "panic" involves writing a message to standard output and then halting (by executing a breakpoint instruction). Panicking is a good response when it is undesirable to even attempt to continue given the current situation.

2.5.3. Use of Inline Functions

SDK libraries often contain a mixture of `static inline` functions in header files, and non-static functions in C source files. In particular, the `hardware_` libraries are likely to contain a higher proportion of inline function definitions in their headers. 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 substantially larger than the function implementation. Compilers have their own metrics to decide when to inline function implementations at their call sites, but the use of `static inline` definitions gives the compiler more freedom to do this.

This is *particularly* effective in the context of hardware register access because these functions often:

- Have relatively many parameters, which…
- …are immediately shifted and masked to combine with some register value, and…
- …are often constants known at compile time

So if the implementation of a hardware access function is inlined, the compiler can propagate the constant parameters through whatever bit manipulation and arithmetic that function may do, collapsing a complex function down to "please write this constant value to this constant address". Again, we are not forcing the compiler to do this, but the SDK consistently tries to give it freedom to do so.

The result is that there is generally no overhead using the lower-level `hardware_` functions as compared with using preprocessor macros with the `hardware_regs` definitions, and they tend to be much less error-prone.

2.5.4. Builder Pattern for Hardware Configuration APIs

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

1. Readability of code (avoid "death by parameters" where a configuration function takes a dozen integers and booleans)
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:

```
int dma_channel = 3;
dma_channel_config config = dma_get_default_channel_config(dma_channel);
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 value of `dma_channel` is known at compile time, so the compiler can replace `dma_channel` with `3` when generating code (*constant folding*). The `dma_` methods are `static inline` methods (from https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_dma/include/hardware/dma.h) meaning the implementations can be folded into your code by the compiler and, consequently, your constant parameters (like `DREQ_SPI0_RX`) are propagated though this local copy of the function implementation. The resulting code is usually smaller, and certainly faster, than the register shuffling caused by setting up a function call.

The net effect is 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;
```

It may seem counterintuitive that building up the configuration by passing a `struct` around, and committing the final result to the IO register, would be so much more compact than a series of direct register modifications using register field accessors. This is because the compiler is customarily forbidden from eliminating IO accesses (illustrated here with a `volatile` keyword), with good reason. Consequently it's easy to unwittingly generate code that repeatedly puts a value into a register and pulls it back out again, changing a few bits at a time, when we only care about the final value of the register. The configuration pattern shown here avoids this common pitfall.

> **NOTE**
>
> The 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.

2.6. Customisation and Configuration Using Preprocessor variables

The 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 libraries 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.

Chapter 5 has more details and a comprehensive list of the available preprocessor defines, what they do, and what their default values are.

Preprocessor variables may be specified in a number of ways, described in the following sections.

> **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 behaviour without the user specifying any settings (beyond those provided by the board configuration).

2.6.1. Preprocessor Variables via Board Configuration File

Many of the common configuration settings are actually related to the particular RP-series microcontroller board being used and include default pin settings for various 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 board configuration provides a header file that specifies defaults if not otherwise specified; for example https://github.com/raspberrypi/pico-sdk/blob/master/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 SDK headers as a result of setting `PICO_BOARD=my_board_name`. You can also create your own board headers.

See Section 6.2 for more full details on `PICO_BOARD` and related CMake variables.

2.6.2. Preprocessor Variables Per Binary or Library via CMake

We could modify the https://github.com/raspberrypi/pico-examples/blob/master/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
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)
```

The `target_compile_definitions` specifies preprocessor definitions that will be passed to the compiler for every source file in the target `hello_world` (which as mentioned before includes *all* of the sources for all dependent *INTERFACE* libraries). `PRIVATE` is required by CMake to specify the scope for the compile definitions. Note that all preprocessor definitions used by the SDK have a `PICO_` prefix.

2.7. 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 SDK

2.7.1. Standard Input/Output (stdio) Support

The SDK provides infrastructure for routing `stdout` and `stdin` to various hardware interfaces, which is provided by the pico_stdio library.

- A UART interface specified by a board configuration header. The default for Raspberry Pi Pico is 115200 baud on GPIO0 (TX) and GPIO1 (RX)
- A USB CDC ACM virtual serial port, using TinyUSB’s CDC support. The virtual serial device can be accessed through the RP-series microcontrollers' dedicated USB hardware interface, in Device mode
- Minimal semihosting support to direct `stdout` to an external debug host connected via the Serial Wire Debug link on the RP-series microcontroller
- Segger RTT

The support is used via the standard calls like `printf`, `puts`, `getchar`, found in the standard `<stdio.h>` header. By default, `stdout` converts bare linefeed characters to carriage return plus linefeed, for better display in a terminal emulator. This can be disabled at runtime, at build time, or the CR-LF support can be completely removed.

`stdout` is broadcast to all interfaces that are enabled, and `stdin` is collected from all interfaces which are enabled and support input. Since some of the interfaces, particularly USB, have heavy runtime and binary size cost, only the UART interface is included by default. You can add/remove interfaces for a given program at build time with e.g.

```
pico_enable_stdio_usb(target_name, 1) # enable USB CDC stdio for TARGET target_name
```

2.7.2. Printf Support

The SDK runtime packages a lightweight `printf` library by Marco Paland, provided via the `pico_printf` library.

This is a small and largely feature complete implementation, however the C library version (or no printf support) can be chosen instead via the CMake function `pico_set_printf_implementation`.

2.7.3. Runtime Initialization and Linking

Using the SDK you can simply write your simple C file with a `main()` method, and a small `CMakeLists.txt` and you can build a binary that works on your RP-series microcontroller.

You can take as much control of this process as you want, but by default, the `pico_runtime` includes these libraries:

- `pico_crt0` - the runtime entry point and default linker scripts which define memory layout
- `pico_standard_link` - configuration for link options and pulling in linker scripts
- `pico_runtime_init` - a default set of initializers to run before reaching `main`.

2.7.4. C-Library Integration

There are a variety of C libraries used by the compilers supported by the SDK. These currently include:

- `newlib`
- `picolibc`
- `llvm-libc`

These each have slightly different integration points for a bare-metal embedded applications, and the SDK runtime takes care of these via the `pico_clib_interface` library.

2.7.5. Floating-point Support

The SDK provides a highly optimized single and double-precision floating point implementation. often significantly faster than the equivalent C library versions. Both basic arithmetic, and scientific functions are provided.

On RP2040 the functions are actually implemented using support provided in the RP2040 bootrom. This means the interface from your code to the ROM floating point library has very minimal impact on your program size, certainly using dramatically less flash storage than including the standard floating point routines shipped with your compiler. The physical ROM storage on the RP-series microcontroller has single-cycle access (with a dedicated arbiter on the RP-series microcontroller busfabric), and accessing code stored here does not put pressure on the flash cache or take up space in memory, so not only are the routines fast, the rest of your code will run faster due them being resident in ROM.

On RP2350 optimized Arm versions of the single-precision floating point functions are provided which use the processors VFP floating point instructions. Optimized versions of the double-precision float point functions are provided using the RP2350's DCP (Double Coprocessor) instructions.

The SDK libraries `pico_float` and `pico_double` provide this support. This includes implementations for all the standard functions from `math.h` as well as additional functions that can be found in `pico/float.h` and `pico/double.h`.

2.7.5.1. Configuration and Alternate Implementations

There are three different floating point implementations provided

Name	Description
`default`	The default; equivalent to `pico`
`pico`	Use the fast/compact SDK/bootrom implementations
`compiler`	Use the standard compiler provided soft floating point implementations
`none`	Map all functions to a 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 `CMakeLists.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 `CMakeLists.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 `pico` 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, which is present on early Raspberry Pi Pico boards. 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.

2.7.5.1.1. NaN Propagation

The SDK implementation by default treats input *NaN*s as infinites. If you require propagation of NaN inputs to outputs and NaN outputs for domain errors, 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.

2.7.6. Hardware Divider

This section applies to RP2040 only.

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.

See Figure 1 and Figure 2 for 32-bit and 64-bit integer divider comparison.

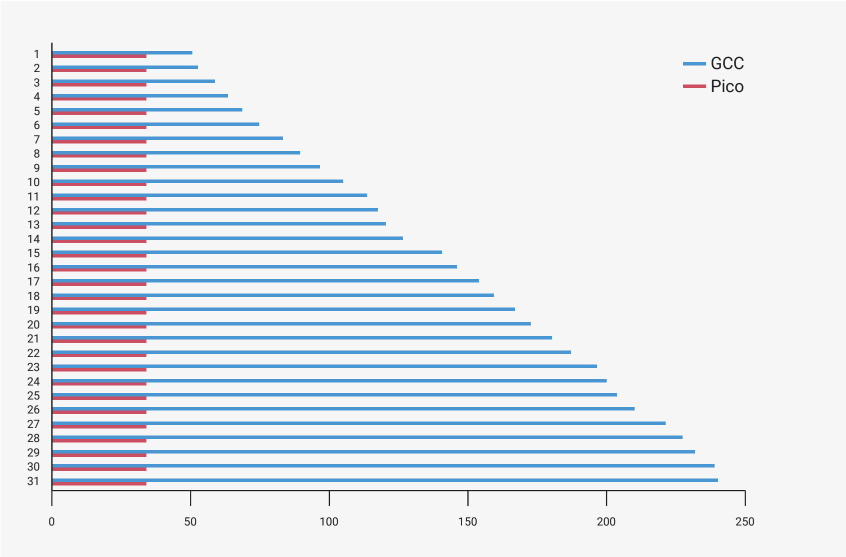


Figure 1. 32-bit divides by divider size using GCC library (blue), or the SDK library (red) with the RP2040 hardware divider.

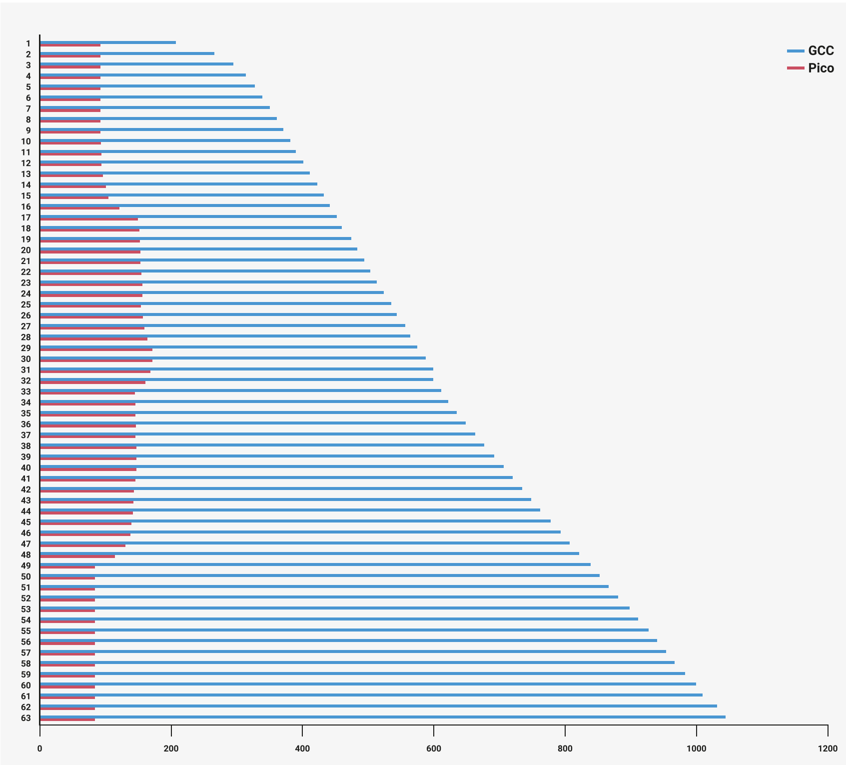


Figure 2. 64-bit divides by divider size using GCC library (blue), or the SDK library (red) with the RP2040 hardware divider.

2.8. 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.

Core 1 (the second core) is started by calling `multicore_launch_core1(some_function_pointer);` on core 0, which wakes the core from its low-power sleep state and provides it with its entry point — some function you have provided which hopefully has a descriptive name like `void core1_main() { }`. This function, as well as others such as pushing and popping data through the inter-core mailbox FIFOs, is listed under pico_multicore.

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 (mutex) from within your own code.

NOTE

> That the 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.

2.9. Using C++

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

C++ files are integrated into SDK projects in the same way as C files: listing them in your `CMakeLists.txt` file under either the `add_executable()` entry, or a separate `target_sources()` entry to append them to your target.

To save space, exception handling is disabled by default; this can be overridden with the CMake environment variable `PICO_CXX_ENABLE_EXCEPTIONS=1`. There are a handful of other C++ related `PICO_CXX` vars listed in Chapter 6.

2.10. Supporting both RP2040 and RP2350

The RP2350 supports both Cortex-M33 (Arm) and Hazard3 (RISC-V) processors. As a result the SDK now supports these processors as well as the Cortex-M0 plus processors on the RP2040.

The majority of existing source code using the SDK should compile and run unmodified, even under RISC-V, with the obvious exception of user Arm assembly code, or code interacting with the processor internals.

See Section 6.2 for details of configuring the SDK build for your particular board and RP-series microcontroller platform.

The SDK now supports the compilers listed below, although GCC is still the only *officially* supported compiler as of this SDK 2.0.0.

TIP

> If you have the correct compiler in your PATH, then compilation should just work based on your `PICO_PLATFORM` and `PICO_COMPILER` value, however for more control you can set your `PICO_TOOLCHAIN_PATH`. See Section 6.3 for full details, on configuring and finding toolchains

For Arm:

- GCC arm-none-eabi (`PICO_COMPILER=pico_arm_gcc` - the default for Arm)
  - version 6 onwards for RP2040
  - version 9 onwards for RP2350 since that is the first version that supports the Arm Cortex-M33
- LLVM Embedded Toolchain For ARM (`PICO_COMPILER=pico_arm_clang`)
  - version 14 onwards
- Pigweed LLVM. This is the vanilla build of LLVM with `llvm-libc` used by PigWeed (`PICO_COMPILER=pico_arm_clang`)
  - clang_linux-x86_64 (sha256 `e12ee0db9226f5b4a4400c5eb2c0f757d7056181b651622b5453acb00105fd87`)
  - clang_win-x86_64 (sha256 `8c41e8b507f4dfede80842f98a716cac209f552064088fa1b7f4c64a1e547534`)
  - clang_mac-x86_64 (sha256 `1d92f52609d3c1e958fd56f5e9a68ab99b2042ddcc6e90a5eb5009cf7ac4897d`)
  - clang_mac-aarch64 (sha256 `53184680db7e0043a8fba1556c7644b8f5e6c8cdffa4436a92a8e8adb0f45b8d`)

For RISC-V:

- GCC (`PICO_COMPILER=pico_arm_gcc` - the default for RISC-V)

  Only very recent versions of GCC fully support the Hazard 3 RISC-V processors, so we recommend the compilers listed below:

  - CORE-V GCC top-of-tree compilers
  - Building your own version of GCC 14 as an advanced option. For example. on current Ubuntu:

```
sudo apt-get install autoconf automake autotools-dev curl python3 python3-pip libmpc-
dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool
patchutils bc zlib1g-dev libexpat-dev ninja-build git cmake libglib2.0-dev libslirp-dev

sudo mkdir -p /opt/riscv/gcc14-rp2350-no-zcmp

sudo chown -R $(whoami) /opt/riscv/gcc14-rp2350-no-zcmp

git clone https://github.com/riscv/riscv-gnu-toolchain

cd riscv-gnu-toolchain

git clone https://github.com/gcc-mirror/gcc gcc-14 -b releases/gcc-14

./configure --prefix=/opt/riscv/gcc14-rp2350-no-zcmp --with
-arch=rv32ima_zicsr_zifencei_zba_zbb_zbs_zbkb_zca_zcb --with-abi=ilp32 --with-multilib
-generator="rv32ima_zicsr_zifencei_zba_zbb_zbs_zbkb_zca_zcb-ilp32--
;rv32imac_zicsr_zifencei_zba_zbb_zbs_zbkb-ilp32--" --with-gcc-src=`pwd`/gcc-14

make -j$(nproc)
```

2.11. Next Steps

This has been quite a deep dive. If you've somehow made it through this chapter *without* building any software, now would be a perfect time to divert to the **Getting started with Raspberry Pi Pico-series** book, which has detailed instructions on connecting to your RP-series microcontroller board and loading an application built with the SDK.

Chapter 3 gives some background on RP-series microcontrollers' unique Programmable I/O subsystem, and walks through building some applications which use PIO to talk to external hardware.

Chapter 4 is a comprehensive listing of the SDK APIs. The APIs are listed according to groups of related functionality (e.g. low-level hardware access).

Chapter 3. Using programmable I/O (PIO)

3.1. What is Programmable I/O (PIO)?

Programmable I/O (PIO) is a new piece of hardware developed for RP-series microcontrollers. It allows you to create new types of (or additional) hardware interfaces on your RP-series microcontroller based device. If you've looked at fixed peripherals on a microcontroller, and thought "I want to add 4 more UARTs", or "I'd like to output DPI video", or even "I need to communicate with this cursed serial device I found on AliExpress, but no machine has hardware support", then you will have fun with this chapter.

PIO hardware is described extensively in chapter 11 of the **RP2350 Datasheet**. This is a companion to that text, focussing on how, when and why to use PIO in your software. To start, we're going to spend a while discussing why I/O is hard, what the current options are, and what PIO does differently, before diving into some software tutorials. We will also try to illuminate some of the more important parts of the hardware along the way, but will defer to the datasheet for full explanations.

TIP

> You can skip to the first software tutorial if you'd prefer to dive straight in.

3.1.1. 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.

3.1.2. I/O Using dedicated hardware on your PC

Traditionally, on your desktop or laptop computer, you have one option for hardware interfacing. Your computer has high speed USB ports, HDMI outputs, PCIe 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 graphics card, hard drive 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. These interfaces are purpose-built, and if you have them, you should use them.

3.1.3. 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, I2S, CAN bus and more over *general purpose I/O* pins (GPIOs). Like USB controllers (also found on some microcontrollers, including the RP-series microcontroller on Pico-series), I2C and SPI are general purpose buses which connect to a wide variety of external hardware, using the same piece of on-chip hardware. This includes sensors, external flash, EEPROM and SRAM memories, GPIO expanders, and more, all of them widely and cheaply available. Even HDMI uses I2C to communicate video timings between Source and Sink, and there is probably a microcontroller *embedded* in your TV to handle this.

These protocols are simpler to integrate into very low-cost *devices* (i.e. not the host), due to their relative simplicity and modest speed. This is important for chips with mostly analogue or high-power circuitry: the silicon fabrication techniques used for these chips do not lend themselves to high speed or gate count, so if your switchmode power supply controller has some serial configuration interface, it is likely to be something like I2C. The number of traces routed on the circuit board, the number of pins required on the device package, and the PCB technology required to maintain signal integrity are also factors in the choice of these protocols. A microcontroller needs to communicate with these devices to be part of a larger *embedded system*.

This is all very well, but the area taken up by these individual serial peripherals, and the associated cost, often leaves you with a limited menu. You may end up paying for a bunch of stuff you don't need, and find yourself without enough of what you really want. 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).

3.1.4. I/O Using software control of GPIOs (*"bit-banging"*)

The third option on your Raspberry Pi or microcontroller — any system with GPIOs which the processor(s) can access easily — 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, and the approximate signalling speeds involved:

Table 3. Types of hardware

Interface Speed	Interface
1-10Hz	Push buttons, indicator LEDs
300Hz	HDMI CEC
10-100kHz	Temperature sensors (DHT11), one-wire serial
<100kHz	I2C Standard mode
22-100+kHz	PCM audio
300+kHz	PWM audio
400-1200kHz	WS2812 LED string
10-3000kHz	UART serial
12MHz	USB Full Speed
1-100MHz	SPI
20-300MHz	DPI/VGA video
480MHz	USB High Speed
10-4000MHz	Ethernet LAN
12-4000MHz	SD card
250-20000MHz	HDMI/DVI video

"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 back then. As processors have become faster in terms of overwhelming number-crunching brute force, the layers of software and hardware between the processor and the outside world have also grown in number and size. In response to the growing distance between processors and memory, PC-class processors keep many hundreds of instructions in-flight on a single core at once, which has drawbacks when trying to switch rapidly between hard real time tasks. However, IRQ-based bitbanging can be an effective strategy on simpler embedded systems.

Above certain speeds — say a factor of 1000 below the processor clock speed — IRQs become impractical, in part due to the timing uncertainty of actually *entering* an interrupt handler. The alternative when *"bit-banging"* is to sit the processor in a carefully timed loop, often painstakingly written 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 thousands of GitHub repositories are dedicated to the task of doing such things (a large proportion of them for LED strings).

Additionally of course, your processor is now busy doing the *"bit-banging"*, and cannot be used for other tasks. If your processor is interrupted even for a few microseconds to attend to one of the hard peripherals it is also responsible for, this can be fatal to the timing of any bit-banged protocol. The greater the ratio between protocol speed and processor speed, the more cycles your processor will spend uselessly idling in between GPIO accesses. Whilst it is eminently possible to drive a 115200 baud UART output using only software, this has a cost of >10,000 cycles per byte if the processor is running at 133MHz, which may be poor investment of those cycles.

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 not much you can hope to do. The main case where software GPIO access is the *best* choice is LEDs and push buttons.

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).

3.1.5. Programmable I/O Hardware using FPGAs and CPLDs

A *field-programmable gate array* (FPGA), or its smaller cousin, the *complex programmable logic device* (CPLD), is in many ways the perfect solution for tailor-made I/O requirements, whether that entails an unusual type or unusual mixture of interfaces. FPGAs are chips with a configurable logic fabric — effectively a sea of gates and flipflops, some other special digital function blocks, and a routing fabric to connect them — which offer the same level of design flexibility available to chip designers. This brings with it all the advantages of dedicated I/O hardware:

- Absolute precision of protocol timing (within limitations of your clock source)
- Capable of very high I/O throughput
- Offload simple, repetitive calculations that are part of the I/O standard (checksums)
- Present a simpler interface to host software; abstract away details of the protocol, and handle these details internally.

The main drawback of FPGAs in embedded systems is their cost. They also present a very unfamiliar programming model to those well-versed in embedded software: you are not programming at all, but rather designing digital hardware. One you have your FPGA you will still need some other processing element in your system to run control software, unless you are using an FPGA expensive enough to either fit a soft CPU core, or contain a hardened CPU core alongside the FPGA fabric.

eFPGAs (embedded FPGAs) are available in some microcontrollers: a slice of FPGA logic fabric integrated into a more conventional microcontroller, usually with access to some GPIOs, and accessible over the system bus. These are attractive from a system integration point of view, but have a significant area overhead compared with the usual serial peripherals found on a microcontroller, so either increase the cost and power dissipation, or are very limited in size. The issue of programming complexity still remains in eFPGA-equipped systems.

3.1.6. Programmable I/O Hardware using PIO

The PIO subsystem on RP-series microcontrollers allows you to write small, simple programs for what are called *PIO state machines*, of which RP2040 has eight split across two PIO *instances*, and RP2350 has twelve split across three PIO instances. A state machine is responsible for setting and reading one or more GPIOs, buffering data to or from the processor (or the RP-series microcontrollers' ultra-fast DMA subsystem), and notifying the processor, via IRQ or polling, when data or attention is needed.

These programs operate with cycle accuracy at up to system clock speed (or the program clocks can be divided down to run at slower speeds for less frisky protocols).

PIO state machines are much more compact than the general-purpose processors on RP2040 and RP2350. In fact, they are similar in size (and therefore cost) to a standard SPI peripheral, such as the PL022 SPI also found on RP-series microcontrollers, because much of their area is spent on components which are common to all serial peripherals, like FIFOs, shift registers and clock dividers. The instruction set is small and regular, so not much silicon is spent on decoding the instructions. There is no need to feel guilty about dedicating a state machine solely to a single I/O task, since you have several!

In spite of this, a PIO state machine gets a lot *more* done in one cycle than a Cortex-M0+ when it comes to I/O: for example, sampling a GPIO value, toggling a clock signal and pushing to a FIFO all in one cycle, every cycle. The trade-off is that a PIO state machine is not remotely capable of running general purpose software. As we shall see though, programming a PIO state machine is quite familiar for anyone who has written assembly code before, and the small instruction set should be fairly quick to pick up for those who haven't.

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. 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 a RP-series microcontroller, 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.

3.2. Getting started with PIO

It is possible to write PIO programs both within the C++ SDK and directly from MicroPython.

Additionally the future intent is to add APIs to trivially have new UARTs, PWM channels etc created for you, using a menu of pre-written PIO programs, but for now you'll have to follow along with example code and do that yourself.

3.2.1. A First PIO Application

Before getting into all of the fine details of the PIO assembly language, we should take the time to look at a small but complete application which:

1. Loads a program into a PIO's instruction memory
2. Sets up a PIO state machine to run the program
3. Interacts with the state machine once it is running.

The main ingredients in this recipe are:

- A PIO program
- Some software, written in C, to run the whole show
- A CMake file describing how these two are combined into a program image to load onto a RP-series microcontroller based development board

TIP

The code listings in this section are all part of a complete application on GitHub, which you can build and run. Just click the link above each listing to go to the source. In this section we are looking at the `pio/hello_pio` example in `pico-examples`. You might choose to build this application and run it, to see what it does, before reading through this section.

NOTE

The focus here is on the main moving parts required to use a PIO program, not so much on the PIO program itself. This is a lot to take in, so we will stay high-level in this example, and dig in deeper on the next one.

3.2.1.1. PIO Program

This is our first PIO program listing. It's written in PIO assembly language.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/hello_pio/hello.pio Lines 8 - 16

```
.program hello

; Repeatedly get one word of data from the TX FIFO, stalling when the FIFO is
; empty. Write the least significant bit to the OUT pin group.

loop:
    pull
    out pins, 1
    jmp loop
```

The `pull` instruction takes one data item from the transmit FIFO buffer, and places it in the *output shift register* (OSR). Data moves from the FIFO to the OSR one word (32 bits) at a time. The OSR is able to *shift* this data out, one or more bits at a time, to further destinations, using an `out` instruction.

> **FIFOs?**
>
> FIFOs are data queues, implemented in hardware. Each state machine has two FIFOs, between the state machine and the system bus, for data travelling out of (TX) and into (RX) the chip. Their name (*first in, first out*) comes from the fact that data appears at the FIFO's output in the same order as it was presented to the FIFO's input.

The `out` instruction here takes one bit from the data we just `pull`-ed from the FIFO, and writes that data to some pins. We will see later how to decide which pins these are.

The `jmp` instruction jumps back to the `loop:` label, so that the program repeats indefinitely. So, to sum up the function of this program: repeatedly take one data item from a FIFO, take one bit from this data item, and write it to a pin.

Our `.pio` file also contains a helper function to set up a PIO state machine for correct execution of this program:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/hello_pio/hello.pio Lines 19 - 34

```
static inline void hello_program_init(PIO pio, uint sm, uint offset, uint pin) {
    pio_sm_config c = hello_program_get_default_config(offset);

    // Map the state machine's OUT pin group to one pin, namely the `pin`
    // parameter to this function.
    sm_config_set_out_pins(&c, pin, 1);
    // Set this pin's GPIO function (connect PIO to the pad)
    pio_gpio_init(pio, pin);
```

```
    // Set the pin direction to output at the PIO
    pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);

    // Load our configuration, and jump to the start of the program
    pio_sm_init(pio, sm, offset, &c);
    // Set the state machine running
    pio_sm_set_enabled(pio, sm, true);
}
```

Here the main thing to set up is the GPIO we intend to output our data to. There are three things to consider here:

1. The state machine needs to be told which GPIO or GPIOs to output to. There are four different pin groups which are used by different instructions in different situations; here we are using the out pin group, because we are just using an `out` instruction.
2. The *GPIO* also needs to be told that PIO is in control of it (GPIO function select)
3. If we are using the pin for output only, we need to make sure that PIO is driving the *output enable* line high. PIO can drive this line up and down programmatically using e.g. an `out pindirs` instruction, but here we are setting it up before starting the program.

3.2.1.2. C Program

PIO won't do anything until it's been configured properly, so we need some software to do that. The PIO file we just looked at — `hello.pio` — is converted automatically (we will see later how) into a header containing our assembled PIO program binary, any helper functions we included in the file, and some useful information about the program. We include this as `hello.pio.h`.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/hello_pio/hello.c

```
/**
 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */

#include <stdio.h>

#include "pico/stdlib.h"
#include "hardware/pio.h"
// Our assembled program:
#include "hello.pio.h"

// This example uses the default led pin
// You can change this by defining HELLO_PIO_LED_PIN to use a different gpio
#if !defined HELLO_PIO_LED_PIN && defined PICO_DEFAULT_LED_PIN
#define HELLO_PIO_LED_PIN PICO_DEFAULT_LED_PIN
#endif

// Check the pin is compatible with the platform
#if HELLO_PIO_LED_PIN >= NUM_BANK0_GPIOS
#error Attempting to use a pin>=32 on a platform that does not support it
#endif

int main() {
#ifndef HELLO_PIO_LED_PIN
#warning pio/hello_pio example requires a board with a regular LED
#else
    PIO pio;
    uint sm;
    uint offset;
```

```

    setup_default_uart();

    // This will find a free pio and state machine for our program and load it for us
    // We use pio_claim_free_sm_and_add_program_for_gpio_range so we can address gpios >= 32 if needed and supported by the hardware
    bool success = pio_claim_free_sm_and_add_program_for_gpio_range(&hello_program, &pio, &sm, &offset, HELLO_PIO_LED_PIN, 1, true);
    hard_assert(success);

    // Configure it to run our program, and start it, using the
    // helper function we included in our .pio file.
    printf("Using gpio %d\n", HELLO_PIO_LED_PIN);
    hello_program_init(pio, sm, offset, HELLO_PIO_LED_PIN);

    // The state machine is now running. Any value we push to its TX FIFO will
    // appear on the LED pin.
    // press a key to exit
    while (getchar_timeout_us(0) == PICO_ERROR_TIMEOUT) {
        // Blink
        pio_sm_put_blocking(pio, sm, 1);
        sleep_ms(500);
        // Blonk
        pio_sm_put_blocking(pio, sm, 0);
        sleep_ms(500);
    }

    // This will free resources and unload our program
    pio_remove_program_and_unclaim_sm(&hello_program, pio, sm, offset);
#endif
}
```

You might recall that RP2040 has two PIO blocks, each of them with four state machines (the {chipname_rp2350 has three PIO blocks each with four state machines). Each PIO block has a 32-slot instruction memory which is visible to the four state machines in the block. We need to load our program into this instruction memory before any of our state machines can run the program. The function `pio_add_program()` finds free space for our program in a given PIO's instruction memory, and loads it.

> **32 Instructions?**
>
> This may not sound like a lot, but the PIO instruction set can be *very* dense once you fully explore its features. A perfectly serviceable UART transmit program can be implemented in four instructions, as shown in the `pio/uart_tx` example in `pico-examples`. There are also a couple of ways for a state machine to execute instructions from other sources — like directly from the FIFOs — which you can read all about in the **RP2350 Datasheet**.

Once the program is loaded, we find a free state machine and tell it to run our program. There is nothing stopping us from ordering multiple state machines to run the same program. Likewise, we could instruct each state machine to run a *different* program, provided they all fit into the instruction memory at once.

We're configuring this state machine to output its data to the LED on your Pico-series device. If you have already built and run the program, you probably noticed this already!

At this point, the state machine is running autonomously. The state machine will immediately *stall*, because it is waiting for data in the TX FIFO, and we haven't provided any. The processor can push data directly into the state machine's TX FIFO using the `pio_sm_put_blocking()` function. (`_blocking` because this function stalls the processor when the TX FIFO is full.) Writing a 1 will turn the LED on, and writing a 0 will turn the LED off.

3.2.1.3. CMake File

We have two lovely text files sat on our computer, with names ending with `.pio` and `.c`, but they aren't doing us much good there. A CMake file describes how these are built into a binary suitable for loading onto your Pico-series device or other RP-series microcontroller based board.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/hello_pio/CMakeLists.txt

```
add_executable(hello_pio)

pico_generate_pio_header(hello_pio ${CMAKE_CURRENT_LIST_DIR}/hello.pio)

target_sources(hello_pio PRIVATE hello.c)

target_link_libraries(hello_pio PRIVATE
        pico_stdlib
        hardware_pio
        )

# Pass cmake -DHELLO_PIO_LED_PIN=x, where x is the pin you want to use
if(HELLO_PIO_LED_PIN)
        target_compile_definitions(hello_pio PRIVATE
                HELLO_PIO_LED_PIN=${HELLO_PIO_LED_PIN}
        )
endif()

pico_add_extra_outputs(hello_pio)

# add url via pico_set_program_url
example_auto_set_url(hello_pio)
```

- `add_executable()`: Declare that we are building a program called `hello_pio`
- `pico_generate_pio_header()`: Declare that we have a PIO program, `hello.pio`, which we want to be built into a C header for use with our program
- `target_sources()`: List the source code files for our `hello_pio` program. In this case, just one C file.
- `target_link_libraries()`: Make sure that our program is built with the PIO hardware API, so we can call functions like `pio_add_program()` in our C file.
- `pico_add_extra_outputs()`: By default we just get an `.elf` file as the build output of our app. Here we declare we also want extra build formats, like a `.uf2` file which can be dragged and dropped directly onto a Pico-series device attached over USB.

Assuming you already have `pico-examples` and the SDK installed on your machine, you can run

```
$ mkdir build
$ cd build
$ cmake ..
$ make hello_pio
```

To build this program.

3.2.2. A Real Example: WS2812 LEDs

The WS2812 LED (sometimes sold as NeoPixel) is an addressable RGB LED. In other words, it's an LED where the red, green and blue components of the light can be individually controlled, and it can be connected in such a way that many

WS2812 LEDs can be controlled individually, with only a single control input. Each LED has a pair of power supply terminals, a serial data input, and a serial data output.

When serial data is presented at the LED's input, it takes the first three bytes for itself (red, green, blue) and the remainder is passed along to its serial data output. Often these LEDs are connected in a single long chain, each LED connected to a common power supply, and each LED's data output connected through to the next LED's input. A long burst of serial data to the first in the chain (the one with its data input unconnected) will deposit three bytes of RGB data in each LED, so their colour and brightness can be individually programmed.

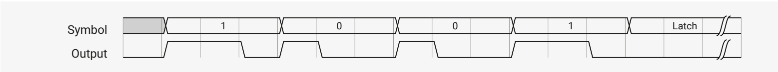


Figure 3. WS2812 line format. Wide positive pulse for 1, narrow positive pulse for 0, very long negative pulse for latch enable

Unfortunately the LEDs receive and retransmit serial data in quite an unusual format. Each bit is transferred as a positive pulse, and the width of the pulse determines whether it is a `1` or a `0` bit. There is a family of WS2812-like LEDs available, which often have slightly different timings, and demand precision. It is possible to bit-bang this protocol, or to write canned bit patterns into some generic serial peripheral like SPI or I2S to get firmer guarantees on the timing, but there is still some software complexity and cost associated with generating the bit patterns.

Ideally we would like to have all of our CPU cycles available to generate colour patterns to put on the lights, or to handle any other responsibilities the processor may have in the *embedded system* the LEDs are connected to.

TIP

Once more, this section is going to discuss a real, complete program, that you can build and run on your Pico-series device. Follow the links above the program listings if you'd prefer to build the program yourself and run it, before going through it in detail. This section explores the `pio/ws2812` example in `pico-examples`.

3.2.2.1. PIO Program

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/ws2812.pio Lines 8 - 31

```
.program ws2812
.side_set 1

; The following constants are selected for broad compatibility with WS2812,
; WS2812B, and SK6812 LEDs. Other constants may support higher bandwidths for
; specific LEDs, such as (7,10,8) for WS2812B LEDs.

.define public T1 3
.define public T2 3
.define public T3 4

.lang_opt python sideset_init = pico.PIO.OUT_HIGH
.lang_opt python out_init     = pico.PIO.OUT_HIGH
.lang_opt python out_shiftdir = 1

.wrap_target
bitloop:
    out x, 1       side 0 [T3 - 1] ; Side-set still takes place when instruction stalls
    jmp !x do_zero side 1 [T1 - 1] ; Branch on the bit we shifted out. Positive pulse
do_one:
    jmp  bitloop   side 1 [T2 - 1] ; Continue driving high, for a long pulse
do_zero:
    nop            side 0 [T2 - 1] ; Or drive low, for a short pulse
.wrap
```

The previous example was a bit of a whistle-stop tour of the anatomy of a PIO-based application. This time we will dissect the code line-by-line. The first line tells the assembler that we are defining a program named ws2812:

```
.program ws2812
```

We can have multiple programs in one `.pio` file (and you will see this if you click the GitHub link above the main program listing), and each of these will have its own `.program` directive with a different name. The assembler will go through each program in turn, and all the assembled programs will appear in the output file.

Each PIO instruction is 16 bits in size. Generally, 5 of those bits in each instruction are used for the “delay” which is usually 0 to 31 cycles (after the instruction completes and before moving to the next instruction). If you have read the PIO chapter of the **RP2350 Datasheet**, you may have already know that these 5 bits can be used for a different purpose:

```
.side_set 1
```

This directive `.side_set 1` says we’re *stealing* one of those delay bits to use for "side-set". The state machine will use this bit to drive the values of some pins, once per 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, to fit into the shared instruction memory.

Note that stealing one bit has left our delay range from 0-15 (4 bits), 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 would mean the side-set is optional (at the cost of another bit to say *whether* the instruction does a side-set), we have to specify a side-set value for *every* instruction in the program. This is the `side N` you will see on each instruction in the listing.

```
.define public T1 2
.define public T2 5
.define public T3 3
```

`.define` lets you declare constants. The `public` keyword means that the assembler will also write out the value of the define in the output file for use by other software: in the context of the SDK, this is a `#define`. We are going to use `T1`, `T2` and `T3` in calculating the delay cycles on each instruction.

```
.lang_opt python
```

This is used to specify some PIO hardware defaults as used by the MicroPython PIO library. We don’t need to worry about them in the context of SDK applications.

```
.wrap_target
```

We’ll ignore this for now, and come back to it later, when we meet its friend `.wrap`.

```
bitloop:
```

This is a label. A label tells the assembler that this point in your code is interesting to you, and you want to refer to it later by name. Labels are mainly used with `jmp` instructions.

```
out x, 1       side 0 [T3 - 1] ; Side-set still takes place when instruction stalls
```

Finally we reach a line with a PIO instruction. There is a lot to see here.

- This is an `out` instruction. `out` takes some bits from the *output shift register* (OSR), and writes them somewhere else. In this case, the OSR will contain pixel data destined for our LEDs.
- `[T3 - 1]` is the number of delay cycles (T3 minus 1). `T3` is a constant we defined earlier.
- `x` (one of two scratch registers; the other imaginatively called `y`) is the destination of the write data. State machines use their scratch registers to hold and compare temporary data.
- `side 0`: Drive low (`0`) the pin configured for side-set.
- Everything after the `;` character is a *comment*. Comments are ignored by the assembler: they are just notes for humans to read.

> **Output Shift Register**
>
> The OSR is a staging area for data entering the state machine through the TX FIFO. Data is pulled from the TX FIFO into the OSR one 32-bit chunk at a time. When an `out` instruction is executed, the OSR can break this data into smaller pieces by *shifting* to the left or right, and sending the bits that drop off the end to one of a handful of different destinations, such as the pins.
>
> The amount of data to be shifted is encoded by the `out` instruction, and the *direction* of the shift (left or right) is configured ahead of time. For full details and diagrams, see the **RP2350 Datasheet**.

So, the state machine will do the following operations when it executes this instruction:

1. Set 0 on the side-set pin (this happens even if the instruction stalls because no data is available in the OSR)
2. Shift one bit out of the OSR into the x register. The value of the x register will be either 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 that when we say cycle, we mean state machine execution cycles: a state machine can be made to execute at a slower rate than the system clock, by configuring its *clock divider*.

Let's look at the next instruction in the program.

```
jmp !x do_zero side 1 [T1 - 1] ; Branch on the bit we shifted out. Positive pulse
```

1. `side 1` on the side-set pin (this is the leading edge of our pulse)
2. If `x == 0` then go to the instruction labelled `do_zero`, otherwise continue on sequentially to the next instruction
3. We delay `T1 - 1` after the instruction (whether the branch is taken or not)

Let's look at what our output pin has done so far in the program.

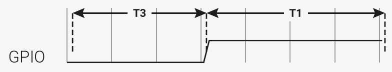


Figure 4. The state machine drives the line low for time T1 as it shifts out one data bit from the OSR, and then high for time T2 whilst branching on the value of the bit.

The pin has been low for time T3, and high for time T1. If the x register is 1 (remember this contains our 1 bit of pixel data) then we will fall through to the instruction labelled `do_one:`

```
do_one:
    jmp  bitloop   side 1 [T2 - 1] ; Continue driving high, for a long pulse
```

On this side of the branch we do the following:

1. `side 1` on the side-set pin (continue the pulse)
2. `jmp` unconditionally back to `bitloop` (the label we defined earlier, at the top of the program); the state machine is done with this data bit, and will get another from its OSR
3. Delay for `T2 - 1` cycles after the instruction

The waveform at our output pin now looks like this:

Figure 5. On a one data bit, the line is driven low for time T3, high for time T1, then high for an additional time T2

This accounts for the case where we shifted a `1` data bit into the x register. For a `0` bit, we will have jumped over the last instruction we looked at, to the instruction labelled `do_zero`:

```
do_zero:
    nop            side 0 [T2 - 1] ; Or drive low, for a short pulse
```

1. `side 0` on the side-set pin (the trailing edge of our pulse)
2. `nop` means no operation. We don't have anything else we particularly want to do, so waste a cycle
3. The instruction takes T2 cycles in total

For the `x == 0` case, we get this on our output pin:

Figure 6. On a zero data bit, the line is driven low for time T3, high for time T1, then low again for time T1

The final line of our program is this:

```
.wrap
```

This matches with the `.wrap_target` directive at the top of the program. Wrapping is a hardware feature of the state machine which behaves like a wormhole: you go in through the `.wrap` statement and appear at the `.wrap_target` *zero* cycles later, unless the `.wrap` is preceded immediately by a `jmp` whose condition is true. This is important for getting precise timing with programs that must run quickly, and often also saves you a slot in the instruction memory.

> **TIP**
>
> Often an explicit `.wrap_target`/`.wrap` pair is not necessary, because the default configuration produced by `pioasm` has an implicit wrap from the end of the program back to the beginning, if you didn't specify one.

NOPs

NOP, or no operation, means precisely that: do nothing! You may notice there is no `nop` instruction defined in the instruction set reference: `nop` is really a synonym for `mov y, y` in PIO assembly.

Why did we insert a `nop` in this example when we could have `jmp`-ed? Good question! It's a dramatic device we contrived so we could discuss `nop` and `.wrap`. Writing documentation is hard. In general, though, `nop` is useful when you need to perform a side-set and have nothing else to do, or you need a very slightly longer delay than is available on a single instruction.

It is hopefully becoming clear why our timings T1, T2, T3 are numbered this way, because what the LED string sees really is one of these two cases:

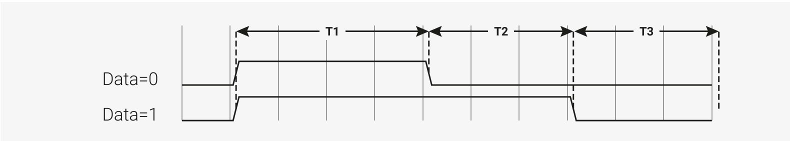


Figure 7. The line is initially low in the idle (latch) state, and the LED is waiting for the first rising edge. It sees our pulse timings in the order T1-T2-T3, until the very last T3, where it sees a much longer negative period once the state machine runs out of data.

This should look familiar if you refer back to Figure 3.

After thoroughly dissecting our program, and hopefully being satisfied that it will repeatedly send one well-formed data bit to a string of WS2812 LEDs, we're left with a question: where is the data coming from? This is more thoroughly explained in the **RP2350 Datasheet**, but the data that we are shifting out from the OSR came from the state machine's TX FIFO. The TX FIFO is a data buffer between the state machine and the rest of RP-series microcontroller, filled either via direct poking from the CPU, or by the system DMA, which is much faster.

The `out` instruction shifts data out from the OSR, and zeroes are shifted in from the other end to fill the vacuum. Because the OSR is 32 bits wide, you will start getting zeroes once you have shifted out a total of 32 bits. There is a `pull` instruction which explicitly takes data from the TX FIFO and put it in the OSR (stalling the state machine if the FIFO is empty).

However, in the majority of cases it is simpler to configure *autopull*, a mode where the state machine automatically refills the OSR from the TX FIFO (an automatic `pull`) when a configured number of bits have been shifted out. Autopull happens in the background, in parallel with whatever else the state machine may be up to (in other words it has a cost of zero cycles). We'll see how this is configured in the next section.

3.2.2.2. State Machine Configuration

When we run `pioasm` on the `.pio` file we have been looking at, and ask it to spit out 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 state machine based on user parameters, and the directives in the actual PIO program (namely the `.side_set` and `.wrap` in this case).

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 within the `.pio` file.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/ws2812.pio Lines 36 - 52

```
36 static inline void ws2812_program_init(PIO pio, uint sm, uint offset, uint pin, float freq,
   bool rgbw) {
37
```

```
    pio_gpio_init(pio, pin);
    pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);

    pio_sm_config c = ws2812_program_get_default_config(offset);
    sm_config_set_sideset_pins(&c, pin);
    sm_config_set_out_shift(&c, false, true, rgbw ? 32 : 24);
    sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);

    int cycles_per_bit = ws2812_T1 + ws2812_T2 + ws2812_T3;
    float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit);
    sm_config_set_clkdiv(&c, div);

    pio_sm_init(pio, sm, offset, &c);
    pio_sm_set_enabled(pio, sm, true);
}
```

In this case we are passing through code for the SDK, as requested by this line you will see if you click the link on the above listing to see the context:

```
% c-sdk {
```

We have here a function `ws2812_program_init` which is provided to help the user to instantiate an instance of the LED driver program, based on a handful of parameters:

`pio`
: Which of the PIO instances we are dealing with

`sm`
: Which state machine on that PIO we want to configure to run the WS2812 program

`offset`
: Where the PIO program was loaded in PIO's 5-bit program address space

`pin`
: which GPIO pin our WS2812 LED chain is connected to

`freq`
: The frequency (or rather baud rate) we want to output data at.

`rgbw`
: True if we are using 4-colour LEDs (red, green, blue, white) rather than the usual 3.

Such that:

- `pio_gpio_init(pio, pin);` Configure a GPIO for use by PIO. (Set the GPIO function select.)
- `pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);` Sets the PIO pin direction of 1 pin starting at pin number `pin` to out
- `pio_sm_config c = ws2812_program_default_config(offset);` Get the default configuration using the generated function for this program (this includes things like the `.wrap` and `.side_set` configurations from the program). We'll modify this configuration before loading it into the state machine.
- `sm_config_set_sideset_pins(&c, pin);` Sets the side-set to write to pins starting at pin `pin` (we 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_set_out_shift(&c, false, true, rgbw ? 32 : 24);` False for shift_to_right (i.e. we want to shift out MSB first). True for autopull. 32 or 24 for the number of bits for the autopull threshold, i.e. the point at which the state machine triggers a refill of the OSR, depending on whether the LEDs are RGB or RGBW.

- `int cycles_per_bit = ws2812_T1 + ws2812_T2 + ws2812_T3;` This is the total number of execution cycles to output a single bit. Here we see the benefit of `.define public`; we can use the T1 - T3 values in our code.
- `float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit); sm_config_clkdiv(&c, div);` Slow the state machine's execution down, based on the system clock speed and the number of execution cycles required per WS2812 data bit, so that we achieve the correct bit rate.
- `pio_sm_init(pio, sm, offset, &c);` Load our configuration into the state machine, and go to the start address (`offset`)
- `pio_sm_set_enabled(pio, sm, true);` And make it go now!

At this point the program will be stuck on the first `out` waiting for data. This is because we have autopull enabled, the OSR is initially empty, and there is no data to be pulled. The state machine refuses to continue until the first piece of data arrives in the FIFO.

As an aside, this last point sheds some light on the slightly cryptic comment at the start of the PIO program:

```
    out x, 1       side 0 [T3 - 1] ; Side-set still takes place when instruction stalls
```

This comment is giving us an important piece of context. We stall on this instruction initially, before the first data is added, and also every time we finish sending the last piece of data at the end of a long serial burst. When a state machine stalls, it does not continue to the next instruction, rather it will reattempt the current instruction on the next divided clock cycle. However, side-set still takes place. This works in our favour here, because we consequently always return the line to the idle (low) state when we stall.

3.2.2.3. C Program

The companion to the `.pio` file we've looked at is a `.c` file which drives some interesting colour patterns out onto a string of LEDs. We'll just look at the parts that are directly relevant to PIO.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/ws2812.c Lines 43 - 45

```c
43 static inline void put_pixel(PIO pio, uint sm, uint32_t pixel_grb) {
44     pio_sm_put_blocking(pio, sm, pixel_grb << 8u);
45 }
```

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/ws2812.c Lines 47 - 52

```c
47 static inline uint32_t urgb_u32(uint8_t r, uint8_t g, uint8_t b) {
48     return
49             ((uint32_t) (r) << 8) |
50             ((uint32_t) (g) << 16) |
51             (uint32_t) (b);
52 }
```

Here we are writing 32-bit values into the FIFO, one at a time, directly from the CPU. `pio_sm_put_blocking` is a helper method that waits until there is room in the FIFO before pushing your data.

You'll notice the `<< 8` in `put_pixel()`: remember we are shifting out starting with the MSB, so we want the 24-bit colour values at the top. This works fine for WGBR too, just that the W is always 0.

This program has a handful of colour patterns, which call our `put_pixel` helper above to output a sequence of pixel values:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/ws2812.c Lines 76 - 81

```
void pattern_random(PIO pio, uint sm, uint len, uint t) {
    if (t % 8)
        return;
    for (uint i = 0; i < len; ++i)
        put_pixel(pio, sm, rand());
}
```

The main function loads the program onto a PIO, configures a state machine for 800 kbaud WS2812 transmission, and then starts cycling through the colour patterns randomly.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/ws2812.c Lines 110 - 143

```
int main() {
    //set_sys_clock_48();
    stdio_init_all();
    printf("WS2812 Smoke Test, using pin %d\n", WS2812_PIN);

    // todo get free sm
    PIO pio;
    uint sm;
    uint offset;

    // This will find a free pio and state machine for our program and load it for us
    // We use pio_claim_free_sm_and_add_program_for_gpio_range (for_gpio_range variant)
    // so we will get a PIO instance suitable for addressing gpios >= 32 if needed and
supported by the hardware
    bool success = pio_claim_free_sm_and_add_program_for_gpio_range(&ws2812_program, &pio,
&sm, &offset, WS2812_PIN, 1, true);
    hard_assert(success);

    ws2812_program_init(pio, sm, offset, WS2812_PIN, 800000, IS_RGBW);

    int t = 0;
    while (1) {
        int pat = rand() % count_of(pattern_table);
        int dir = (rand() >> 30) & 1 ? 1 : -1;
        puts(pattern_table[pat].name);
        puts(dir == 1 ? "(forward)" : "(backward)");
        for (int i = 0; i < 1000; ++i) {
            pattern_table[pat].pat(pio, sm, NUM_PIXELS, t);
            sleep_ms(10);
            t += dir;
        }
    }

    // This will free resources and unload our program
    pio_remove_program_and_unclaim_sm(&ws2812_program, pio, sm, offset);
}
```

3.2.3. PIO and DMA (A Logic Analyser)

So far we have looked at writing data to PIO directly from the processor. This often leads to the processor spinning its wheels waiting for room in a FIFO to make a data transfer, which is not a good investment of its time. It also limits the total data throughput you can achieve.

RP-series microcontrollers are equipped with a powerful *direct memory access* unit (DMA), which can transfer data for you in the background. Suitably programmed, the DMA can make quite long sequences of transfers without supervision.

Up to one word per system clock can be transferred to or from a PIO state machine, which is, to be quite technically precise, more bandwidth than you can shake a stick at. The bandwidth is shared across all state machines, but you can use the full amount on *one* state machine.

Let's take a look at the logic_analyser example, which uses PIO to sample some of the RP-series microcontroller's own pins, and capture a logic trace of what is going on there, at full system speed.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/logic_analyser/logic_analyser.c Lines 40 - 63

```
40 void logic_analyser_init(PIO pio, uint sm, uint pin_base, uint pin_count, float div) {
41     // Load a program to capture n pins. This is just a single `in pins, n`
42     // instruction with a wrap.
43     uint16_t capture_prog_instr = pio_encode_in(pio_pins, pin_count);
44     struct pio_program capture_prog = {
45             .instructions = &capture_prog_instr,
46             .length = 1,
47             .origin = -1
48     };
49     uint offset = pio_add_program(pio, &capture_prog);
50 
51     // Configure state machine to loop over this `in` instruction forever,
52     // with autopush enabled.
53     pio_sm_config c = pio_get_default_sm_config();
54     sm_config_set_in_pins(&c, pin_base);
55     sm_config_set_wrap(&c, offset, offset);
56     sm_config_set_clkdiv(&c, div);
57     // Note that we may push at a < 32 bit threshold if pin_count does not
58     // divide 32. We are using shift-to-right, so the sample data ends up
59     // left-justified in the FIFO in this case, with some zeroes at the LSBs.
60     sm_config_set_in_shift(&c, true, true, bits_packed_per_word(pin_count));
61     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_RX);
62     pio_sm_init(pio, sm, offset, &c);
63 }
```

Our program consists only of a single in pins, <pin_count> instruction, with program wrapping and autopull enabled. Because the amount of data to be shifted is only known at runtime, and because the program is so short, we are generating the program dynamically here (using the pio_encode_ functions) instead of pushing it through pioasm. The program is wrapped in a data structure stating how big the program is, and where it must be loaded — in this case origin = -1 meaning "don't care".

Input Shift Register

The *input shift register* (ISR) is the mirror image of the OSR. Generally data flows through a state machine in one of two directions: System → TX FIFO → OSR → Pins, or Pins → ISR → RX FIFO → System. An in instruction shifts data into the ISR.

If you don't need the ISR's shifting ability — for example, if your program is output-only — you can use the ISR as a third scratch register. It's 32 bits in size, the same as X, Y and the OSR. The full details are in the **RP2350 Datasheet**.

We load the program into the chosen PIO, and then configure the input pin mapping on the chosen state machine so that its in pins instruction will see the pins we care about. For an in instruction we only need to worry about configuring the base pin, i.e. the pin which is the least significant bit of the in instruction's sample. The number of pins to be sampled is determined by the bit count parameter of the in pins instruction — it will sample *n* pins starting at the base we specified, and shift them into the ISR.

Pin Groups (Mapping)

We mentioned earlier that there are four pin groups to configure, to connect a state machine's internal data buses to the GPIOs it manipulates. A state machine accesses all pins within a group at once, and pin groups can overlap. So far we have seen the *out*, *side-set* and *in* pin groups. The fourth is *set*.

The out group is the pins affected by shifting out data from the OSR, using `out pins` or `out pindirs`, up to 32 bits at a time. The set group is used with `set pins` and `set pindirs` instructions, up to 5 bits at a time, with data that is encoded directly in the instruction. It's useful for toggling control signals. The side-set group is similar to the set group, but runs simultaneously with another instruction. Note: `mov pin` uses the in or out group, depending on direction.

Configuring the clock divider optionally slows down the state machine's execution: a clock divisor of *n* means 1 instruction will be executed per *n* system clock cycles. The default system clock frequency for SDK is 125MHz.

`sm_config_set_in_shift` sets the shift direction to rightward, enables autopush, and sets the autopush threshold to 32. The state machine keeps an eye on the total amount of data shifted into the ISR, and on the `in` which reaches or breaches a total shift count of 32 (or whatever number you have configured), the ISR contents, along with the new data from the `in`. goes straight to the RX FIFO. The ISR is cleared to zero in the same operation.

`sm_config_set_fifo_join` is used to manipulate the FIFOs so that the DMA can get more throughput. If we want to sample every pin on every clock cycle, that's a lot of bandwidth! We've finished describing how the state machine should be configured, so we use `pio_sm_init` to load the configuration into the state machine, and get the state machine into a clean initial state.

FIFO Joining

Each state machine is equipped with a FIFO going in each direction: the TX FIFO buffers data on its way out of the system, and the RX FIFO does the same for data coming in. Each FIFO has four data slots, each holding 32 bits of data. Generally you want FIFOs to be as deep as possible, so there is more slack time between the timing-critical operation of a peripheral, and data transfers from system agents which may be quite busy or have high access latency. However this comes with significant hardware cost.

If you are only using one of the two FIFOs — TX or RX — a state machine can pool its resources to provide a single FIFO with double the depth. The **RP2350 Datasheet** goes into much more detail, including how this mechanism actually works under the hood.

Our state machine is ready to sample some pins. Let's take a look at how we hook up the DMA to our state machine, and tell the state machine to start sampling once it sees some trigger condition.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/logic_analyser/logic_analyser.c Lines 65 - 87

```
65 void logic_analyser_arm(PIO pio, uint sm, uint dma_chan, uint32_t *capture_buf, size_t capture_size_words,
66                         uint trigger_pin, bool trigger_level) {
67     pio_sm_set_enabled(pio, sm, false);
68     // Need to clear _input shift counter_, as well as FIFO, because there may be
69     // partial ISR contents left over from a previous run. sm_restart does this.
70     pio_sm_clear_fifos(pio, sm);
71     pio_sm_restart(pio, sm);
72
73     dma_channel_config c = dma_channel_get_default_config(dma_chan);
74     channel_config_set_read_increment(&c, false);
75     channel_config_set_write_increment(&c, true);
76     channel_config_set_dreq(&c, pio_get_dreq(pio, sm, false));
77
78     dma_channel_configure(dma_chan, &c,
79         capture_buf,        // Destination pointer
80         &pio->rxf[sm],      // Source pointer
81         capture_size_words, // Number of transfers
82         true                // Start immediately
83     );
84
85     pio_sm_exec(pio, sm, pio_encode_wait_gpio(trigger_level, trigger_pin));
```

```
86     pio_sm_set_enabled(pio, sm, true);
87 }
```

We want the DMA to read from the RX FIFO on our PIO state machine, so every DMA read is from the same address. The *write* address, on the other hand, should increment after every DMA transfer so that the DMA gradually fills up our capture buffer as data comes in. We need to specify a *data request* signal (DREQ) so that the DMA transfers data at the proper rate.

> **Data request signals**
>
> The DMA can transfer data incredibly fast, and almost invariably this will be much faster than your PIO program actually needs. The DMA paces itself based on a data request handshake with the state machine, so there's no worry about it overflowing or underflowing a FIFO, as long as you have selected the correct DREQ signal. The state machine coordinates with the DMA to tell it when it has room available in its TX FIFO, or data available in its RX FIFO.

We need to provide the DMA channel with an initial read address, an initial write address, and the total number of reads/writes to be performed (*not* the total number of bytes). We start the DMA channel immediately — from this point on, the DMA is poised, waiting for the state machine to produce data. As soon as data appears in the RX FIFO, the DMA will pounce and whisk the data away to our capture buffer in system memory.

As things stand right now, the state machine will immediately go into a 1-cycle loop of `in` instructions once enabled. Since the system memory available for capture is quite limited, it would be better for the state machine to wait for some trigger before it starts sampling. Specifically, we are using a `wait pin` instruction to stall the state machine until a certain pin goes high or low, and again we are using one of the `pio_encode_` functions to encode this instruction on-the-fly.

`pio_sm_exec` tells the state machine to immediately execute some instruction you give it. This instruction never gets written to the instruction memory, and if the instruction stalls (as it will in this case — a `wait` instruction's job is to stall) then the state machine will latch the instruction until it completes. With the state machine stalled on the `wait` instruction, we can enable it without being immediately flooded by data.

At this point everything is armed and waiting for the trigger signal from the chosen GPIO. This will lead to the following sequence of events:

1. The `wait` instruction will clear
2. On the very next cycle, state machine will start to execute `in` instructions from the program memory
3. As soon as data appears in the RX FIFO, the DMA will start to transfer it.
4. Once the requested amount of data has been transferred by the DMA, it'll automatically stop

> **State Machine EXEC Functionality**
>
> So far our state machines have executed instructions from the instruction memory, but there are other options. One is the `SMx_INSTR` register (used by `pio_sm_exec()`): the state machine will immediately execute whatever you write here, momentarily interrupting the current program it's running if necessary. This is useful for poking around inside the state machine from the system side, for initial setup.
>
> The other two options, which use the same underlying hardware, are `out exec` (shift out an instruction from the data being streamed through the OSR, and execute it) and `mov exec` (execute an instruction stashed in e.g. a scratch register). Besides making people's eyes bulge, these are really useful if you want the state machine to perform some data-defined operation at a certain point in an output stream.

The example code provides this cute function for displaying the captured logic trace as ASCII art in a terminal:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/logic_analyser/logic_analyser.c Lines 89 - 108

```
89 void print_capture_buf(const uint32_t *buf, uint pin_base, uint pin_count, uint32_t
   n_samples) {
```

```
    // Display the capture buffer in text form, like this:
    // 00: __--__--__--__--__--__--
    // 01: ____----____----____----
    printf("Capture:\n");
    // Each FIFO record may be only partially filled with bits, depending on
    // whether pin_count is a factor of 32.
    uint record_size_bits = bits_packed_per_word(pin_count);
    for (uint pin = 0; pin < pin_count; ++pin) {
        printf("%02d: ", pin + pin_base);
        for (uint32_t sample = 0; sample < n_samples; ++sample) {
            uint bit_index = pin + sample * pin_count;
            uint word_index = bit_index / record_size_bits;
            // Data is left-justified in each FIFO entry, hence the (32 - record_size_bits) offset
            uint word_mask = 1u << (bit_index % record_size_bits + 32 - record_size_bits);
            printf(buf[word_index] & word_mask ? "-" : "_");
        }
        printf("\n");
    }
}
```

We have everything we need now for a RP-series microcontroller to capture a logic trace of its own pins, whilst running some other program. Here we're setting up a PWM slice to output at around 15MHz on two GPIOs, and attaching our brand spanking new logic analyser to those same two GPIOs.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/logic_analyser/logic_analyser.c Lines 110 - 159

```
110 int main() {
111     stdio_init_all();
112     printf("PIO logic analyser example\n");
113 
114     // We're going to capture into a u32 buffer, for best DMA efficiency. Need
115     // to be careful of rounding in case the number of pins being sampled
116     // isn't a power of 2.
117     uint total_sample_bits = CAPTURE_N_SAMPLES * CAPTURE_PIN_COUNT;
118     total_sample_bits += bits_packed_per_word(CAPTURE_PIN_COUNT) - 1;
119     uint buf_size_words = total_sample_bits / bits_packed_per_word(CAPTURE_PIN_COUNT);
120     uint32_t *capture_buf = malloc(buf_size_words * sizeof(uint32_t));
121     hard_assert(capture_buf);
122 
123     // Grant high bus priority to the DMA, so it can shove the processors out
124     // of the way. This should only be needed if you are pushing things up to
125     // >16bits/clk here, i.e. if you need to saturate the bus completely.
126     bus_ctrl_hw->priority = BUSCTRL_BUS_PRIORITY_DMA_W_BITS | BUSCTRL_BUS_PRIORITY_DMA_R_BITS;
127 
128     PIO pio = pio0;
129     uint sm = 0;
130     uint dma_chan = 0;
131 
132     logic_analyser_init(pio, sm, CAPTURE_PIN_BASE, CAPTURE_PIN_COUNT, 1.f);
133 
134     printf("Arming trigger\n");
135     logic_analyser_arm(pio, sm, dma_chan, capture_buf, buf_size_words, CAPTURE_PIN_BASE, true);
136 
137     printf("Starting PWM example\n");
138     // PWM example: -----------------------------------------------------------
139     gpio_set_function(CAPTURE_PIN_BASE, GPIO_FUNC_PWM);
140     gpio_set_function(CAPTURE_PIN_BASE + 1, GPIO_FUNC_PWM);
141     // Topmost value of 3: count from 0 to 3 and then wrap, so period is 4 cycles
142     pwm_hw->slice[0].top = 3;
```

```
    // Divide frequency by two to slow things down a little
    pwm_hw->slice[0].div = 4 << PWM_CH0_DIV_INT_LSB;
    // Set channel A to be high for 1 cycle each period (duty cycle 1/4) and
    // channel B for 3 cycles (duty cycle 3/4)
    pwm_hw->slice[0].cc =
            (1 << PWM_CH0_CC_A_LSB) |
            (3 << PWM_CH0_CC_B_LSB);
    // Enable this PWM slice
    pwm_hw->slice[0].csr = PWM_CH0_CSR_EN_BITS;
    // ----------------------------------------------------------------------

    // The logic analyser should have started capturing as soon as it saw the
    // first transition. Wait until the last sample comes in from the DMA.
    dma_channel_wait_for_finish_blocking(dma_chan);

    print_capture_buf(capture_buf, CAPTURE_PIN_BASE, CAPTURE_PIN_COUNT, CAPTURE_N_SAMPLES);
}
```

The output of the program looks like this:

```
Starting PWM example
Capture:
16: ----____________----____________----____________----____________----_______
17: ------------____------------____------------____------------____-----------
```

3.2.4. Further examples

Hopefully what you have seen so far has given some idea of how PIO applications can be built with the SDK. The **RP2350 Datasheet** contains *many* more documented examples, which highlight particular hardware features of PIO, or show how particular hardware interfaces can be implemented.

You can also browse the `pio/` directory in the Pico Examples repository.

3.3. Using PIOASM, the PIO Assembler

Up until now, we have glossed over the details of how the assembly program in our `.pio` file is translated into a binary program, ready to be loaded into our PIO state machine. Programs that handle this task — translating assembly code into binary — are generally referred to as *assemblers*, and PIO is no exception in this regard. The SDK includes an assembler for PIO, called `pioasm`. The SDK handles the details of building this tool for you behind the scenes, and then using it to build your PIO programs, for you to `#include` from your C or C++ program. `pioasm` can also be used directly, and has a few features not used by the C++ SDK, such as generating programs suitable for use with the MicroPython PIO library.

If you have built the `pico-examples` repository at any point, you will likely already have a `pioasm` binary in your build directory, located under `build/tools/pioasm/pioasm`, which was bootstrapped for you before building any applications that depend on it. If we want a standalone copy of `pioasm`, perhaps just to explore the available command-line options, we can obtain it as follows (assuming the SDK is extracted at `$PICO_SDK_PATH`):

```
$ mkdir pioasm_build
$ cd pioasm_build
$ cmake $PICO_SDK_PATH/tools/pioasm
$ make
```

And then invoke as:

```
$ ./pioasm
```

3.3.1. Usage

A description of the command line arguments can be obtained by running:

```
$ pioasm -?
```

giving:

```
usage: pioasm <options> <input> (<output>)

Assemble file of PIO program(s) for use in applications.
<input>             the input filename
<output>            the output filename (or filename prefix if the output
                        format produces multiple outputs).
                    if not specified, a single output will be written to stdout

options:
-o <output_format>   select output_format (default 'c-sdk'); available options are:
                        c-sdk
                            C header suitable for use with the Raspberry Pi Pico SDK
                        python
                            Python file suitable for use with MicroPython
                        hex
                            Raw hex output (only valid for single program inputs)
-v <version>         specify the default PIO version (0 or 1)
-p <output_param>    add a parameter to be passed to the outputter
-?, --help           print this help and exit
```

NOTE

Within the SDK you do not need to invoke pioasm directly, as the `CMake` function `pico_generate_pio_header(TARGET PIO_FILE)` takes care of invoking pioasm and adding the generated header to the include path of the target TARGET for you.

3.3.2. Directives

The following directives control the assembly of PIO programs:

Table 4. alphabetical list of pioasm directives

.define (*PUBLIC*) *<symbol>* *<value>*	Define an integer symbol named *<symbol>* with the value *<value>* (see Section 3.3.3). If this *.define* appears before the first program in the input file, then this define is global to all programs, otherwise it is local to the program in which it occurs. If *PUBLIC* is specified the symbol will be emitted into the assembled output for use by user code. For the SDK this takes the form of: `#define <program_name>_<symbol> value` for program symbols or `#define <symbol> value` for global symbols

.clock_div <divider>

If this directive is present, *<divider>* is the state machine clock divider for the program. Note, that divider is a floating point value, but *may not* currently use arithmetic expressions or defined values. This directive affects the default state machine configuration for a program. This directive is only valid within a program before the first instruction

.fifo <fifo_config>

If this directive is present, it is used to specify the FIFO configuration for the program. It affects the default state machine configuration for a program, but also restricts what instructions may be used (for example PUSH makes no sense if there is no IN FIFO configrued).

The following values are supported:

`txrx`: 4 FIFO entries for each of TX and RX; this is the default. `tx` - All 8 FIFO entries for TX.
`rx` - All 8 FIFO entries for RX.
`txput` - 4 FIFO entries for TX, and 4 FIFO entries for `mov rxfifo[index], isr` aka *put*. This value is not supported on PIO version 0.
`txget` - 4 FIFO entries for TX, and 4 FIFO entries for `mov osr, rxfifo[index]` aka *get*. This value is not supported on PIO version 0.
`putget` - 4 FIFO entries for `mov rxfifo[index], isr` aka `put`, and 4 FIFO entries for `mov osr, rxfifo[index]` aka `get`. This value is not supported on PIO version 0.

This directive is only valid within a program before the first instruction

.mov_status rxfifo < <n>
.mov_status txfifo < <n>
.mov_status irq <(next|prev)> set <n>

This directive configures the source for the `mov , STATUS`. One of the three syntaxes can be used to set the status based on the RXFIFO level being below a value N, the TXFIFO level being below a value N, or an IRQ flag N being set on this PIO instance (or the next higer numbered, or lowered numbered PIO instance if `next` or `prev` or specified). Note, that the IRQ option requires PIO version 1.

This directive affects the default state machine configuration for a program. This directive is only valid within a program before the first instruction

.in <count> (left|right) (auto) (<threshold>)

If this directive is present, *<count>* indicates the number of IN bits to be used. 'left' or 'right' if specified, control the ISR shift direction; 'auto', if present, enables "auto-push"; *<threshold>*, if present, specifies the "auto-push" threshold. This directive affects the default state machine configuration for a program. This directive is only valid within a program before the first instruction

When assembling for PIO version 0, *count* must be 32.

.program <name>

Start a new program with the name *<name>*. Note that that name is used in code so should be alphanumeric/underscore not starting with a digit. The program lasts until another *.program* directive or the end of the source file. PIO instructions are only allowed within a program

.origin <offset>

Optional directive to specify the PIO instruction memory offset at which the program *must* load. Most commonly this is used for programs that must load at offset 0, because they use data based JMPs with the (absolute) jmp target being stored in only a few bits. This directive is invalid outside a program

.out <count> (left|right) (auto) (<threshold>)	If this directive is present, *<count>* indicates the number of OUT bits to be used. 'left' or 'right' if specified control the OSR shift direction; 'auto', if present, enables "auto-pull"; *<threshold>*, if present, specifies the "auto-pull" threshold. This directive affects the default state machine configuration for a program. This directive is only valid within a program before the first instruction
.pio_version <version>	This directive sets the target PIO hardware version. The version for RP2350 is `1` or `RP2350`, and is also the default version number. For backwards compatibility with RP2040, `0` or `RP2040` may be used. If this directive appears before the first program in the input file, then this define is the default for all programs, otherwise it specifies the version for the program in which it occurs. If specified for a program, it must occur before the first instruction.
.set <count>	If this directive is present, *<count>* indicates the number of SET bits to be used. This directive affects the default state machine configuration for a program. This directive is only valid within a program before the first instruction
.side_set <count> (opt) (pindirs)	If this directive is present, *<count>* indicates the number of side-set bits to be used. Additionally *opt* may be specified to indicate that a `side <value>` is optional for instructions (note this requires stealing an extra bit — in addition to the *<count>* bits — from those available for the instruction delay). Finally, *pindirs* may be specified to indicate that the side set values should be applied to the PINDIRs and not the PINs. This directive is only valid within a program before the first instruction
.wrap_target	Place prior to an instruction, this directive specifies the instruction where execution continues due to program wrapping. This directive is invalid outside of a program, may only be used once within a program, and if not specified defaults to the start of the program
.wrap	Placed after an instruction, this directive specifies the instruction after which, in normal control flow (i.e. `jmp` with false condition, or no `jmp`), the program wraps (to *.wrap_target* instruction). This directive is invalid outside of a program, may only be used once within a program, and if not specified defaults to after the last program instruction.
.lang_opt <lang> <name> <option>	Specifies an option for the program related to a particular language generator. (See Section 3.3.10). This directive is invalid outside of a program
.word <value>	Stores a raw 16-bit value as an instruction in the program. This directive is invalid outside of a program.

3.3.3. Values

The following types of values can be used to define integer numbers or branch targets

Table 5. Values in pioasm, i.e. <value>

integer	An integer value e.g. 3 or -7
hex	A hexadecimal value e.g. `0xf`
binary	A binary value e.g. `0b1001`
symbol	A value defined by a `.define` (see `pioasm_define`)
<label>	The instruction offset of the label within the program. This makes most sense when used with a JMP instruction (see Section 3.4.4)

(<expression>)	An expression to be evaluated; see expressions. Note that the parentheses are necessary.

3.3.4. Expressions

Expressions may be freely used within pioasm values.

Table 6. Expressions in pioasm i.e. <expression>

<expression> + <expression>	The sum of two expressions
<expression> - <expression>	The difference of two expressions
*<expression> * <expression>*	The multiplication of two expressions
<expression> / <expression>	The integer division of two expressions
- <expression>	The negation of another expression
<expression> << <expression>	One expression shifted left by another expression
<expression> >> <expression>	One expression shifted right by another expression
:: <expression>	The bit reverse of another expression
<value>	Any value (see Section 3.3.3)

3.3.5. Comments

Line comments are supported with `//` or `;`

C-style block comments are supported via `/*` and `*/`

3.3.6. Labels

Labels are of the form:

```
<symbol>:
```

or

```
PUBLIC <symbol>:
```

at the start of a line.

TIP

> A label is really just an automatic `.define` with a value set to the current program instruction offset. A *PUBLIC* label is exposed to the user code in the same way as a *PUBLIC* `.define`.

3.3.7. Instructions

All pioasm instructions follow a common pattern:

<instruction> (*side <side_set_value>*) (*[<delay_value>]*)

where:

<instruction>	Is an assembly instruction detailed in the following sections. (See Section 3.4)

<side_set_value> Is a value (see Section 3.3.3) to apply to the side_set pins at the start of the instruction. Note that the rules for a side-set value via `side <side_set_value>` are dependent on the `.side_set` (see `pioasm_side_set`) directive for the program. If no `.side_set` is specified then the `side <side_set_value>` is invalid, if an optional number of sideset pins is specified then `side <side_set_value>` may be present, and if a non-optional number of sideset pins is specified, then `side <side_set_value>` is required. The *<side_set_value>* must fit within the number of side-set bits specified in the `.side_set` directive.

<delay_value> Specifies the number of cycles to delay after the instruction completes. The delay_value is specified as a value (see Section 3.3.3), and in general is between 0 and 31 inclusive (a 5-bit value), however the number of bits is reduced when sideset is enabled via the `.side_set` (see `pioasm_side_set`) directive. If the *<delay_value>* is not present, then the instruction has no delay

NOTE

pioasm instruction names, keywords and directives are case insensitive; lower case is used in the *Assembly Syntax* sections below as this is the style used in the SDK.

NOTE

Commas appear in some *Assembly Syntax* sections below, but are entirely optional, e.g. `out pins, 3` may be written `out pins 3`, and `jmp x-- label` may be written as `jmp x--, label`. The *Assembly Syntax* sections below uses the first style in each case as this is the style used in the SDK.

3.3.8. Pseudoinstructions

Currently pioasm provides one pseudoinstruction, as a convenience:

`nop` Assembles to `mov y, y`. "No operation", has no particular side effect, but a useful vehicle for a side-set operation or an extra delay.

3.3.9. Output pass through

Text in the PIO file may be passed, unmodified, to the output based on the language generator being used.

For example the following (comment and function) would be included in the generated header when the default `c-sdk` language generator is used.

```
% c-sdk {

// an inline function (since this is going in a header file)
static inline int some_c_code() {
    return 0;
}
%}
```

The general format is

```
% target {
pass through contents
%}
```

with `targets` being recognized by a particular language generator (see Section 3.3.10; note that `target` is usually the language generator name e.g. `c-sdk`, but could potentially be `some_language.some_group` if the language generator supports different classes of pass through with different output locations.

This facility allows you to encapsulate both the PIO program and the associated setup required in the same source file. See Section 3.3.10 for a more complete example.

3.3.10. Language generators

The following example shows a multi program source file (with multiple programs) which we will use to highlight c-sdk and python output features

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/ws2812.pio

```
;
; Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
;
; SPDX-License-Identifier: BSD-3-Clause
;
.pio_version 0 // only requires PIO version 0

.program ws2812
.side_set 1

; The following constants are selected for broad compatibility with WS2812,
; WS2812B, and SK6812 LEDs. Other constants may support higher bandwidths for
; specific LEDs, such as (7,10,8) for WS2812B LEDs.

.define public T1 3
.define public T2 3
.define public T3 4

.lang_opt python sideset_init = pico.PIO.OUT_HIGH
.lang_opt python out_init     = pico.PIO.OUT_HIGH
.lang_opt python out_shiftdir = 1

.wrap_target
bitloop:
    out x, 1       side 0 [T3 - 1] ; Side-set still takes place when instruction stalls
    jmp !x do_zero side 1 [T1 - 1] ; Branch on the bit we shifted out. Positive pulse
do_one:
    jmp  bitloop   side 1 [T2 - 1] ; Continue driving high, for a long pulse
do_zero:
    nop            side 0 [T2 - 1] ; 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_init(pio, pin);
    pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);

    pio_sm_config c = ws2812_program_get_default_config(offset);
    sm_config_set_sideset_pins(&c, pin);
    sm_config_set_out_shift(&c, false, true, rgbw ? 32 : 24);
    sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);

    int cycles_per_bit = ws2812_T1 + ws2812_T2 + ws2812_T3;
```

```
    float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit);
    sm_config_set_clkdiv(&c, div);

    pio_sm_init(pio, sm, offset, &c);
    pio_sm_set_enabled(pio, sm, true);
}
%}

.program ws2812_parallel

.define public T1 3
.define public T2 3
.define public T3 4

.wrap_target
    out x, 32
    mov pins, !null [T1-1]
    mov pins, x     [T2-1]
    mov pins, null  [T3-2]
.wrap

% c-sdk {
#include "hardware/clocks.h"

static inline void ws2812_parallel_program_init(PIO pio, uint sm, uint offset, uint pin_base,
uint pin_count, float freq) {
    for(uint i=pin_base; i<pin_base+pin_count; i++) {
        pio_gpio_init(pio, i);
    }
    pio_sm_set_consecutive_pindirs(pio, sm, pin_base, pin_count, true);

    pio_sm_config c = ws2812_parallel_program_get_default_config(offset);
    sm_config_set_out_shift(&c, true, true, 32);
    sm_config_set_out_pins(&c, pin_base, pin_count);
    sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);

    int cycles_per_bit = ws2812_parallel_T1 + ws2812_parallel_T2 + ws2812_parallel_T3;
    float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit);
    sm_config_set_clkdiv(&c, div);

    pio_sm_init(pio, sm, offset, &c);
    pio_sm_set_enabled(pio, sm, true);
}
%}
```

3.3.10.1. c-sdk

The c-sdk language generator produces a single header file with all the programs in the PIO source file:

The pass through sections (`% c-sdk {`) are embedded in the output, and the `PUBLIC` defines are available via `#define`

> **TIP**
>
> `pioasm` creates a function for each program (e.g. `ws2812_program_get_default_config()`) returning a `pio_sm_config` based on the `.side_set`, `.wrap` and `.wrap_target` settings of the program, which you can then use as a basis for configuration the PIO state machine.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/generated/ws2812.pio.h

```
// -------------------------------------------------- //
// This file is autogenerated by pioasm; do not edit! //
// -------------------------------------------------- //

#pragma once

#if !PICO_NO_HARDWARE
#include "hardware/pio.h"
#endif

// ------ //
// ws2812 //
// ------ //

#define ws2812_wrap_target 0
#define ws2812_wrap 3
#define ws2812_pio_version 0

#define ws2812_T1 3
#define ws2812_T2 3
#define ws2812_T3 4

static const uint16_t ws2812_program_instructions[] = {
            //     .wrap_target
    0x6321, //  0: out    x, 1            side 0 [3]
    0x1223, //  1: jmp    !x, 3           side 1 [2]
    0x1200, //  2: jmp    0               side 1 [2]
    0xa242, //  3: nop                    side 0 [2]
            //     .wrap
};

#if !PICO_NO_HARDWARE
static const struct pio_program ws2812_program = {
    .instructions = ws2812_program_instructions,
    .length = 4,
    .origin = -1,
    .pio_version = ws2812_pio_version,
#if PICO_PIO_VERSION > 0
    .used_gpio_ranges = 0x0
#endif
};

static inline pio_sm_config ws2812_program_get_default_config(uint offset) {
    pio_sm_config c = pio_get_default_sm_config();
    sm_config_set_wrap(&c, offset + ws2812_wrap_target, offset + ws2812_wrap);
    sm_config_set_sideset(&c, 1, false, false);
    return c;
}

#include "hardware/clocks.h"
static inline void ws2812_program_init(PIO pio, uint sm, uint offset, uint pin, float freq,
bool rgbw) {
    pio_gpio_init(pio, pin);
    pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
```

```
    pio_sm_config c = ws2812_program_get_default_config(offset);
    sm_config_set_sideset_pins(&c, pin);
    sm_config_set_out_shift(&c, false, true, rgbw ? 32 : 24);
    sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
    int cycles_per_bit = ws2812_T1 + ws2812_T2 + ws2812_T3;
    float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit);
    sm_config_set_clkdiv(&c, div);
    pio_sm_init(pio, sm, offset, &c);
    pio_sm_set_enabled(pio, sm, true);
}

#endif

// --------------- //
// ws2812_parallel //
// --------------- //

#define ws2812_parallel_wrap_target 0
#define ws2812_parallel_wrap 3
#define ws2812_parallel_pio_version 0

#define ws2812_parallel_T1 3
#define ws2812_parallel_T2 3
#define ws2812_parallel_T3 4

static const uint16_t ws2812_parallel_program_instructions[] = {
            //     .wrap_target
    0x6020, //  0: out    x, 32
    0xa20b, //  1: mov    pins, ~null            [2]
    0xa201, //  2: mov    pins, x                [2]
    0xa203, //  3: mov    pins, null             [2]
            //     .wrap
};

#if !PICO_NO_HARDWARE
static const struct pio_program ws2812_parallel_program = {
    .instructions = ws2812_parallel_program_instructions,
    .length = 4,
    .origin = -1,
    .pio_version = ws2812_parallel_pio_version,
#if PICO_PIO_VERSION > 0
    .used_gpio_ranges = 0x0
#endif
};

static inline pio_sm_config ws2812_parallel_program_get_default_config(uint offset) {
    pio_sm_config c = pio_get_default_sm_config();
    sm_config_set_wrap(&c, offset + ws2812_parallel_wrap_target, offset +
ws2812_parallel_wrap);
    return c;
}

#include "hardware/clocks.h"
static inline void ws2812_parallel_program_init(PIO pio, uint sm, uint offset, uint
pin_base, uint pin_count, float freq) {
    for(uint i=pin_base; i<pin_base+pin_count; i++) {
        pio_gpio_init(pio, i);
    }
    pio_sm_set_consecutive_pindirs(pio, sm, pin_base, pin_count, true);
    pio_sm_config c = ws2812_parallel_program_get_default_config(offset);
    sm_config_set_out_shift(&c, true, true, 32);
    sm_config_set_out_pins(&c, pin_base, pin_count);
    sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
    int cycles_per_bit = ws2812_parallel_T1 + ws2812_parallel_T2 + ws2812_parallel_T3;
```

```
    float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit);
    sm_config_set_clkdiv(&c, div);
    pio_sm_init(pio, sm, offset, &c);
    pio_sm_set_enabled(pio, sm, true);
}

#endif
```

3.3.10.2. python

The python language generator produces a single python file with all the programs in the PIO source file:

The pass through sections (`% python {`) would be embedded in the output, and the `PUBLIC` defines are available as python variables.

Also note the use of `.lang_opt python` to pass initializers for the `@pico.asm_pio` decorator

TIP

The python language output is provided as a utility. MicroPython supports programming with the PIO natively, so you may only want to use pioasm when sharing PIO code between the SDK and MicroPython. No effort is currently made to preserve label names, symbols or comments, as it is assumed you are either using the PIO file as a source or python; not both. The python language output can of course be used to bootstrap your MicroPython PIO development based on an existing PIO file.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/generated/ws2812.py

```python
# -------------------------------------------------- #
# This file is autogenerated by pioasm; do not edit! #
# -------------------------------------------------- #

import rp2
from machine import Pin
# ------ #
# ws2812 #
# ------ #

ws2812_T1 = 3
ws2812_T2 = 3
ws2812_T3 = 4

@rp2.asm_pio(sideset_init=pico.PIO.OUT_HIGH, out_init=pico.PIO.OUT_HIGH, out_shiftdir=1)
def ws2812():
    wrap_target()
    label("0")
    out(x, 1)               .side(0) [3]  # 0
    jmp(not_x, "3")         .side(1) [2]  # 1
    jmp("0")                .side(1) [2]  # 2
    label("3")
    nop()                   .side(0) [2]  # 3
    wrap()



# -------------- #
# ws2812_parallel #
# -------------- #

ws2812_parallel_T1 = 3
ws2812_parallel_T2 = 3
```

```
ws2812_parallel_T3 = 4

@rp2.asm_pio()
def ws2812_parallel():
    wrap_target()
    out(x, 32)                        # 0
    mov(pins, invert(null))       [2] # 1
    mov(pins, x)                  [2] # 2
    mov(pins, null)               [2] # 3
    wrap()
```

3.3.10.3. hex

The hex generator only supports a single input program, as it just dumps the raw instructions (one per line) as a 4-character hexadecimal number.

Given:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.pio

```
;
; Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
;
; SPDX-License-Identifier: BSD-3-Clause
;
.pio_version 0 // only requires PIO version 0

.program squarewave
    set pindirs, 1   ; Set pin to output
again:
    set pins, 1 [1]  ; Drive pin high and then delay for one cycle
    set pins, 0      ; Drive pin low
    jmp again        ; Set PC to label `again`
```

The *hex* output produces:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/generated/squarewave.hex

```
e081
e101
e000
0001
```

3.4. PIO Instruction Set Reference

NOTE

This section refers in places to concepts and pieces of hardware discussed in the **RP2350 Datasheet**. You are encouraged to read the PIO chapter of the datasheet to get the full context for what these instructions do.

The following sections document instruction behaviour on both PIO version 0 (RP2040) and PIO version 1 (RP2350). When no version restrictions are mentioned, this means the behaviour applies to both versions. PIO version 1 is strictly additive over version 0, so some features may be indicated as version-1-only, but none are version-0-only.

For documentation specific to a particular PIO version, see the device datasheet for a device equipped with that version.

3.4.1. Encoding (version 0, RP2040)

PIO instructions are 16 bits long, and have the following encoding:

Table 7. PIO instruction encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	0	0	0	Delay/side-set					Condition			Address				
WAIT	0	0	1	Delay/side-set					Pol	Source		Index				
IN	0	1	0	Delay/side-set					Source			Bit count				
OUT	0	1	1	Delay/side-set					Destination			Bit count				
PUSH	1	0	0	Delay/side-set					0	IfF	Blk	0	0	0	0	0
PULL	1	0	0	Delay/side-set					1	IfE	Blk	0	0	0	0	0
MOV	1	0	1	Delay/side-set					Destination			Op		Source		
IRQ	1	1	0	Delay/side-set					0	Clr	Wait	Index				
SET	1	1	1	Delay/side-set					Destination			Data				

3.4.2. Encoding (version 1, RP2350)

PIO instructions are 16 bits long, and have the following encoding:

Table 8. PIO instruction encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	0	0	0	Delay/side-set					Condition			Address				
WAIT	0	0	1	Delay/side-set					Pol	Source		Index				
IN	0	1	0	Delay/side-set					Source			Bit count				
OUT	0	1	1	Delay/side-set					Destination			Bit count				
PUSH	1	0	0	Delay/side-set					0	IfF	Blk	0	0	0	0	0
MOV	1	0	0	Delay/side-set					0	0	0	1	IdxI	0	Index	
PULL	1	0	0	Delay/side-set					1	IfE	Blk	0	0	0	0	0
MOV	1	0	0	Delay/side-set					1	0	0	1	IdxI	0	Index	
MOV	1	0	1	Delay/side-set					Destination			Op		Source		
IRQ	1	1	0	Delay/side-set					0	Clr	Wait	IdxMode		Index		
SET	1	1	1	Delay/side-set					Destination			Data				

3.4.3. Summary

All PIO instructions execute in one clock cycle.

The `Delay/side-set` field is present in all instructions. Its exact use is configured for each state machine by `PINCTRL_SIDESET_COUNT`:

- Up to 5 MSBs encode a side-set operation, which optionally asserts a constant value onto some GPIOs, concurrently with main instruction execution logic

- Remaining LSBs (up to 5) encode the number of idle cycles inserted between this instruction and the next

3.4.4. JMP

3.4.4.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`JMP`	0	0	0	Delay/side-set					Condition			Address				

3.4.4.2. Operation

Set program counter to `Address` if `Condition` is true, otherwise no operation.

Delay cycles on a `JMP` always take effect, whether `Condition` is true or false, and they take place *after* `Condition` is evaluated and the program counter is updated.

- Condition:
  - 000: *(no condition)*: Always
  - 001: `!X`: scratch X zero
  - 010: `X--`: scratch X non-zero, prior to decrement
  - 011: `!Y`: scratch Y zero
  - 100: `Y--`: scratch Y non-zero, prior to decrement
  - 101: `X!=Y`: scratch X not equal scratch Y
  - 110: `PIN`: branch on input pin
  - 111: `!OSRE`: output shift register not empty
- Address: Instruction address to jump to. In the instruction encoding this is an absolute address within the PIO instruction memory.

`JMP PIN` branches on the GPIO selected by `EXECCTRL_JMP_PIN`, a configuration field which selects one out of the maximum of 32 GPIO inputs visible to a state machine, independently of the state machine's other input mapping. The branch is taken if the GPIO is high.

`!OSRE` compares the bits shifted out since the last `PULL` with the shift count threshold configured by `SHIFTCTRL_PULL_THRESH`. This is the same threshold used by autopull.

`JMP X--` and `JMP Y--` always decrement scratch register X or Y, respectively. The decrement is not conditional on the current value of the scratch register. The branch is conditioned on the *initial* value of the register, i.e. before the decrement took place: if the register is initially nonzero, the branch is taken.

3.4.4.3. Assembler Syntax

jmp (<cond>) <target>

where:

<cond>	Is an optional condition listed above (e.g. `!x` for scratch X zero). If a condition code is not specified, the branch is always taken

<target>	Is a program label or value (see Section 3.3.3) representing instruction offset within the program (the first instruction being offset 0). Note that because the PIO JMP instruction uses absolute addresses in the PIO instruction memory, JMPs need to be adjusted based on the program load offset at runtime. This is handled for you when loading a program with the SDK, but care should be taken when encoding JMP instructions for use by `OUT EXEC`

3.4.5. WAIT

3.4.5.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`WAIT`	0	0	1	Delay/side-set					Pol	Source		Index				

3.4.5.2. Operation

Stall until some condition is met.

Like all stalling instructions, delay cycles begin after the instruction *completes*. That is, if any delay cycles are present, they do not begin counting until *after* the wait condition is met.

- Polarity:
  - 1: wait for a 1.
  - 0: wait for a 0.
- Source: what to wait on. Values are:
  - 00: `GPIO`: System GPIO input selected by `Index`. This is an absolute GPIO index, and is not affected by the state machine's input IO mapping.
  - 01: `PIN`: Input pin selected by `Index`. This state machine's input IO mapping is applied first, and then `Index` selects which of the mapped bits to wait on. In other words, the pin is selected by adding `Index` to the `PINCTRL_IN_BASE` configuration, modulo 32.
  - 10: `IRQ`: PIO IRQ flag selected by `Index`
  - 11: (*version 1 and above*) `JMPPIN`: wait on the pin indexed by the `PINCTRL_JMP_PIN` configuration, plus an Index in the range 0-3, all modulo 32. Other values of Index are reserved.
- Index: which pin or bit to check.

`WAIT x IRQ` behaves slightly differently from other `WAIT` sources:

- If `Polarity` is 1, the selected IRQ flag is cleared by the state machine upon the wait condition being met.
- The flag index is decoded in the same way as the `IRQ` index field, decoding down from the two MSBs (aligning with the `IRQ` instruction IdxMode field):
  - `00`: the three LSBs are used directly to index the IRQ flags in this PIO block.
  - `01` (*version 1 and above*) (`PREV`), the instruction references an IRQ from the next-lower-numbered PIO in the system, wrapping to the highest-numbered PIO if this is PIO0.
  - `10` (`REL`), the state machine ID (0…3) is added to the IRQ index, by way of modulo-4 addition on the two LSBs. For example, state machine 2 with a flag value of '0x11' will wait on flag 3, and a flag value of '0x13' will wait on flag 1. This allows multiple state machines running the same program to synchronise with each other.
  - `11` (*version 1 and above*) (`NEXT`), the instruction references an IRQ from the next-higher-numbered PIO in the system, wrapping to PIO0 if this is the highest-numbered PIO.

> **⚠ CAUTION**
>
> `WAIT 1 IRQ x` should not be used with IRQ flags presented to the interrupt controller, to avoid a race condition with a system interrupt handler

3.4.5.3. Assembler Syntax

wait <polarity> gpio <gpio_num>

wait <polarity> pin <pin_num>

wait <polarity> irq <irq_num> (*rel*)

where:

<polarity>	Is a value (see Section 3.3.3) specifying the polarity (either 0 or 1)
<pin_num>	Is a value (see Section 3.3.3) specifying the input pin number (as mapped by the SM input pin mapping)
<gpio_num>	Is a value (see Section 3.3.3) specifying the actual GPIO pin number
<irq_num> (*rel*)	Is a value (see Section 3.3.3) specifying The irq number to wait on (0-7). If *rel* is present, then the actual irq number used is calculating by replacing the low two bits of the irq number (irq_num_{10}) with the low two bits of the sum (irq_num_{10} + sm_num_{10}) where sm_num_{10} is the state machine number

3.4.6. IN

3.4.6.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`IN`	0	1	0	Delay/side-set					Source			Bit count				

3.4.6.2. Operation

Shift `Bit count` bits from `Source` into the Input Shift Register (ISR). Shift direction is configured for each state machine by `SHIFTCTRL_IN_SHIFTDIR`. Additionally, increase the input shift count by `Bit count`, saturating at 32.

- Source:
  - 000: `PINS`
  - 001: `X` (scratch register X)
  - 010: `Y` (scratch register Y)
  - 011: `NULL` (all zeroes)
  - 100: Reserved
  - 101: Reserved
  - 110: `ISR`
  - 111: `OSR`
- Bit count: How many bits to shift into the ISR. 1…32 bits, 32 is encoded as `00000`.

If automatic push is enabled, `IN` will also push the ISR contents to the RX FIFO if the push threshold is reached (`SHIFTCTRL_PUSH_THRESH`). `IN` still executes in one cycle, whether an automatic push takes place or not. The state machine will stall if the RX FIFO is full when an automatic push occurs. An automatic push clears the ISR contents to all-zeroes, and clears the input shift count.

`IN` always uses the least significant `Bit count` bits of the source data. For example, if `PINCTRL_IN_BASE` is set to 5, the instruction `IN PINS, 3` will take the values of pins 5, 6 and 7, and shift these into the ISR. First the ISR is shifted to the left or right to make room for the new input data, then the input data is copied into the gap this leaves. The bit order of the input data is not dependent on the shift direction.

`NULL` can be used for shifting the ISR's contents. For example, UARTs receive the LSB first, so must shift to the right. After 8 `IN PINS, 1` instructions, the input serial data will occupy bits 31…24 of the ISR. An `IN NULL, 24` instruction will shift in 24 zero bits, aligning the input data at ISR bits 7…0. Alternatively, the processor or DMA could perform a byte read from FIFO address + 3, which would take bits 31…24 of the FIFO contents.

3.4.6.3. Assembler Syntax

in <source>, <bit_count>

where:

<source>	Is one of the sources specified above.
<bit_count>	Is a value (see Section 3.3.3) specifying the number of bits to shift (valid range 1-32)

3.4.7. OUT

3.4.7.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`OUT`	0	1	1	Delay/side-set					Destination			Bit count				

3.4.7.2. Operation

Shift `Bit count` bits out of the Output Shift Register (OSR), and write those bits to `Destination`. Additionally, increase the output shift count by `Bit count`, saturating at 32.

- Destination:
  - 000: `PINS`
  - 001: `X` (scratch register X)
  - 010: `Y` (scratch register Y)
  - 011: `NULL` (discard data)
  - 100: `PINDIRS`
  - 101: `PC`
  - 110: `ISR` (also sets ISR shift counter to `Bit count`)
  - 111: `EXEC` (Execute OSR shift data as instruction)
- Bit count: how many bits to shift out of the OSR. 1…32 bits, 32 is encoded as `00000`.

A 32-bit value is written to `Destination`: the lower `Bit count` bits come from the OSR, and the remainder are zeroes. This value is the least significant `Bit count` bits of the OSR if `SHIFTCTRL_OUT_SHIFTDIR` is to the right, otherwise it is the most significant bits.

`PINS` and `PINDIRS` use the `OUT` pin mapping.

If automatic pull is enabled, the OSR is automatically refilled from the TX FIFO if the pull threshold, `SHIFTCTRL_PULL_THRESH`, is reached. The output shift count is simultaneously cleared to 0. In this case, the `OUT` will stall if the TX FIFO is empty, but otherwise still executes in one cycle.

`OUT EXEC` allows instructions to be included inline in the FIFO datastream. The `OUT` itself executes on one cycle, and the instruction from the OSR is executed on the next cycle. There are no restrictions on the types of instructions which can be executed by this mechanism. Delay cycles on the initial `OUT` are ignored, but the executee may insert delay cycles as normal.

`OUT PC` behaves as an unconditional jump to an address shifted out from the OSR.

3.4.7.3. Assembler Syntax

out <destination>, <bit_count>

where:

<destination>	Is one of the destinations specified above.
<bit_count>	Is a value (see Section 3.3.3) specifying the number of bits to shift (valid range 1-32)

3.4.8. PUSH

3.4.8.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`PUSH`	1	0	0	Delay/side-set					0	IfF	Blk	0	0	0	0	0

3.4.8.2. Operation

Push the contents of the ISR into the RX FIFO, as a single 32-bit word. Clear ISR to all-zeroes.

- `IfFull`: If 1, do nothing unless the total input shift count has reached its threshold, `SHIFTCTRL_PUSH_THRESH` (the same as for autopush).
- `Block`: If 1, stall execution if RX FIFO is full.

`PUSH IFFULL` helps to make programs more compact, like autopush. It is useful in cases where the `IN` would stall at an inappropriate time if autopush were enabled, e.g. if the state machine is asserting some external control signal at this point.

The PIO assembler sets the `Block` bit by default. If the `Block` bit is not set, the `PUSH` does not stall on a full RX FIFO, instead continuing immediately to the next instruction. The FIFO state and contents are unchanged when this happens. The ISR is still cleared to all-zeroes, and the `FDEBUG_RXSTALL` flag is set (the same as a blocking `PUSH` or autopush to a full RX FIFO) to indicate data was lost.

3.4.8.3. Assembler Syntax

push (iffull)

push (*iffull*) block

push (*iffull*) noblock

where:

iffull	Is equivalent to `IfFull == 1` above. i.e. the default if this is not specified is `IfFull == 0`
block	Is equivalent to `Block == 1` above. This is the default if neither *block* nor *noblock* are specified
noblock	Is equivalent to `Block == 0` above.

3.4.9. PULL

3.4.9.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`PULL`	1	0	0	Delay/side-set					1	IfE	Blk	0	0	0	0	0

3.4.9.2. Operation

Load a 32-bit word from the TX FIFO into the OSR.

- `IfEmpty`: If 1, do nothing unless the total output shift count has reached its threshold, `SHIFTCTRL_PULL_THRESH` (the same as for autopull).
- `Block`: If 1, stall if TX FIFO is empty. If 0, pulling from an empty FIFO copies scratch X to OSR.

Some peripherals (UART, SPI…) should halt when no data is available, and pick it up as it comes in; others (I2S) should clock continuously, and it is better to output placeholder or repeated data than to stop clocking. This can be achieved with the `Block` parameter.

A nonblocking `PULL` on an empty FIFO has the same effect as `MOV OSR, X`. The program can either preload scratch register X with a suitable default, or execute a `MOV X, OSR` after each `PULL NOBLOCK`, so that the last valid FIFO word will be recycled until new data is available.

`PULL IFEMPTY` is useful if an `OUT` with autopull would stall in an inappropriate location when the TX FIFO is empty. For example, a UART transmitter should not stall immediately after asserting the start bit. `IfEmpty` permits some of the same program simplifications as autopull, but the stall occurs at a controlled point in the program.

NOTE

> When autopull is enabled, any `PULL` instruction is a no-op when the OSR is full, so that the `PULL` instruction behaves as a barrier. `OUT NULL, 32` can be used to explicitly discard the OSR contents. See the **RP2350 Datasheet** for more detail on autopull.

3.4.9.3. Assembler Syntax

pull (*ifempty*)

pull (*ifempty*) block

pull (*ifempty*) noblock

where:

ifempty	Is equivalent to `IfEmpty == 1` above. i.e. the default if this is not specified is `IfEmpty == 0`

block	Is equivalent to `Block == 1` above. This is the default if neither *block* nor *noblock* are specified
noblock	Is equivalent to `Block == 0` above.

3.4.10. MOV (to RX)

3.4.10.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`MOV`	1	0	0	Delay/side-set					0	0	0	1	IdxI	Index		

(*version 1 and above*)

3.4.10.2. Operation

Write the ISR to a selected RX FIFO entry. The state machine can write the RX FIFO entries in any order, indexed either by the Y register, or an immediate Index in the instruction. Requires the `SHIFTCTRL_FJOIN_RX_PUT` configuration field to be set, otherwise its operation is undefined. The FIFO configuration can be specified for the program via the `.fifo` directive (see `pioasm_fifo`).

If IdxI (index by immediate) is set, the RX FIFO's registers are indexed by the two least-significant bits of the Index operand. Otherwise, they are indexed by the two least-significant bits of the Y register. When IdxI is clear, all nonzero values of Index are reserved encodings, and their operation is undefined.

When only `SHIFTCTRL_FJOIN_RX_PUT` is set (in `SM0_SHIFTCTRL` through `SM3_SHIFTCTRL`), the system can also read the RX FIFO registers with random access via `RXF0_PUTGET0` through `RXF0_PUTGET3` (where RXFx indicates which state machine's FIFO is being accessed). In this state, the FIFO register storage is repurposed as status registers, which the state machine can update at any time and the system can read at any time. For example, a quadrature decoder program could maintain the current step count in a status register at all times, rather than pushing to the RX FIFO and potentially blocking.

When both `SHIFTCTRL_FJOIN_RX_PUT` and `SHIFTCTRL_FJOIN_RX_GET` are set, the system can no longer access the RX FIFO storage registers, but the state machine can now put/get the registers in arbitrary order, allowing them to be used as additional scratch storage.

NOTE

> The RX FIFO storage registers have only a single read port and write port, and access through each port is assigned to only one of (system, state machine) at any time.

3.4.10.3. Assembler Syntax

mov rxfifo[y], isr

mov rxfifo[<index>], isr

where:

y	Is the literal token "y", indicating the RX FIFO entry is indexed by the Y register
<index>	Is a value (see Section 3.3.3) specifying the RX FIFO entry to write (valid range 0-3)

3.4.11. MOV (from RX)

3.4.11.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOV	1	0	0	Delay/side-set					1	0	0	1	IdxI	Index		

(*version 1 and above*)

3.4.11.2. Operation

Read the selected RX FIFO entry into the OSR. The PIO state machine can read the FIFO entries in any order, indexed either by the Y register, or an immediate Index in the instruction. Requires the `SHIFTCTRL_FJOIN_RX_GET` configuration field to be set, otherwise its operation is undefined.

If IdxI (index by immediate) is set, the RX FIFO's registers are indexed by the two least-significant bits of the Index operand. Otherwise, they are indexed by the two least-significant bits of the Y register. When IdxI is clear, all nonzero values of Index are reserved encodings, and their operation is undefined.

When only `SHIFTCTRL_FJOIN_RX_GET` is set, the system can also write the RX FIFO registers with random access via `RXF0_PUTGET0` through `RXF0_PUTGET3` (where RXFx indicates which state machine's FIFO is being accessed). In this state, the RX FIFO register storage is repurposed as additional configuration registers, which the system can update at any time and the state machine can read at any time. For example, a UART TX program might use these registers to configure the number of data bits, or the presence of an additional stop bit.

When both `SHIFTCTRL_FJOIN_RX_PUT` and `SHIFTCTRL_FJOIN_RX_GET` are set, the system can no longer access the RX FIFO storage registers, but the state machine can now put/get the registers in arbitrary order, allowing them to be used as additional scratch storage.

NOTE

> The RX FIFO storage registers have only a single read port and write port, and access through each port is assigned to only one of (system, state machine) at any time.

3.4.11.3. Assembler Syntax

mov osr, rxfifo[y]

mov osr, rxfifo[<index>]

where:

y	Is the literal token "y", indicating the RX FIFO entry is indexed by the Y register
<index>	Is a value (see Section 3.3.3) specifying the RX FIFO entry to read (valid range 0-3)

3.4.12. MOV

3.4.12.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOV	1	0	1	Delay/side-set					Destination			Op		Source		

3.4.12.2. Operation

Copy data from `Source` to `Destination`.

- Destination:
  - 000: `PINS` (Uses same pin mapping as `OUT`)
  - 001: `X` (Scratch register X)
  - 010: `Y` (Scratch register Y)
  - 011: (*version 1 and above*) `PINDIRS` (Uses same pin mapping as `OUT`)
  - 100: `EXEC` (Execute data as instruction)
  - 101: `PC`
  - 110: `ISR` (Input shift counter is reset to 0 by this operation, i.e. empty)
  - 111: `OSR` (Output shift counter is reset to 0 by this operation, i.e. full)
- Operation:
  - 00: None
  - 01: Invert (bitwise complement)
  - 10: Bit-reverse
  - 11: Reserved
- Source:
  - 000: `PINS` (Uses same pin mapping as `IN`)
  - 001: `X`
  - 010: `Y`
  - 011: `NULL`
  - 100: Reserved
  - 101: `STATUS`
  - 110: `ISR`
  - 111: `OSR`

`MOV PC` causes an unconditional jump. `MOV EXEC` has the same behaviour as `OUT EXEC` (Section 3.4.7), and allows register contents to be executed as an instruction. The `MOV` itself executes in 1 cycle, and the instruction in `Source` on the next cycle. Delay cycles on `MOV EXEC` are ignored, but the executee may insert delay cycles as normal.

The `STATUS` source has a value of all-ones or all-zeroes, depending on some state machine status such as FIFO full/empty, configured by `EXECCTRL_STATUS_SEL`.

`MOV` can manipulate the transferred data in limited ways, specified by the `Operation` argument. Invert sets each bit in `Destination` to the logical NOT of the corresponding bit in `Source`, i.e. 1 bits become 0 bits, and vice versa. Bit reverse sets each bit *n* in `Destination` to bit 31 - *n* in `Source`, assuming the bits are numbered 0 to 31.

`MOV dst, PINS` reads pins using the `IN` pin mapping, and writes the full 32-bit value to the destination without masking. The LSB of the read value is the pin indicated by `PINCTRL_IN_BASE`, and each successive bit comes from a higher-numbered pin, wrapping after 31.

3.4.12.3. Assembler Syntax

mov <destination>, (op) <source>

where:

<destination>	Is one of the destinations specified above.
<op>	If present, is: `!` or `~` for NOT (Note: this is always a bitwise NOT) `::` for bit reverse
<source>	Is one of the sources specified above.

3.4.13. IRQ

3.4.13.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`IRQ`	1	1	0	Delay/side-set					0	Clr	Wait	IdxMode		Index		

3.4.13.2. Operation

Set or clear the IRQ flag selected by `Index` argument. * Clear: if 1, clear the flag selected by `Index`, instead of raising it. If `Clear` is set, the `Wait` bit has no effect. * Wait: if 1, halt until the raised flag is lowered again, e.g. if a system interrupt handler has acknowledged the flag. * Index: specifies an IRQ index from 0-7. This IRQ flag will be set/cleared depending on the Clear bit. * IdxMode: modify the behaviour if the Index field, either modifying the index, or indexing IRQ flags from a different PIO block: **`00`: the three LSBs are used directly to index the IRQ flags in this PIO block.** `01` (*version 1 and above*) (`PREV`): the instruction references an IRQ flag from the next-lower-numbered PIO in the system, wrapping to the highest-numbered PIO if this is PIO0. **`10` (`REL`): the state machine ID (0…3) is added to the IRQ flag index, by way of modulo-4 addition on the two LSBs. For example, state machine 2 with a flag value of '0x11' will wait on flag 3, and a flag value of '0x13' will wait on flag 1. This allows multiple state machines running the same program to synchronise with each other.** `11` (*version 1 and above*) (`NEXT`): the instruction references an IRQ flag from the next-higher-numbered PIO in the system, wrapping to PIO0 if this is the highest-numbered PIO.

On PIO version 0, IRQ flags 4-7 are visible only to the state machines; IRQ flags 0-3 can be routed out to system level interrupts, on either of the PIO's two external interrupt request lines, configured by `IRQ0_INTE` and `IRQ1_INTE`. PIO version 1 lifts this limitation and allows all eight flags to assert system interrupts.

The modulo addition mode allows relative addressing of 'IRQ' and 'WAIT' instructions, for synchronising state machines which are running the same program. Bit 2 (the third LSB) is unaffected by this addition.

The modulo addition mode (`REL`) allows relative addressing of 'IRQ' and 'WAIT' instructions, for synchronising state machines which are running the same program. Bit 2 (the third LSB) is unaffected by this addition.

The `NEXT`/`PREV` modes (*version 1 and above*) can be used to synchronise between state machines in different PIO blocks. If these state machines' clocks are divided, their clock dividers must be the same, and must have been synchronised by writing `CTRL.NEXTPREV_CLKDIV_RESTART` in addition to the relevant NEXT_PIO_MASK/PREV_PIO_MASK bits. Note that the cross-PIO connection is severed between PIOs with different accessibility to Non-secure code, as per ACCESSCTRL.

If `Wait` is set, `Delay` cycles do not begin until after the wait period elapses.

3.4.13.3. Assembler Syntax

irq <irq_num> (*rel*)

irq set <irq_num> (*rel*)

irq nowait <irq_num> (*rel*)

irq wait <irq_num> (rel)

irq clear <irq_num> (rel)

where:

<irq_num> (rel)	Is a value (see Section 3.3.3) specifying The irq number to wait on (0-7). If *rel* is present, then the actual irq number used is calculating by replacing the low two bits of the irq number (irq_num_{10}) with the low two bits of the sum (irq_num_{10} + sm_num_{10}) where sm_num_{10} is the state machine number
irq	Means set the IRQ without waiting
irq set	Also means set the IRQ without waiting
irq nowait	Again, means set the IRQ without waiting
irq wait	Means set the IRQ and wait for it to be cleared before proceeding
irq clear	Means clear the IRQ

3.4.14. SET

3.4.14.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`SET`	1	1	1	Delay/side-set					Destination			Data				

3.4.14.2. Operation

Write immediate value `Data` to `Destination`.

- Destination:
  - 000: `PINS`
  - 001: `X` (scratch register X) 5 LSBs are set to `Data`, all others cleared to 0.
  - 010: `Y` (scratch register Y) 5 LSBs are set to `Data`, all others cleared to 0.
  - 011: Reserved
  - 100: `PINDIRS`
  - 101: Reserved
  - 110: Reserved
  - 111: Reserved
- Data: 5-bit immediate value to drive to pins or register.

This can be used to assert control signals such as a clock or chip select, or to initialise loop counters. As `Data` is 5 bits in size, scratch registers can be `SET` to values from 0-31, which is sufficient for a 32-iteration loop.

The mapping of `SET` and `OUT` onto pins is configured independently. They may be mapped to distinct locations, for example if one pin is to be used as a clock signal, and another for data. They may also be overlapping ranges of pins: a UART transmitter might use `SET` to assert start and stop bits, and `OUT` instructions to shift out FIFO data to the same pins.

3.4.14.3. Assembler Syntax

set <destination>, <value>

where:

<destination>	Is one of the destinations specified above.
<value>	The value (see Section 3.3.3) to set (valid range 0-31)

Chapter 4. Library documentation

Full library API documentation can also be found online at https://www.raspberrypi.com/documentation/pico-sdk/

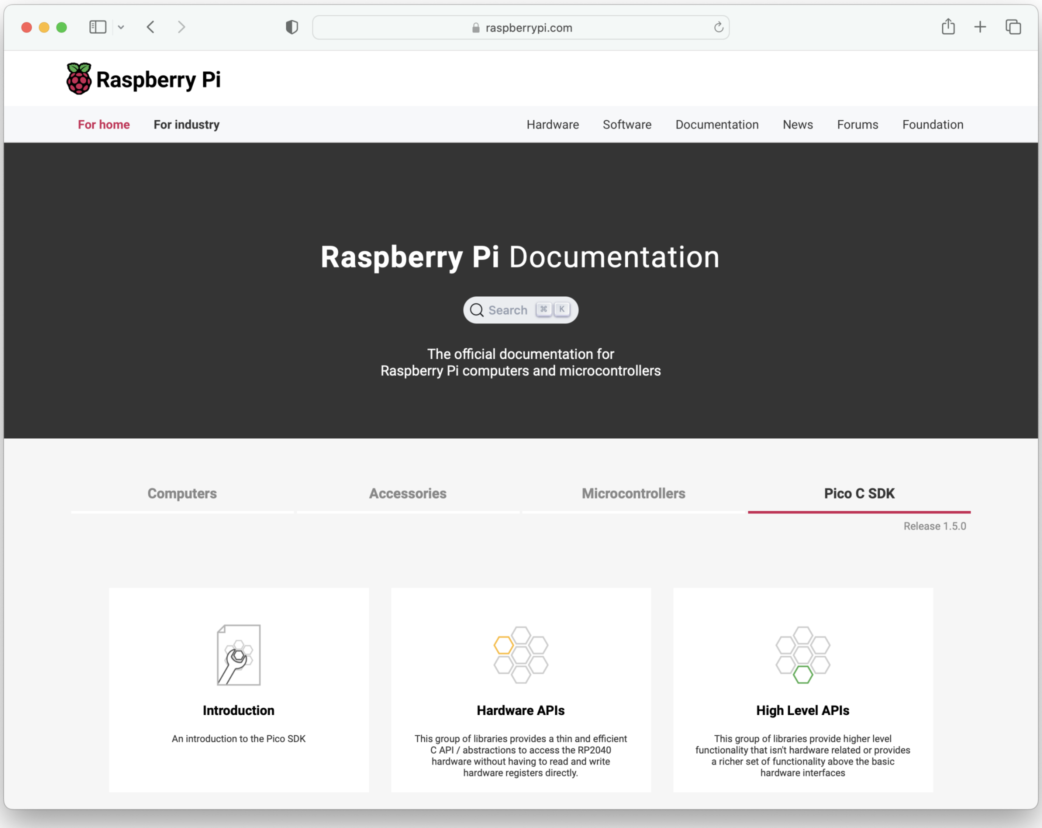


Figure 8. The Raspberry Pi documentation site.

> **NOTE**
>
> You can also build the API documentation locally, see Appendix B.

4.1. Hardware APIs

This group of libraries provides a thin and efficient C API / abstractions to access the RP-series microcontroller hardware without having to read and write hardware registers directly.

hardware_adc	Analog to Digital Converter (ADC) API.
hardware_base	Low-level types and (atomic) accessors for memory-mapped hardware registers.
hardware_claim	Lightweight hardware resource management API.
hardware_clocks	Clock Management API.
hardware_divider	RP2040 Low Low-level hardware-divider API. Non-RP2040 platforms provide software versions of all the functions.
hardware_dcp	Assembly macros for the Double Coprocessor. RP2350
hardware_dma	DMA Controller API.
channel_config	DMA channel configuration.
hardware_exception	Methods for setting processor exception handlers.
hardware_flash	Low level flash programming and erase API.
hardware_gpio	General Purpose Input/Output (GPIO) API.
hardware_hazard3	Accessors for Hazard3-specific RISC-V CSRs, and intrinsics for Hazard3 custom instructions. RP2350
hardware_i2c	I2C Controller API.
hardware_interp	Hardware Interpolator API.
interp_config	Interpolator configuration.
hardware_irq	Hardware interrupt handling API.
hardware_pio	Programmable I/O (PIO) API.
sm_config	PIO state machine configuration.
pio_instructions	PIO instruction encoding.
hardware_pll	Phase Locked Loop control APIs.
hardware_powman	Power Management API. RP2350
hardware_pwm	Hardware Pulse Width Modulation (PWM) API.
hardware_resets	Hardware Reset API.
hardware_riscv	Accessors for standard RISC-V hardware (mainly CSRs) RP2350
hardware_riscv_platform_timer	Accessors for standard RISC-V platform timer (mtime/mtimecmp), available on Raspberry Pi microcontrollers with RISC-V processors. RP2350
hardware_rtc	Hardware Real Time Clock API. RP2040
hardware_rcp	Inline functions and assembly macros for the Redundancy Coprocessor. RP2350
hardware_spi	Hardware SPI API.
hardware_sha256	Hardware SHA-256 Accelerator API. RP2350
hardware_sync	Low level hardware spin locks, barrier and processor event APIs.
hardware_ticks	Hardware Tick 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_xip_cache	Low-level cache maintenance operations for the XIP cache.
hardware_xosc	Crystal Oscillator (XOSC) API.

4.1.1. hardware_adc

Analog to Digital Converter (ADC) API.

4.1.1.1. Detailed Description

RP-series microcontrollers have an internal analogue-digital converter (ADC) with the following features:

- SAR ADC
- 500 kS/s (Using an independent 48MHz clock)
- 12 bit (RP2040 8.7 ENOB, RP2350 9.2 ENOB)
- RP2040 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
- RP2350 5 or 9 input mux:
  - 4 inputs available on QFN-60 package pins shared with GPIO[29:26]
  - 8 inputs available on QFN-80 package pins shared with GPIO[47:40]
  - 8 element receive sample FIFO
- One input dedicated to the internal temperature sensor (see Section 12.4.6)
- 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_set_round_robin()), the ADC will use that input and move to the next one after a read.

RP2040, RP2350 QFN-60: User ADC inputs are on 0-3 (GPIO 26-29), the temperature sensor is on input 4. RP2350 QFN-80 : User ADC inputs are on 0-7 (GPIO 40-47), the temperature sensor is on input 8.

Temperature sensor values can be approximated in centigrade as:

T = 27 - (ADC_Voltage - 0.706)/0.001721

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);
    }
}
```

4.1.1.2. 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.

`static void adc_select_input (uint input)`

ADC input select.

`static uint adc_get_selected_input (void)`

Get the currently selected ADC input channel.

`static void adc_set_round_robin (uint input_mask)`

Round Robin sampling selector.

`static void adc_set_temp_sensor_enabled (bool enable)`

Enable the onboard temperature sensor.

`static uint16_t adc_read (void)`

Perform a single conversion.

`static void adc_run (bool run)`

Enable or disable free-running sampling mode.

`static void adc_set_clkdiv (float clkdiv)`

Set the ADC Clock divisor.

`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.

`static bool adc_fifo_is_empty (void)`

Check FIFO empty state.

`static uint8_t adc_fifo_get_level (void)`

Get number of entries in the ADC FIFO.

`static uint16_t adc_fifo_get (void)`

Get ADC result from FIFO.

`static uint16_t adc_fifo_get_blocking (void)`

Wait for the ADC FIFO to have data.

`static void adc_fifo_drain (void)`

Drain the ADC FIFO.

`static void adc_irq_set_enabled (bool enabled)`

Enable/Disable ADC interrupts.

4.1.1.3. Function Documentation

4.1.1.3.1. adc_fifo_drain

`static void adc_fifo_drain (void) [inline], [static]`

Drain the ADC FIFO.

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

4.1.1.3.2. adc_fifo_get

`static uint16_t adc_fifo_get (void) [inline], [static]`

Get ADC result from FIFO.

Pops the latest result from the ADC FIFO.

4.1.1.3.3. adc_fifo_get_blocking

`static uint16_t adc_fifo_get_blocking (void) [inline], [static]`

Wait for the ADC FIFO to have data.

Blocks until data is present in the FIFO

4.1.1.3.4. adc_fifo_get_level

`static uint8_t adc_fifo_get_level (void) [inline], [static]`

Get number of entries in the ADC FIFO.

On RP2040 the FIFO is 4 samples long. On RP2350 the FIFO is 8 samples long.

This function will return how many samples are currently present.

4.1.1.3.5. adc_fifo_is_empty

`static bool adc_fifo_is_empty (void) [inline], [static]`

Check FIFO empty state.

Returns

Returns true if the FIFO is empty

4.1.1.3.6. adc_fifo_setup

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

Setup the ADC FIFO.

On RP2040 the FIFO is 4 samples long.

On RP2350 the FIFO is 8 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.

4.1.1.3.7. adc_get_selected_input

`static uint adc_get_selected_input (void) [inline], [static]`

Get the currently selected ADC input channel.

Returns

The currently selected input channel.

On RP2040 0…3 are GPIOs 26…29 respectively. Input 4 is the onboard temperature sensor.

On RP2350A 0…3 are GPIOs 26…29 respectively. Input 4 is the onboard temperature sensor. On RP2350B 0…7 are GPIOs 40…47 respectively. Input 8 is the onboard temperature sensor.

4.1.1.3.8. adc_gpio_init

`static void adc_gpio_init (uint gpio) [inline], [static]`

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 on RP2040 or RP2350A, 40-48 inclusive on RP2350B

4.1.1.3.9. adc_init

`void adc_init (void)`

Initialise the ADC HW.

4.1.1.3.10. adc_irq_set_enabled

`static void adc_irq_set_enabled (bool enabled) [inline], [static]`

Enable/Disable ADC interrupts.

Parameters

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

4.1.1.3.11. adc_read

`static uint16_t adc_read (void) [inline], [static]`

Perform a single conversion.

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

Returns

Result of the conversion.

4.1.1.3.12. adc_run

`static void adc_run (bool run) [inline], [static]`

Enable or disable free-running sampling mode.

Parameters

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

4.1.1.3.13. adc_select_input

`static void adc_select_input (uint input) [inline], [static]`

ADC input select.

Select an ADC input On RP2040 0…3 are GPIOs 26…29 respectively. Input 4 is the onboard temperature sensor. On RP2350A 0…3 are GPIOs 26…29 respectively. Input 4 is the onboard temperature sensor. On RP2350B 0…7 are GPIOs 40…47 respectively. Input 8 is the onboard temperature sensor.

Parameters

`input`	Input to select.

4.1.1.3.14. adc_set_clkdiv

`static void adc_set_clkdiv (float clkdiv) [inline], [static]`

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.

4.1.1.3.15. adc_set_round_robin

`static void adc_set_round_robin (uint input_mask) [inline], [static]`

Round Robin sampling selector.

This function sets which inputs are to be run through in round robin mode. RP2040, RP2350 QFN-60: Value between 0 and 0x1f (bit 0 to bit 4 for GPIO 26 to 29 and temperature sensor input respectively) RP2350 QFN-80: Value between 0 and 0xff (bit 0 to bit 7 for GPIO 40 to 47 and temperature sensor input respectively)

Parameters

`input_mask`	A bit pattern indicating which of the 5/8 inputs are to be sampled. Write a value of 0 to disable round robin sampling.

4.1.1.3.16. adc_set_temp_sensor_enabled

```
static void adc_set_temp_sensor_enabled (bool enable) [inline], [static]
```

Enable the onboard temperature sensor.

Parameters

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

4.1.2. hardware_base

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

4.1.2.1. Detailed Description

`hardware_base` defines the low level types and access functions for memory mapped hardware registers. It is included 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
		16	16-bit wide access
		32	32-bit wide access

When dealing with these types, you will always use 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`.

RP-series 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.

4.1.2.2. Functions

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

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

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

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

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

Atomically flip the specified bits in a HW register.

`static __force_inline 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.

4.1.2.3. Function Documentation

4.1.2.3.1. hw_clear_bits

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

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

Parameters

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

4.1.2.3.2. hw_set_bits

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

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

Parameters

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

4.1.2.3.3. hw_write_masked

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

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 *different* 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

4.1.2.3.4. hw_xor_bits

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

Atomically flip the specified bits in a HW register.

Parameters

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

4.1.3. hardware_claim

Lightweight hardware resource management API.

4.1.3.1. Detailed Description

`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:

1. Claiming resources (and asserting if they are already claimed)
2. Freeing (unclaiming) resources
3. Finding unused resources

4.1.3.2. 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.

`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.

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

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

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

Atomically unclaim a resource.

`uint32_t hw_claim_lock (void)`

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

`void hw_claim_unlock (uint32_t token)`

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

4.1.3.3. Function Documentation

4.1.3.3.1. 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)

4.1.3.3.2. hw_claim_lock

`uint32_t hw_claim_lock (void)`

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.

> **NOTE**
>
> hw_claim_lock() uses a spin lock internally, so disables interrupts on the calling core, and will deadlock if the calling core already owns the lock.

Returns

a token to pass to hw_claim_unlock()

4.1.3.3.3. 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

4.1.3.3.4. hw_claim_unlock

`void hw_claim_unlock (uint32_t token)`

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

> **NOTE**
>
> This method MUST be called from the same core that call hw_claim_lock()

Parameters

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

4.1.3.3.5. 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 resource is not free
`bit_lsb`	the lower bound (inclusive) of the resource range to claim from
`bit_msb`	the upper 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 claimed or -1 if none are available in the range, and required = false

4.1.3.3.6. hw_is_claimed

`bool hw_is_claimed (const uint8_t * bits, uint bit_index) [inline]`

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 check (bit index into array of bits)

Returns

true if the resource is claimed

4.1.4. hardware_clocks

Clock Management API.

4.1.4.1. Detailed Description

This API provides 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 appropriate datasheet for more details on the RP-series 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.

- On RP2040 the clock sources are:

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_XOSC_CLKSRC
System clock	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLK_SYS		
USB Clock	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLK_USB		

Aux Source	clk_gpout[x]	clk_ref	clk_sys
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
XOSC	CLOCKS_CLK_RTC_CTRL_AUXSRC_VALUE_XOSC_CLKSRC

On RP2350 the clock sources are:

- **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	
LPOSC	CLOCKS_CLK_REF_CTRL_SRC_VALUE_LPOSC_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_XOSC_CLKSRC
LPOSC	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_LPOSC_CLKSRC		
System clock	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLK_SYS		
USB Clock	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLK_USB		

Aux Source	clk_gpout[x]	clk_ref	clk_sys
ADC clock	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLK_ADC		
REF clock	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLK_REF		
PERI clock	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLK_PERI		
HSTX clock	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLK_PERI		

Aux Source	clk_peri	clk_hstx	clk_usb	clk_adc
System PLL	CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS	CLOCKS_CLK_HSTX_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_HSTX_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	CLOCKS_CLK_HSTX_CTRL_AUXSRC_VALUE_CLK_SYS		

4.1.4.1.1. Example

```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);
#ifdef CLOCKS_FC0_SRC_VALUE_CLK_RTC
    uint f_clk_rtc = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_RTC);
#endif

    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);
#ifdef CLOCKS_FC0_SRC_VALUE_CLK_RTC
    printf("clk_rtc  = %dkHz\n", f_clk_rtc);
#endif

    // 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;
}
```

4.1.4.2. Typedefs

`typedef enum clock_num_rp2040 clock_num_t` RP2040

Clock numbers on RP2040 (used as typedef clock_num_t)

`typedef enum clock_dest_num_rp2040 clock_dest_num_t` RP2040

Clock destination numbers on RP2040 (used as typedef clock_dest_num_t)

`typedef enum clock_num_rp2350 clock_num_t` RP2350

Clock numbers on RP2350 (used as typedef clock_num_t)

`typedef enum clock_dest_num_rp2350 clock_dest_num_t` RP2350

Clock destination numbers on RP2350 (used as typedef clock_dest_num_t)

`typedef void(* resus_callback_t)(void)`

Resus callback function type.

4.1.4.3. Enumerations

`enum clock_num_rp2040 { clk_gpout0 = 0, clk_gpout1 = 1, clk_gpout2 = 2, clk_gpout3 = 3, clk_ref = 4, clk_sys = 5, clk_peri = 6, clk_usb = 7, clk_adc = 8, clk_rtc = 9, CLK_COUNT }` RP2040

Clock numbers on RP2040 (used as typedef clock_num_t)

`enum clock_dest_num_rp2040 { CLK_DEST_SYS_CLOCKS = 0, CLK_DEST_ADC_ADC = 1, CLK_DEST_SYS_ADC = 2, CLK_DEST_SYS_BUSCTRL = 3, CLK_DEST_SYS_BUSFABRIC = 4, CLK_DEST_SYS_DMA = 5, CLK_DEST_SYS_I2C0 = 6, CLK_DEST_SYS_I2C1 = 7, CLK_DEST_SYS_IO = 8, CLK_DEST_SYS_JTAG = 9, CLK_DEST_SYS_VREG_AND_CHIP_RESET = 10, CLK_DEST_SYS_PADS = 11, CLK_DEST_SYS_PIO0 = 12, CLK_DEST_SYS_PIO1 = 13, CLK_DEST_SYS_PLL_SYS = 14, CLK_DEST_SYS_PLL_USB = 15, CLK_DEST_SYS_PSM = 16, CLK_DEST_SYS_PWM = 17, CLK_DEST_SYS_RESETS = 18, CLK_DEST_SYS_ROM = 19, CLK_DEST_SYS_ROSC = 20, CLK_DEST_RTC_RTC = 21, CLK_DEST_SYS_RTC = 22, CLK_DEST_SYS_SIO = 23, CLK_DEST_PERI_SPI0 = 24, CLK_DEST_SYS_SPI0 = 25, CLK_DEST_PERI_SPI1 = 26, CLK_DEST_SYS_SPI1 = 27, CLK_DEST_SYS_SRAM0 = 28, CLK_DEST_SYS_SRAM1 = 29, CLK_DEST_SYS_SRAM2 = 30, CLK_DEST_SYS_SRAM3 = 31, CLK_DEST_SYS_SRAM4 = 32, CLK_DEST_SYS_SRAM5 = 33, CLK_DEST_SYS_SYSCFG = 34, CLK_DEST_SYS_SYSINFO = 35, CLK_DEST_SYS_TBMAN = 36, CLK_DEST_SYS_TIMER = 37, CLK_DEST_PERI_UART0 = 38, CLK_DEST_SYS_UART0 = 39, CLK_DEST_PERI_UART1 = 40, CLK_DEST_SYS_UART1 = 41, CLK_DEST_SYS_USBCTRL = 42, CLK_DEST_USB_USBCTRL = 43, CLK_DEST_SYS_WATCHDOG = 44, CLK_DEST_SYS_XIP = 45, CLK_DEST_SYS_XOSC = 46, NUM_CLOCK_DESTINATIONS }` RP2040

Clock destination numbers on RP2040 (used as typedef clock_dest_num_t)

`enum clock_num_rp2350 { clk_gpout0 = 0, clk_gpout1 = 1, clk_gpout2 = 2, clk_gpout3 = 3, clk_ref = 4, clk_sys = 5, clk_peri = 6, clk_hstx = 7, clk_usb = 8, clk_adc = 9, CLK_COUNT }` RP2350

Clock numbers on RP2350 (used as typedef clock_num_t)

`enum clock_dest_num_rp2350 { CLK_DEST_SYS_CLOCKS = 0, CLK_DEST_SYS_ACCESSCTRL = 1, CLK_DEST_ADC = 2, CLK_DEST_SYS_ADC = 3, CLK_DEST_SYS_BOOTRAM = 4, CLK_DEST_SYS_BUSCTRL = 5, CLK_DEST_SYS_BUSFABRIC = 6, CLK_DEST_SYS_DMA = 7, CLK_DEST_SYS_GLITCH_DETECTOR = 8, CLK_DEST_HSTX = 9, CLK_DEST_SYS_HSTX = 10, CLK_DEST_SYS_I2C0 = 11, CLK_DEST_SYS_I2C1 = 12, CLK_DEST_SYS_IO = 13, CLK_DEST_SYS_JTAG = 14, CLK_DEST_REF_OTP = 15, CLK_DEST_SYS_OTP = 16, CLK_DEST_SYS_PADS = 17, CLK_DEST_SYS_PIO0 = 18, CLK_DEST_SYS_PIO1 = 19, CLK_DEST_SYS_PIO2 = 20, CLK_DEST_SYS_PLL_SYS = 21, CLK_DEST_SYS_PLL_USB = 22, CLK_DEST_REF_POWMAN = 23, CLK_DEST_SYS_POWMAN = 24, CLK_DEST_SYS_PWM = 25, CLK_DEST_SYS_RESETS = 26, CLK_DEST_SYS_ROM = 27, CLK_DEST_SYS_ROSC = 28, CLK_DEST_SYS_PSM = 29, CLK_DEST_SYS_SHA256 = 30, CLK_DEST_SYS_SIO = 31, CLK_DEST_PERI_SPI0 = 32, CLK_DEST_SYS_SPI0 = 33, CLK_DEST_PERI_SPI1 = 34, CLK_DEST_SYS_SPI1 = 35, CLK_DEST_SYS_SRAM0 = 36, CLK_DEST_SYS_SRAM1 = 37, CLK_DEST_SYS_SRAM2 = 38, CLK_DEST_SYS_SRAM3 = 39, CLK_DEST_SYS_SRAM4 = 40, CLK_DEST_SYS_SRAM5 = 41, CLK_DEST_SYS_SRAM6 = 42, CLK_DEST_SYS_SRAM7 = 43, CLK_DEST_SYS_SRAM8 = 44, CLK_DEST_SYS_SRAM9 = 45, CLK_DEST_SYS_SYSCFG = 46, CLK_DEST_SYS_SYSINFO = 47, CLK_DEST_SYS_TBMAN = 48, CLK_DEST_REF_TICKS = 49, CLK_DEST_SYS_TICKS = 50, CLK_DEST_SYS_TIMER0 = 51, CLK_DEST_SYS_TIMER1 = 52, CLK_DEST_SYS_TRNG = 53, CLK_DEST_PERI_UART0 = 54, CLK_DEST_SYS_UART0 = 55, CLK_DEST_PERI_UART1 = 56, CLK_DEST_SYS_UART1 = 57, CLK_DEST_SYS_USBCTRL = 58, CLK_DEST_USB = 59, CLK_DEST_SYS_WATCHDOG = 60, CLK_DEST_SYS_XIP = 61, CLK_DEST_SYS_XOSC = 62, NUM_CLOCK_DESTINATIONS }` RP2350

Clock destination numbers on RP2350 (used as typedef clock_dest_num_t)

4.1.4.4. Functions

`bool clock_configure (clock_handle_t clock, uint32_t src, uint32_t auxsrc, uint32_t src_freq, uint32_t freq)`

Configure the specified clock with automatic clock divisor setup.

`void clock_configure_undivided (clock_handle_t clock, uint32_t src, uint32_t auxsrc, uint32_t src_freq)`

Configure the specified clock to use the undivided input source.

`void clock_configure_int_divider (clock_handle_t clock, uint32_t src, uint32_t auxsrc, uint32_t src_freq, uint32_t int_divider)`

Configure the specified clock to use the undivided input source.

`void clock_stop (clock_handle_t clock)`

Stop the specified clock.

`uint32_t clock_get_hz (clock_handle_t clock)`

Get the current frequency of the specified clock.

`uint32_t frequency_count_khz (uint src)`

Measure a clocks frequency using the Frequency counter.

`void clock_set_reported_hz (clock_handle_t clock, uint hz)`

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

`void clocks_enable_resus (resus_callback_t resus_callback)`

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

`void clock_gpio_init_int_frac16 (uint gpio, uint src, uint32_t div_int, uint16_t div_frac16)`

Output an optionally divided clock to the specified gpio pin.

`static void clock_gpio_init_int_frac8 (uint gpio, uint src, uint32_t div_int, uint8_t div_frac8)`

Output an optionally divided clock to the specified gpio pin.

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

Output an optionally divided clock to the specified gpio pin.

`bool clock_configure_gpin (clock_handle_t clock, uint gpio, uint32_t src_freq, uint32_t freq)`

Configure a clock to come from a gpio input.

`void set_sys_clock_48mhz (void)`

Initialise the system clock to 48MHz.

`void set_sys_clock_pll (uint32_t vco_freq, uint post_div1, uint post_div2)`

Initialise the system clock.

`bool check_sys_clock_hz (uint32_t freq_hz, uint *vco_freq_out, uint *post_div1_out, uint *post_div2_out)`

Check if a given system clock frequency is valid/attainable.

`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.

`static bool set_sys_clock_hz (uint32_t freq_hz, bool required)`

Attempt to set a system clock frequency in hz.

`static bool set_sys_clock_khz (uint32_t freq_khz, bool required)`

Attempt to set a system clock frequency in khz.

4.1.4.5. Typedef Documentation

4.1.4.5.1. clock_num_t RP2040

```
typedef enum clock_num_rp2040 clock_num_t
```

Clock numbers on RP2040 (used as typedef clock_num_t)

4.1.4.5.2. clock_dest_num_t RP2040

```
typedef enum clock_dest_num_rp2040 clock_dest_num_t
```

Clock destination numbers on RP2040 (used as typedef clock_dest_num_t)

4.1.4.5.3. clock_num_t RP2350

```
typedef enum clock_num_rp2350 clock_num_t
```

Clock numbers on RP2350 (used as typedef clock_num_t)

4.1.4.5.4. clock_dest_num_t RP2350

```
typedef enum clock_dest_num_rp2350 clock_dest_num_t
```

Clock destination numbers on RP2350 (used as typedef clock_dest_num_t)

4.1.4.5.5. resus_callback_t

```
typedef void(* resus_callback_t) (void)
```

Resus callback function type.

User provided callback for a resus event (when clk_sys is stopped by the programmer and is restarted for them).

4.1.4.6. Enumeration Type Documentation

4.1.4.6.1. clock_num_rp2040 RP2040

```
enum clock_num_rp2040
```

Clock numbers on RP2040 (used as typedef clock_num_t)

Table 9. Enumerator

clk_gpout0	Select CLK_GPOUT0 as clock source.
clk_gpout1	Select CLK_GPOUT1 as clock source.
clk_gpout2	Select CLK_GPOUT2 as clock source.
clk_gpout3	Select CLK_GPOUT3 as clock source.
clk_ref	Select CLK_REF as clock source.
clk_sys	Select CLK_SYS as clock source.
clk_peri	Select CLK_PERI as clock source.
clk_usb	Select CLK_USB as clock source.

clk_adc	Select CLK_ADC as clock source.
clk_rtc	Select CLK_RTC as clock source.

4.1.4.6.2. clock_dest_num_rp2040 RP2040

`enum clock_dest_num_rp2040`

Clock destination numbers on RP2040 (used as typedef clock_dest_num_t)

Table 10. Enumerator

CLK_DEST_SYS_CLOCKS	Select SYS_CLOCKS as clock destination.
CLK_DEST_ADC_ADC	Select ADC_ADC as clock destination.
CLK_DEST_SYS_ADC	Select SYS_ADC as clock destination.
CLK_DEST_SYS_BUSCTRL	Select SYS_BUSCTRL as clock destination.
CLK_DEST_SYS_BUSFABRIC	Select SYS_BUSFABRIC as clock destination.
CLK_DEST_SYS_DMA	Select SYS_DMA as clock destination.
CLK_DEST_SYS_I2C0	Select SYS_I2C0 as clock destination.
CLK_DEST_SYS_I2C1	Select SYS_I2C1 as clock destination.
CLK_DEST_SYS_IO	Select SYS_IO as clock destination.
CLK_DEST_SYS_JTAG	Select SYS_JTAG as clock destination.
CLK_DEST_SYS_VREG_AND_CHIP_RESET	Select SYS_VREG_AND_CHIP_RESET as clock destination.
CLK_DEST_SYS_PADS	Select SYS_PADS as clock destination.
CLK_DEST_SYS_PIO0	Select SYS_PIO0 as clock destination.
CLK_DEST_SYS_PIO1	Select SYS_PIO1 as clock destination.
CLK_DEST_SYS_PLL_SYS	Select SYS_PLL_SYS as clock destination.
CLK_DEST_SYS_PLL_USB	Select SYS_PLL_USB as clock destination.
CLK_DEST_SYS_PSM	Select SYS_PSM as clock destination.
CLK_DEST_SYS_PWM	Select SYS_PWM as clock destination.
CLK_DEST_SYS_RESETS	Select SYS_RESETS as clock destination.
CLK_DEST_SYS_ROM	Select SYS_ROM as clock destination.
CLK_DEST_SYS_ROSC	Select SYS_ROSC as clock destination.
CLK_DEST_RTC_RTC	Select RTC_RTC as clock destination.
CLK_DEST_SYS_RTC	Select SYS_RTC as clock destination.
CLK_DEST_SYS_SIO	Select SYS_SIO as clock destination.
CLK_DEST_PERI_SPI0	Select PERI_SPI0 as clock destination.
CLK_DEST_SYS_SPI0	Select SYS_SPI0 as clock destination.
CLK_DEST_PERI_SPI1	Select PERI_SPI1 as clock destination.
CLK_DEST_SYS_SPI1	Select SYS_SPI1 as clock destination.
CLK_DEST_SYS_SRAM0	Select SYS_SRAM0 as clock destination.
CLK_DEST_SYS_SRAM1	Select SYS_SRAM1 as clock destination.

CLK_DEST_SYS_SRAM2	Select SYS_SRAM2 as clock destination.
CLK_DEST_SYS_SRAM3	Select SYS_SRAM3 as clock destination.
CLK_DEST_SYS_SRAM4	Select SYS_SRAM4 as clock destination.
CLK_DEST_SYS_SRAM5	Select SYS_SRAM5 as clock destination.
CLK_DEST_SYS_SYSCFG	Select SYS_SYSCFG as clock destination.
CLK_DEST_SYS_SYSINFO	Select SYS_SYSINFO as clock destination.
CLK_DEST_SYS_TBMAN	Select SYS_TBMAN as clock destination.
CLK_DEST_SYS_TIMER	Select SYS_TIMER as clock destination.
CLK_DEST_PERI_UART0	Select PERI_UART0 as clock destination.
CLK_DEST_SYS_UART0	Select SYS_UART0 as clock destination.
CLK_DEST_PERI_UART1	Select PERI_UART1 as clock destination.
CLK_DEST_SYS_UART1	Select SYS_UART1 as clock destination.
CLK_DEST_SYS_USBCTRL	Select SYS_USBCTRL as clock destination.
CLK_DEST_USB_USBCTRL	Select USB_USBCTRL as clock destination.
CLK_DEST_SYS_WATCHDOG	Select SYS_WATCHDOG as clock destination.
CLK_DEST_SYS_XIP	Select SYS_XIP as clock destination.
CLK_DEST_SYS_XOSC	Select SYS_XOSC as clock destination.

4.1.4.6.3. clock_num_rp2350 RP2350

`enum clock_num_rp2350`

Clock numbers on RP2350 (used as typedef clock_num_t)

Table 11. Enumerator

clk_gpout0	Select CLK_GPOUT0 as clock source.
clk_gpout1	Select CLK_GPOUT1 as clock source.
clk_gpout2	Select CLK_GPOUT2 as clock source.
clk_gpout3	Select CLK_GPOUT3 as clock source.
clk_ref	Select CLK_REF as clock source.
clk_sys	Select CLK_SYS as clock source.
clk_peri	Select CLK_PERI as clock source.
clk_hstx	Select CLK_HSTX as clock source.
clk_usb	Select CLK_USB as clock source.
clk_adc	Select CLK_ADC as clock source.

4.1.4.6.4. clock_dest_num_rp2350 RP2350

`enum clock_dest_num_rp2350`

Clock destination numbers on RP2350 (used as typedef clock_dest_num_t)

Table 12. Enumerator

CLK_DEST_SYS_CLOCKS	Select SYS_CLOCKS as clock destination.
CLK_DEST_SYS_ACCESSCTRL	Select SYS_ACCESSCTRL as clock destination.
CLK_DEST_ADC	Select ADC as clock destination.
CLK_DEST_SYS_ADC	Select SYS_ADC as clock destination.
CLK_DEST_SYS_BOOTRAM	Select SYS_BOOTRAM as clock destination.
CLK_DEST_SYS_BUSCTRL	Select SYS_BUSCTRL as clock destination.
CLK_DEST_SYS_BUSFABRIC	Select SYS_BUSFABRIC as clock destination.
CLK_DEST_SYS_DMA	Select SYS_DMA as clock destination.
CLK_DEST_SYS_GLITCH_DETECTOR	Select SYS_GLITCH_DETECTOR as clock destination.
CLK_DEST_HSTX	Select HSTX as clock destination.
CLK_DEST_SYS_HSTX	Select SYS_HSTX as clock destination.
CLK_DEST_SYS_I2C0	Select SYS_I2C0 as clock destination.
CLK_DEST_SYS_I2C1	Select SYS_I2C1 as clock destination.
CLK_DEST_SYS_IO	Select SYS_IO as clock destination.
CLK_DEST_SYS_JTAG	Select SYS_JTAG as clock destination.
CLK_DEST_REF_OTP	Select REF_OTP as clock destination.
CLK_DEST_SYS_OTP	Select SYS_OTP as clock destination.
CLK_DEST_SYS_PADS	Select SYS_PADS as clock destination.
CLK_DEST_SYS_PIO0	Select SYS_PIO0 as clock destination.
CLK_DEST_SYS_PIO1	Select SYS_PIO1 as clock destination.
CLK_DEST_SYS_PIO2	Select SYS_PIO2 as clock destination.
CLK_DEST_SYS_PLL_SYS	Select SYS_PLL_SYS as clock destination.
CLK_DEST_SYS_PLL_USB	Select SYS_PLL_USB as clock destination.
CLK_DEST_REF_POWMAN	Select REF_POWMAN as clock destination.
CLK_DEST_SYS_POWMAN	Select SYS_POWMAN as clock destination.
CLK_DEST_SYS_PWM	Select SYS_PWM as clock destination.
CLK_DEST_SYS_RESETS	Select SYS_RESETS as clock destination.
CLK_DEST_SYS_ROM	Select SYS_ROM as clock destination.
CLK_DEST_SYS_ROSC	Select SYS_ROSC as clock destination.
CLK_DEST_SYS_PSM	Select SYS_PSM as clock destination.
CLK_DEST_SYS_SHA256	Select SYS_SHA256 as clock destination.
CLK_DEST_SYS_SIO	Select SYS_SIO as clock destination.
CLK_DEST_PERI_SPI0	Select PERI_SPI0 as clock destination.
CLK_DEST_SYS_SPI0	Select SYS_SPI0 as clock destination.
CLK_DEST_PERI_SPI1	Select PERI_SPI1 as clock destination.
CLK_DEST_SYS_SPI1	Select SYS_SPI1 as clock destination.
CLK_DEST_SYS_SRAM0	Select SYS_SRAM0 as clock destination.

CLK_DEST_SYS_SRAM1	Select SYS_SRAM1 as clock destination.
CLK_DEST_SYS_SRAM2	Select SYS_SRAM2 as clock destination.
CLK_DEST_SYS_SRAM3	Select SYS_SRAM3 as clock destination.
CLK_DEST_SYS_SRAM4	Select SYS_SRAM4 as clock destination.
CLK_DEST_SYS_SRAM5	Select SYS_SRAM5 as clock destination.
CLK_DEST_SYS_SRAM6	Select SYS_SRAM6 as clock destination.
CLK_DEST_SYS_SRAM7	Select SYS_SRAM7 as clock destination.
CLK_DEST_SYS_SRAM8	Select SYS_SRAM8 as clock destination.
CLK_DEST_SYS_SRAM9	Select SYS_SRAM9 as clock destination.
CLK_DEST_SYS_SYSCFG	Select SYS_SYSCFG as clock destination.
CLK_DEST_SYS_SYSINFO	Select SYS_SYSINFO as clock destination.
CLK_DEST_SYS_TBMAN	Select SYS_TBMAN as clock destination.
CLK_DEST_REF_TICKS	Select REF_TICKS as clock destination.
CLK_DEST_SYS_TICKS	Select SYS_TICKS as clock destination.
CLK_DEST_SYS_TIMER0	Select SYS_TIMER0 as clock destination.
CLK_DEST_SYS_TIMER1	Select SYS_TIMER1 as clock destination.
CLK_DEST_SYS_TRNG	Select SYS_TRNG as clock destination.
CLK_DEST_PERI_UART0	Select PERI_UART0 as clock destination.
CLK_DEST_SYS_UART0	Select SYS_UART0 as clock destination.
CLK_DEST_PERI_UART1	Select PERI_UART1 as clock destination.
CLK_DEST_SYS_UART1	Select SYS_UART1 as clock destination.
CLK_DEST_SYS_USBCTRL	Select SYS_USBCTRL as clock destination.
CLK_DEST_USB	Select USB as clock destination.
CLK_DEST_SYS_WATCHDOG	Select SYS_WATCHDOG as clock destination.
CLK_DEST_SYS_XIP	Select SYS_XIP as clock destination.
CLK_DEST_SYS_XOSC	Select SYS_XOSC as clock destination.

4.1.4.7. Function Documentation

4.1.4.7.1. check_sys_clock_hz

`bool check_sys_clock_hz (uint32_t freq_hz, uint * vco_freq_out, uint * post_div1_out, uint * post_div2_out)`

Check if a given system clock frequency is valid/attainable.

Parameters

`freq_hz`	Requested frequency
`vco_freq_out`	On success, the voltage controlled oscillator frequency 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.

4.1.4.7.2. 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_out`	On success, the voltage controlled oscillator frequency 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.

4.1.4.7.3. clock_configure

`bool clock_configure (clock_handle_t clock, uint32_t src, uint32_t auxsrc, uint32_t src_freq, uint32_t freq)`

Configure the specified clock with automatic clock divisor setup.

This method allows both the src_frequency of the input clock source AND the desired frequency to be specified, and will set the clock divider to achieve the exact or higher frequency achievable, with the maximum being the src_freq.

Note: The RP2350 clock hardware supports divisors from 1.0->65536.0 in steps of 1/65536

Note: The RP2040 clock hardware only supports divisors of exactly 1.0 or 2.0->16777216.0 in steps of 1/256

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

Parameters

`clock`	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

Returns

true if the clock is updated, false if freq > src_freq

4.1.4.7.4. clock_configure_gpin

`bool clock_configure_gpin (clock_handle_t clock, uint gpio, uint32_t src_freq, uint32_t freq)`

Configure a clock to come from a gpio input.

Parameters

`clock`	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

4.1.4.7.5. clock_configure_int_divider

`void clock_configure_int_divider (clock_handle_t clock, uint32_t src, uint32_t auxsrc, uint32_t src_freq, uint32_t int_divider)`

Configure the specified clock to use the undivided input source.

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

Parameters

`clock`	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
`int_divider`	an integer divider

4.1.4.7.6. clock_configure_undivided

`void clock_configure_undivided (clock_handle_t clock, uint32_t src, uint32_t auxsrc, uint32_t src_freq)`

Configure the specified clock to use the undivided input source.

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

Parameters

`clock`	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

4.1.4.7.7. clock_get_hz

`uint32_t clock_get_hz (clock_handle_t clock)`

Get the current frequency of the specified clock.

Parameters

`clock`	Clock

Returns

Clock frequency in Hz

4.1.4.7.8. clock_gpio_init

`static void clock_gpio_init (uint gpio, uint src, float div) [inline], [static]`

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, 25. 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 float amount to divide the source clock by. This is useful to not overwhelm the GPIO pin with a fast clock.

4.1.4.7.9. clock_gpio_init_int_frac16

`void clock_gpio_init_int_frac16 (uint gpio, uint src, uint32_t div_int, uint16_t div_frac16)`

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, 25. 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_int`	The integer part of the value to divide the source clock by. This is useful to not overwhelm the GPIO pin with a fast clock. This is in range of 1..2^24-1 on RP2040 and 1..2^16-1 on RP2350
`div_frac16`	The fractional part of the value to divide the source clock by. This is in range of 0..65535 (/65536).

4.1.4.7.10. clock_gpio_init_int_frac8

`static void clock_gpio_init_int_frac8 (uint gpio, uint src, uint32_t div_int, uint8_t div_frac8) [inline], [static]`

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, 25. 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_int`	The integer part of the value to divide the source clock by. This is useful to not overwhelm the GPIO pin with a fast clock. This is in range of 1..2^24-1 on RP2040 and 1..2^16-1 on RP2350
`div_frac8`	The fractional part of the value to divide the source clock by. This is in range of 0..255 (/256).

4.1.4.7.11. clock_set_reported_hz

`void clock_set_reported_hz (clock_handle_t clock, 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()

4.1.4.7.12. clock_stop

`void clock_stop (clock_handle_t clock)`

Stop the specified clock.

Parameters

`clock`	The clock to stop

4.1.4.7.13. 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.

4.1.4.7.14. 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.

4.1.4.7.15. set_sys_clock_48mhz

`void set_sys_clock_48mhz (void)`

Initialise the system clock to 48MHz.

Set the system clock to 48MHz, and set the peripheral clock to match.

4.1.4.7.16. set_sys_clock_hz

`static bool set_sys_clock_hz (uint32_t freq_hz, bool required) [inline], [static]`

Attempt to set a system clock frequency in hz.

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 set_sys_clock_pll

Parameters

`freq_hz`	Requested frequency
`required`	if true then this function will assert if the frequency is not attainable.

Returns

true if the clock was configured

4.1.4.7.17. set_sys_clock_khz

`static bool set_sys_clock_khz (uint32_t freq_khz, bool required) [inline], [static]`

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 set_sys_clock_pll

Parameters

`freq_khz`	Requested frequency
`required`	if true then this function will assert if the frequency is not attainable.

Returns

true if the clock was configured

4.1.4.7.18. set_sys_clock_pll

`void set_sys_clock_pll (uint32_t vco_freq, uint post_div1, uint post_div2)`

Initialise the system clock.

Parameters

`vco_freq`	The voltage controller oscillator frequency 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.

See the PLL documentation in the datasheet for details of driving the PLLs.

4.1.5. hardware_divider

RP2040 Low Low-level hardware-divider API. Non-RP2040 platforms provide software versions of all the functions.

4.1.5.1. Detailed Description

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 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.

On RP2350 there is no hardware divider, and the functions are implemented in software

See also

pico_divider

Example

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

int main() {
    stdio_init_all();
    printf("Hello, divider!\n");
```

```

    // This is the basic hardware divider function
    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 cycles.

    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;
}
```

4.1.5.2. Functions

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

Do a signed HW divide and wait for result.

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

Do an unsigned HW divide and wait for result.

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

Start a signed asynchronous divide.

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

Start an unsigned asynchronous divide.

`static void hw_divider_wait_ready (void)`

Wait for a divide to complete.

`static divmod_result_t hw_divider_result_nowait (void)`

Return result of HW divide, nowait.

`static divmod_result_t hw_divider_result_wait (void)`

Return result of last asynchronous HW divide.

`static uint32_t to_quotient_u32 (divmod_result_t r)`

Efficient extraction of unsigned quotient from 32p32 fixed point.

`static int32_t to_quotient_s32 (divmod_result_t r)`

Efficient extraction of signed quotient from 32p32 fixed point.

`static uint32_t to_remainder_u32 (divmod_result_t r)`

Efficient extraction of unsigned remainder from 32p32 fixed point.

`static int32_t to_remainder_s32 (divmod_result_t r)`

Efficient extraction of signed remainder from 32p32 fixed point.

`static uint32_t hw_divider_u32_quotient_wait (void)`

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

`static int32_t hw_divider_s32_quotient_wait (void)`

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

`static uint32_t hw_divider_u32_remainder_wait (void)`

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

`static int32_t hw_divider_s32_remainder_wait (void)`

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

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

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

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

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

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

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

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

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

`static void hw_divider_pause (void)`

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.

`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.

`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.

`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.

`static void hw_divider_save_state (hw_divider_state_t *dest)`

Save the calling cores hardware divider state.

`static void hw_divider_restore_state (hw_divider_state_t *src)`

Load a saved hardware divider state into the current core's hardware divider.

4.1.5.3. Function Documentation

4.1.5.3.1. hw_divider_divmod_s32

`static divmod_result_t hw_divider_divmod_s32 (int32_t a, int32_t b) [inline], [static]`

Do a signed HW divide and wait for result.

Divide `a` by `b`, wait for calculation to complete, return result as a pair of 32-bit quotient/remainder values.

Parameters

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

Returns

Results of divide as a pair of 32-bit quotient/remainder values.

4.1.5.3.2. hw_divider_divmod_s32_start

`static void hw_divider_divmod_s32_start (int32_t a, int32_t b) [inline], [static]`

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

4.1.5.3.3. hw_divider_divmod_u32

`static divmod_result_t hw_divider_divmod_u32 (uint32_t a, uint32_t b) [inline], [static]`

Do an unsigned HW divide and wait for result.

Divide `a` by `b`, wait for calculation to complete, return result as a pair of 32-bit quotient/remainder values.

Parameters

`a` The dividend

`b` The divisor

Returns

Results of divide as a pair of 32-bit quotient/remainder values.

4.1.5.3.4. hw_divider_divmod_u32_start

`static void hw_divider_divmod_u32_start (uint32_t a, uint32_t b) [inline], [static]`

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

4.1.5.3.5. hw_divider_pause

`static void hw_divider_pause (void) [inline], [static]`

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

4.1.5.3.6. hw_divider_quotient_s32

`static int32_t hw_divider_quotient_s32 (int32_t a, int32_t b) [inline], [static]`

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

4.1.5.3.7. hw_divider_remainder_s32

`static int32_t hw_divider_remainder_s32 (int32_t a, int32_t b) [inline], [static]`

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

4.1.5.3.8. hw_divider_restore_state

`static void hw_divider_restore_state (hw_divider_state_t * src) [inline], [static]`

Load a saved hardware divider state into the current core's hardware divider.

Copy the passed hardware divider state into the hardware divider.

Parameters

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

4.1.5.3.9. hw_divider_result_nowait

`static divmod_result_t hw_divider_result_nowait (void) [inline], [static]`

Return result of HW divide, nowait.

NOTE

> This is UNSAFE in that the calculation may not have been completed.

Returns

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

4.1.5.3.10. hw_divider_result_wait

`static divmod_result_t hw_divider_result_wait (void) [inline], [static]`

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.

4.1.5.3.11. hw_divider_s32_quotient_inlined

`static int32_t hw_divider_s32_quotient_inlined (int32_t a, int32_t b) [inline], [static]`

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

4.1.5.3.12. hw_divider_s32_quotient_wait

`static int32_t hw_divider_s32_quotient_wait (void) [inline], [static]`

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.

4.1.5.3.13. hw_divider_s32_remainder_inlined

`static int32_t hw_divider_s32_remainder_inlined (int32_t a, int32_t b) [inline], [static]`

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

4.1.5.3.14. hw_divider_s32_remainder_wait

`static int32_t hw_divider_s32_remainder_wait (void) [inline], [static]`

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.

4.1.5.3.15. hw_divider_save_state

`static void hw_divider_save_state (hw_divider_state_t * dest) [inline], [static]`

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

4.1.5.3.16. hw_divider_u32_quotient

`static uint32_t hw_divider_u32_quotient (uint32_t a, uint32_t b) [inline], [static]`

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

4.1.5.3.17. hw_divider_u32_quotient_inlined

`static uint32_t hw_divider_u32_quotient_inlined (uint32_t a, uint32_t b) [inline], [static]`

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

4.1.5.3.18. hw_divider_u32_quotient_wait

`static uint32_t hw_divider_u32_quotient_wait (void) [inline], [static]`

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.

4.1.5.3.19. hw_divider_u32_remainder

`static uint32_t hw_divider_u32_remainder (uint32_t a, uint32_t b) [inline], [static]`

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

4.1.5.3.20. hw_divider_u32_remainder_inlined

`static uint32_t hw_divider_u32_remainder_inlined (uint32_t a, uint32_t b) [inline], [static]`

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 divide

4.1.5.3.21. hw_divider_u32_remainder_wait

`static uint32_t hw_divider_u32_remainder_wait (void) [inline], [static]`

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.

4.1.5.3.22. hw_divider_wait_ready

`static void hw_divider_wait_ready (void) [inline], [static]`

Wait for a divide to complete.

Wait for a divide to complete

4.1.5.3.23. to_quotient_s32

`static int32_t to_quotient_s32 (divmod_result_t r) [inline], [static]`

Efficient extraction of signed quotient from 32p32 fixed point.

Parameters

`r` A pair of 32-bit quotient/remainder values.

Returns

Unsigned quotient

4.1.5.3.24. to_quotient_u32

`static uint32_t to_quotient_u32 (divmod_result_t r) [inline], [static]`

Efficient extraction of unsigned quotient from 32p32 fixed point.

Parameters

`r` A pair of 32-bit quotient/remainder values.

Returns

Unsigned quotient

4.1.5.3.25. to_remainder_s32

`static int32_t to_remainder_s32 (divmod_result_t r) [inline], [static]`

Efficient extraction of signed remainder from 32p32 fixed point.

Parameters

`r` A pair of 32-bit quotient/remainder values.

Returns

Signed remainder

> **NOTE**
>
> On arm this is just a 32 bit register move or a nop

4.1.5.3.26. to_remainder_u32

`static uint32_t to_remainder_u32 (divmod_result_t r) [inline], [static]`

Efficient extraction of unsigned remainder from 32p32 fixed point.

Parameters

`r` A pair of 32-bit quotient/remainder values.

Returns

Unsigned remainder

> **NOTE**
>
> On Arm this is just a 32 bit register move or a nop

4.1.6. hardware_dcp RP2350

Assembly macros for the Double Coprocessor.

4.1.7. hardware_dma

DMA Controller API.

4.1.7.1. Detailed Description

The RP-series microcontroller 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 RP-series microcontroller'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

4.1.7.2. Modules

channel_config

DMA channel configuration.

4.1.7.3. Macros

- `#define DMA_IRQ_NUM(irq_index)`

4.1.7.4. Typedefs

`typedef enum dreq_num_rp2350 dreq_num_t` RP2350

DREQ numbers for DMA pacing on RP2350 (used as typedef dreq_num_t)

`typedef enum dreq_num_rp2040 dreq_num_t` RP2040

DREQ numbers for DMA pacing on RP2040 (used as typedef dreq_num_t)

4.1.7.5. Enumerations

`enum dreq_num_rp2350 { DREQ_PIO0_TX0 = 0, DREQ_PIO0_TX1 = 1, DREQ_PIO0_TX2 = 2, DREQ_PIO0_TX3 = 3, DREQ_PIO0_RX0 = 4, DREQ_PIO0_RX1 = 5, DREQ_PIO0_RX2 = 6, DREQ_PIO0_RX3 = 7, DREQ_PIO1_TX0 = 8, DREQ_PIO1_TX1 = 9, DREQ_PIO1_TX2 = 10, DREQ_PIO1_TX3 = 11, DREQ_PIO1_RX0 = 12, DREQ_PIO1_RX1 = 13, DREQ_PIO1_RX2 = 14, DREQ_PIO1_RX3 = 15, DREQ_PIO2_TX0 = 16, DREQ_PIO2_TX1 = 17, DREQ_PIO2_TX2 = 18, DREQ_PIO2_TX3 = 19, DREQ_PIO2_RX0 = 20, DREQ_PIO2_RX1 = 21, DREQ_PIO2_RX2 = 22, DREQ_PIO2_RX3 = 23, DREQ_SPI0_TX = 24, DREQ_SPI0_RX = 25, DREQ_SPI1_TX = 26, DREQ_SPI1_RX = 27, DREQ_UART0_TX = 28, DREQ_UART0_RX = 29, DREQ_UART1_TX = 30, DREQ_UART1_RX = 31, DREQ_PWM_WRAP0 = 32, DREQ_PWM_WRAP1 = 33, DREQ_PWM_WRAP2 = 34, DREQ_PWM_WRAP3 = 35, DREQ_PWM_WRAP4 = 36, DREQ_PWM_WRAP5 = 37, DREQ_PWM_WRAP6 = 38, DREQ_PWM_WRAP7 = 39, DREQ_PWM_WRAP8 = 40, DREQ_PWM_WRAP9 = 41, DREQ_PWM_WRAP10 = 42, DREQ_PWM_WRAP11 = 43, DREQ_I2C0_TX = 44, DREQ_I2C0_RX = 45, DREQ_I2C1_TX = 46, DREQ_I2C1_RX = 47, DREQ_ADC = 48, DREQ_XIP_STREAM = 49, DREQ_XIP_QMITX = 50, DREQ_XIP_QMIRX = 51, DREQ_HSTX = 52, DREQ_CORESIGHT = 53, DREQ_SHA256 = 54, DREQ_DMA_TIMER0 = 59, DREQ_DMA_TIMER1 = 60, DREQ_DMA_TIMER2 = 61, DREQ_DMA_TIMER3 = 62, DREQ_FORCE = 63, DREQ_COUNT }` RP2350

DREQ numbers for DMA pacing on RP2350 (used as typedef dreq_num_t)

`enum dreq_num_rp2040 { DREQ_PIO0_TX0 = 0, DREQ_PIO0_TX1 = 1, DREQ_PIO0_TX2 = 2, DREQ_PIO0_TX3 = 3, DREQ_PIO0_RX0 = 4, DREQ_PIO0_RX1 = 5, DREQ_PIO0_RX2 = 6, DREQ_PIO0_RX3 = 7, DREQ_PIO1_TX0 = 8, DREQ_PIO1_TX1 = 9, DREQ_PIO1_TX2 = 10, DREQ_PIO1_TX3 = 11, DREQ_PIO1_RX0 = 12, DREQ_PIO1_RX1 = 13, DREQ_PIO1_RX2 = 14, DREQ_PIO1_RX3 = 15, DREQ_SPI0_TX = 16, DREQ_SPI0_RX = 17, DREQ_SPI1_TX = 18, DREQ_SPI1_RX = 19, DREQ_UART0_TX = 20, DREQ_UART0_RX = 21, DREQ_UART1_TX = 22, DREQ_UART1_RX = 23, DREQ_PWM_WRAP0 = 24, DREQ_PWM_WRAP1 = 25, DREQ_PWM_WRAP2 = 26, DREQ_PWM_WRAP3 = 27, DREQ_PWM_WRAP4 = 28, DREQ_PWM_WRAP5 = 29, DREQ_PWM_WRAP6 = 30, DREQ_PWM_WRAP7 = 31, DREQ_I2C0_TX = 32, DREQ_I2C0_RX = 33, DREQ_I2C1_TX = 34, DREQ_I2C1_RX = 35, DREQ_ADC = 36, DREQ_XIP_STREAM = 37, DREQ_XIP_SSITX = 38, DREQ_XIP_SSIRX = 39, DREQ_DMA_TIMER0 = 59, DREQ_DMA_TIMER1 = 60, DREQ_DMA_TIMER2 = 61, DREQ_DMA_TIMER3 = 62, DREQ_FORCE = 63, DREQ_COUNT }` RP2040

DREQ numbers for DMA pacing on RP2040 (used as typedef dreq_num_t)

`enum dma_channel_transfer_size { DMA_SIZE_8 = 0, DMA_SIZE_16 = 1, DMA_SIZE_32 = 2 }`

Enumeration of available DMA channel transfer sizes.

4.1.7.6. Functions

`void dma_channel_claim (uint channel)`

Mark a dma channel as used.

`void dma_claim_mask (uint32_t channel_mask)`

Mark multiple dma channels as used.

`void dma_channel_unclaim (uint channel)`

Mark a dma channel as no longer used.

`void dma_unclaim_mask (uint32_t channel_mask)`

Mark multiple dma channels as no longer used.

`int dma_claim_unused_channel (bool required)`

Claim a free dma channel.

`bool dma_channel_is_claimed (uint channel)`

Determine if a dma channel is claimed.

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

Set a channel configuration.

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

Set the DMA initial read address.

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

Set the DMA initial write address.

`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.

`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.

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

Start a DMA transfer from a buffer immediately.

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

Start a DMA transfer to a buffer immediately.

`static void dma_start_channel_mask (uint32_t chan_mask)`

Start one or more channels simultaneously.

`static void dma_channel_start (uint channel)`

Start a single DMA channel.

`static void dma_channel_abort (uint channel)`

Stop a DMA transfer.

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

Enable single DMA channel’s interrupt via DMA_IRQ_0.

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

Enable multiple DMA channels' interrupts via DMA_IRQ_0.

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

Enable single DMA channel’s interrupt via DMA_IRQ_1.

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

Enable multiple DMA channels' interrupts via DMA_IRQ_1.

`static void dma_irqn_set_channel_enabled (uint irq_index, uint channel, bool enabled)`

Enable single DMA channel interrupt on either DMA_IRQ_0 or DMA_IRQ_1.

`static void dma_irqn_set_channel_mask_enabled (uint irq_index, uint32_t channel_mask, bool enabled)`

Enable multiple DMA channels' interrupt via either DMA_IRQ_0 or DMA_IRQ_1.

`static bool dma_channel_get_irq0_status (uint channel)`

Determine if a particular channel is a cause of DMA_IRQ_0.

`static bool dma_channel_get_irq1_status (uint channel)`

Determine if a particular channel is a cause of DMA_IRQ_1.

`static bool dma_irqn_get_channel_status (uint irq_index, uint channel)`

Determine if a particular channel is a cause of DMA_IRQ_N.

`static void dma_channel_acknowledge_irq0 (uint channel)`

Acknowledge a channel IRQ, resetting it as the cause of DMA_IRQ_0.

`static void dma_channel_acknowledge_irq1 (uint channel)`

Acknowledge a channel IRQ, resetting it as the cause of DMA_IRQ_1.

`static void dma_irqn_acknowledge_channel (uint irq_index, uint channel)`

Acknowledge a channel IRQ, resetting it as the cause of DMA_IRQ_N.

`static bool dma_channel_is_busy (uint channel)`

Check if DMA channel is busy.

`static void dma_channel_wait_for_finish_blocking (uint channel)`

Wait for a DMA channel transfer to complete.

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

Enable the DMA sniffing targeting the specified channel.

`static void dma_sniffer_set_byte_swap_enabled (bool swap)`

Enable the Sniffer byte swap function.

`static void dma_sniffer_set_output_invert_enabled (bool invert)`

Enable the Sniffer output invert function.

`static void dma_sniffer_set_output_reverse_enabled (bool reverse)`

Enable the Sniffer output bit reversal function.

`static void dma_sniffer_disable (void)`

Disable the DMA sniffer.

`static void dma_sniffer_set_data_accumulator (uint32_t seed_value)`

Set the sniffer’s data accumulator with initial value.

`static uint32_t dma_sniffer_get_data_accumulator (void)`

Get the sniffer’s data accumulator value.

`void dma_timer_claim (uint timer)`

Mark a dma timer as used.

`void dma_timer_unclaim (uint timer)`

Mark a dma timer as no longer used.

`int dma_claim_unused_timer (bool required)`

Claim a free dma timer.

`bool dma_timer_is_claimed (uint timer)`

Determine if a dma timer is claimed.

`static void dma_timer_set_fraction (uint timer, uint16_t numerator, uint16_t denominator)`

Set the multiplier for the given DMA timer.

`static uint dma_get_timer_dreq (uint timer_num)`

Return the DREQ number for a given DMA timer.

`static int dma_get_irq_num (uint irq_index)`

Return DMA_IRQ_<irqn>

`void dma_channel_cleanup (uint channel)`

Performs DMA channel cleanup after use.

4.1.7.7. Macro Definition Documentation

4.1.7.7.1. DMA_IRQ_NUM

```
#define DMA_IRQ_NUM(irq_index)
```

Returns the irq_num_t for the nth DMA interrupt.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.7.8. Typedef Documentation

4.1.7.8.1. dreq_num_t RP2350

```
typedef enum dreq_num_rp2350 dreq_num_t
```

DREQ numbers for DMA pacing on RP2350 (used as typedef dreq_num_t)

4.1.7.8.2. dreq_num_t RP2040

```
typedef enum dreq_num_rp2040 dreq_num_t
```

DREQ numbers for DMA pacing on RP2040 (used as typedef dreq_num_t)

4.1.7.9. Enumeration Type Documentation

4.1.7.9.1. dreq_num_rp2350 RP2350

```
enum dreq_num_rp2350
```

DREQ numbers for DMA pacing on RP2350 (used as typedef dreq_num_t)

Table 13. Enumerator

DREQ_PIO0_TX0	Select PIO0's TX FIFO 0 as DREQ.
DREQ_PIO0_TX1	Select PIO0's TX FIFO 1 as DREQ.
DREQ_PIO0_TX2	Select PIO0's TX FIFO 2 as DREQ.
DREQ_PIO0_TX3	Select PIO0's TX FIFO 3 as DREQ.
DREQ_PIO0_RX0	Select PIO0's RX FIFO 0 as DREQ.
DREQ_PIO0_RX1	Select PIO0's RX FIFO 1 as DREQ.
DREQ_PIO0_RX2	Select PIO0's RX FIFO 2 as DREQ.
DREQ_PIO0_RX3	Select PIO0's RX FIFO 3 as DREQ.

DREQ_PIO1_TX0	Select PIO1's TX FIFO 0 as DREQ.
DREQ_PIO1_TX1	Select PIO1's TX FIFO 1 as DREQ.
DREQ_PIO1_TX2	Select PIO1's TX FIFO 2 as DREQ.
DREQ_PIO1_TX3	Select PIO1's TX FIFO 3 as DREQ.
DREQ_PIO1_RX0	Select PIO1's RX FIFO 0 as DREQ.
DREQ_PIO1_RX1	Select PIO1's RX FIFO 1 as DREQ.
DREQ_PIO1_RX2	Select PIO1's RX FIFO 2 as DREQ.
DREQ_PIO1_RX3	Select PIO1's RX FIFO 3 as DREQ.
DREQ_PIO2_TX0	Select PIO2's TX FIFO 0 as DREQ.
DREQ_PIO2_TX1	Select PIO2's TX FIFO 1 as DREQ.
DREQ_PIO2_TX2	Select PIO2's TX FIFO 2 as DREQ.
DREQ_PIO2_TX3	Select PIO2's TX FIFO 3 as DREQ.
DREQ_PIO2_RX0	Select PIO2's RX FIFO 0 as DREQ.
DREQ_PIO2_RX1	Select PIO2's RX FIFO 1 as DREQ.
DREQ_PIO2_RX2	Select PIO2's RX FIFO 2 as DREQ.
DREQ_PIO2_RX3	Select PIO2's RX FIFO 3 as DREQ.
DREQ_SPI0_TX	Select SPI0's TX FIFO as DREQ.
DREQ_SPI0_RX	Select SPI0's RX FIFO as DREQ.
DREQ_SPI1_TX	Select SPI1's TX FIFO as DREQ.
DREQ_SPI1_RX	Select SPI1's RX FIFO as DREQ.
DREQ_UART0_TX	Select UART0's TX FIFO as DREQ.
DREQ_UART0_RX	Select UART0's RX FIFO as DREQ.
DREQ_UART1_TX	Select UART1's TX FIFO as DREQ.
DREQ_UART1_RX	Select UART1's RX FIFO as DREQ.
DREQ_PWM_WRAP0	Select PWM Counter 0's Wrap Value as DREQ.
DREQ_PWM_WRAP1	Select PWM Counter 1's Wrap Value as DREQ.
DREQ_PWM_WRAP2	Select PWM Counter 2's Wrap Value as DREQ.
DREQ_PWM_WRAP3	Select PWM Counter 3's Wrap Value as DREQ.
DREQ_PWM_WRAP4	Select PWM Counter 4's Wrap Value as DREQ.
DREQ_PWM_WRAP5	Select PWM Counter 5's Wrap Value as DREQ.
DREQ_PWM_WRAP6	Select PWM Counter 6's Wrap Value as DREQ.
DREQ_PWM_WRAP7	Select PWM Counter 7's Wrap Value as DREQ.
DREQ_PWM_WRAP8	Select PWM Counter 8's Wrap Value as DREQ.
DREQ_PWM_WRAP9	Select PWM Counter 9's Wrap Value as DREQ.
DREQ_PWM_WRAP10	Select PWM Counter 0's Wrap Value as DREQ.
DREQ_PWM_WRAP11	Select PWM Counter 1's Wrap Value as DREQ.
DREQ_I2C0_TX	Select I2C0's TX FIFO as DREQ.

DREQ_I2C0_RX	Select I2C0's RX FIFO as DREQ.
DREQ_I2C1_TX	Select I2C1's TX FIFO as DREQ.
DREQ_I2C1_RX	Select I2C1's RX FIFO as DREQ.
DREQ_ADC	Select the ADC as DREQ.
DREQ_XIP_STREAM	Select the XIP Streaming FIFO as DREQ.
DREQ_XIP_QMITX	Select XIP_QMITX as DREQ.
DREQ_XIP_QMIRX	Select XIP_QMIRX as DREQ.
DREQ_HSTX	Select HSTX as DREQ.
DREQ_CORESIGHT	Select CORESIGHT as DREQ.
DREQ_SHA256	Select SHA256 as DREQ.
DREQ_DMA_TIMER0	Select DMA_TIMER0 as DREQ.
DREQ_DMA_TIMER1	Select DMA_TIMER0 as DREQ.
DREQ_DMA_TIMER2	Select DMA_TIMER1 as DREQ.
DREQ_DMA_TIMER3	Select DMA_TIMER3 as DREQ.
DREQ_FORCE	Select FORCE as DREQ.

4.1.7.9.2. dreq_num_rp2040 RP2040

`enum dreq_num_rp2040`

DREQ numbers for DMA pacing on RP2040 (used as typedef dreq_num_t)

Table 14. Enumerator

DREQ_PIO0_TX0	Select PIO0's TX FIFO 0 as DREQ.
DREQ_PIO0_TX1	Select PIO0's TX FIFO 1 as DREQ.
DREQ_PIO0_TX2	Select PIO0's TX FIFO 2 as DREQ.
DREQ_PIO0_TX3	Select PIO0's TX FIFO 3 as DREQ.
DREQ_PIO0_RX0	Select PIO0's RX FIFO 0 as DREQ.
DREQ_PIO0_RX1	Select PIO0's RX FIFO 1 as DREQ.
DREQ_PIO0_RX2	Select PIO0's RX FIFO 2 as DREQ.
DREQ_PIO0_RX3	Select PIO0's RX FIFO 3 as DREQ.
DREQ_PIO1_TX0	Select PIO1's TX FIFO 0 as DREQ.
DREQ_PIO1_TX1	Select PIO1's TX FIFO 1 as DREQ.
DREQ_PIO1_TX2	Select PIO1's TX FIFO 2 as DREQ.
DREQ_PIO1_TX3	Select PIO1's TX FIFO 3 as DREQ.
DREQ_PIO1_RX0	Select PIO1's RX FIFO 0 as DREQ.
DREQ_PIO1_RX1	Select PIO1's RX FIFO 1 as DREQ.
DREQ_PIO1_RX2	Select PIO1's RX FIFO 2 as DREQ.
DREQ_PIO1_RX3	Select PIO1's RX FIFO 3 as DREQ.
DREQ_SPI0_TX	Select SPI0's TX FIFO as DREQ.

DREQ_SPI0_RX	Select SPI0's RX FIFO as DREQ.
DREQ_SPI1_TX	Select SPI1's TX FIFO as DREQ.
DREQ_SPI1_RX	Select SPI1's RX FIFO as DREQ.
DREQ_UART0_TX	Select UART0's TX FIFO as DREQ.
DREQ_UART0_RX	Select UART0's RX FIFO as DREQ.
DREQ_UART1_TX	Select UART1's TX FIFO as DREQ.
DREQ_UART1_RX	Select UART1's RX FIFO as DREQ.
DREQ_PWM_WRAP0	Select PWM Counter 0's Wrap Value as DREQ.
DREQ_PWM_WRAP1	Select PWM Counter 1's Wrap Value as DREQ.
DREQ_PWM_WRAP2	Select PWM Counter 2's Wrap Value as DREQ.
DREQ_PWM_WRAP3	Select PWM Counter 3's Wrap Value as DREQ.
DREQ_PWM_WRAP4	Select PWM Counter 4's Wrap Value as DREQ.
DREQ_PWM_WRAP5	Select PWM Counter 5's Wrap Value as DREQ.
DREQ_PWM_WRAP6	Select PWM Counter 6's Wrap Value as DREQ.
DREQ_PWM_WRAP7	Select PWM Counter 7's Wrap Value as DREQ.
DREQ_I2C0_TX	Select I2C0's TX FIFO as DREQ.
DREQ_I2C0_RX	Select I2C0's RX FIFO as DREQ.
DREQ_I2C1_TX	Select I2C1's TX FIFO as DREQ.
DREQ_I2C1_RX	Select I2C1's RX FIFO as DREQ.
DREQ_ADC	Select the ADC as DREQ.
DREQ_XIP_STREAM	Select the XIP Streaming FIFO as DREQ.
DREQ_XIP_SSITX	Select the XIP SSI TX FIFO as DREQ.
DREQ_XIP_SSIRX	Select the XIP SSI RX FIFO as DREQ.
DREQ_DMA_TIMER0	Select DMA_TIMER0 as DREQ.
DREQ_DMA_TIMER1	Select DMA_TIMER0 as DREQ.
DREQ_DMA_TIMER2	Select DMA_TIMER1 as DREQ.
DREQ_DMA_TIMER3	Select DMA_TIMER3 as DREQ.
DREQ_FORCE	Select FORCE as DREQ.

4.1.7.9.3. dma_channel_transfer_size

`enum dma_channel_transfer_size`

Enumeration of available DMA channel transfer sizes.

Names indicate the number of bits.

Table 15. Enumerator

DMA_SIZE_8	Byte transfer (8 bits)
DMA_SIZE_16	Half word transfer (16 bits)
DMA_SIZE_32	Word transfer (32 bits)

4.1.7.10. Function Documentation

4.1.7.10.1. dma_channel_abort

`static void dma_channel_abort (uint channel) [inline], [static]`

Stop a DMA transfer.

Function will only return once the DMA has stopped.

RP2040 only: Note that due to errata RP2040-E13, aborting a channel which has transfers in-flight (i.e. an individual read has taken place but the corresponding write has not), the ABORT status bit will clear prematurely, and subsequently the in-flight transfers will trigger a completion interrupt once they complete.

The effect of this is that you *may* see a spurious completion interrupt on the channel as a result of calling this method.

The calling code should be sure to ignore a completion IRQ as a result of this method. This may not require any additional work, as aborting a channel which may be about to complete, when you have a completion IRQ handler registered, is inherently race-prone, and so code is likely needed to disambiguate the two occurrences.

If that is not the case, but you do have a channel completion IRQ handler registered, you can simply disable/re-enable the IRQ around the call to this method as shown by this code fragment (using DMA IRQ0).

```
// disable the channel on IRQ0
dma_channel_set_irq0_enabled(channel, false);
// abort the channel
dma_channel_abort(channel);
// clear the spurious IRQ (if there was one)
dma_channel_acknowledge_irq0(channel);
// re-enable the channel on IRQ0
dma_channel_set_irq0_enabled(channel, true);
```

RP2350 only: Due to errata RP2350-E5 (see the RP2350 datasheet for further detail), it is necessary to clear the enable bit of the aborted channel and any chained channels prior to the abort to prevent re-triggering.

Parameters

`channel`	DMA channel

4.1.7.10.2. dma_channel_acknowledge_irq0

`static void dma_channel_acknowledge_irq0 (uint channel) [inline], [static]`

Acknowledge a channel IRQ, resetting it as the cause of DMA_IRQ_0.

Parameters

`channel`	DMA channel

4.1.7.10.3. dma_channel_acknowledge_irq1

`static void dma_channel_acknowledge_irq1 (uint channel) [inline], [static]`

Acknowledge a channel IRQ, resetting it as the cause of DMA_IRQ_1.

Parameters

`channel`	DMA channel

4.1.7.10.4. 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

4.1.7.10.5. dma_channel_cleanup

`void dma_channel_cleanup (uint channel)`

Performs DMA channel cleanup after use.

This can be used to cleanup dma channels when they're no longer needed, such that they are in a clean state for reuse. IRQ's for the channel are disabled, any in flight-transfer is aborted and any outstanding interrupts are cleared. The channel is then clear to be reused for other purposes.

```
if (dma_channel >= 0) {
    dma_channel_cleanup(dma_channel);
    dma_channel_unclaim(dma_channel);
    dma_channel = -1;
}
```

Parameters

`channel`	DMA channel

4.1.7.10.6. 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) [inline], [static]`

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

4.1.7.10.7. dma_channel_get_irq0_status

`static bool dma_channel_get_irq0_status (uint channel) [inline], [static]`

Determine if a particular channel is a cause of DMA_IRQ_0.

Parameters

`channel`	DMA channel

Returns

true if the channel is a cause of DMA_IRQ_0, false otherwise

4.1.7.10.8. dma_channel_get_irq1_status

`static bool dma_channel_get_irq1_status (uint channel) [inline], [static]`

Determine if a particular channel is a cause of DMA_IRQ_1.

Parameters

`channel`	DMA channel

Returns

true if the channel is a cause of DMA_IRQ_1, false otherwise

4.1.7.10.9. dma_channel_is_busy

`static bool dma_channel_is_busy (uint channel) [inline], [static]`

Check if DMA channel is busy.

Parameters

`channel`	DMA channel

Returns

true if the channel is currently busy

4.1.7.10.10. dma_channel_is_claimed

`bool dma_channel_is_claimed (uint channel)`

Determine if a dma channel is claimed.

Parameters

`channel`	the dma channel

Returns

true if the channel is claimed, false otherwise

See also

dma_channel_claim

dma_channel_claim_mask

4.1.7.10.11. dma_channel_set_config

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

Set a channel configuration.

Parameters

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

4.1.7.10.12. dma_channel_set_irq0_enabled

`static void dma_channel_set_irq0_enabled (uint channel, bool enabled) [inline], [static]`

Enable single DMA channel's interrupt via DMA_IRQ_0.

Parameters

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

4.1.7.10.13. dma_channel_set_irq1_enabled

`static void dma_channel_set_irq1_enabled (uint channel, bool enabled) [inline], [static]`

Enable single DMA channel's interrupt via DMA_IRQ_1.

Parameters

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

4.1.7.10.14. dma_channel_set_read_addr

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

Set the DMA initial read address.

Parameters

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

4.1.7.10.15. dma_channel_set_trans_count

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

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

4.1.7.10.16. dma_channel_set_write_addr

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

Set the DMA initial write address.

Parameters

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

4.1.7.10.17. dma_channel_start

`static void dma_channel_start (uint channel) [inline], [static]`

Start a single DMA channel.

Parameters

`channel`	DMA channel

4.1.7.10.18. dma_channel_transfer_from_buffer_now

`static void dma_channel_transfer_from_buffer_now (uint channel, const volatile void * read_addr, uint32_t transfer_count) [inline], [static]`

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.

4.1.7.10.19. dma_channel_transfer_to_buffer_now

`static void dma_channel_transfer_to_buffer_now (uint channel, volatile void * write_addr, uint32_t transfer_count) [inline], [static]`

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.

4.1.7.10.20. dma_channel_unclaim

`void dma_channel_unclaim (uint channel)`

Mark a dma channel as no longer used.

Parameters

`channel` the dma channel to release

4.1.7.10.21. dma_channel_wait_for_finish_blocking

`static void dma_channel_wait_for_finish_blocking (uint channel) [inline], [static]`

Wait for a DMA channel transfer to complete.

Parameters

`channel` DMA channel

4.1.7.10.22. 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)

4.1.7.10.23. 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

4.1.7.10.24. dma_claim_unused_timer

`int dma_claim_unused_timer (bool required)`

Claim a free dma timer.

Parameters

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

Returns

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

4.1.7.10.25. dma_get_irq_num

`static int dma_get_irq_num (uint irq_index) [inline], [static]`

Return DMA_IRQ_<irqn>

Parameters

`irq_index` 0 the DMA irq index

Returns

The irq_num_t to use for DMA

4.1.7.10.26. dma_get_timer_dreq

`static uint dma_get_timer_dreq (uint timer_num) [inline], [static]`

Return the DREQ number for a given DMA timer.

Parameters

`timer_num`	DMA timer number 0-3

4.1.7.10.27. dma_irqn_acknowledge_channel

`static void dma_irqn_acknowledge_channel (uint irq_index, uint channel) [inline], [static]`

Acknowledge a channel IRQ, resetting it as the cause of DMA_IRQ_N.

Parameters

`irq_index`	the IRQ index; either 0 or 1 for DMA_IRQ_0 or DMA_IRQ_1
`channel`	DMA channel

4.1.7.10.28. dma_irqn_get_channel_status

`static bool dma_irqn_get_channel_status (uint irq_index, uint channel) [inline], [static]`

Determine if a particular channel is a cause of DMA_IRQ_N.

Parameters

`irq_index`	the IRQ index; either 0 or 1 for DMA_IRQ_0 or DMA_IRQ_1
`channel`	DMA channel

Returns

true if the channel is a cause of the DMA_IRQ_N, false otherwise

4.1.7.10.29. dma_irqn_set_channel_enabled

`static void dma_irqn_set_channel_enabled (uint irq_index, uint channel, bool enabled) [inline], [static]`

Enable single DMA channel interrupt on either DMA_IRQ_0 or DMA_IRQ_1.

Parameters

`irq_index`	the IRQ index; either 0 or 1 for DMA_IRQ_0 or DMA_IRQ_1
`channel`	DMA channel
`enabled`	true to enable interrupt via irq_index for specified channel, false to disable.

4.1.7.10.30. dma_irqn_set_channel_mask_enabled

`static void dma_irqn_set_channel_mask_enabled (uint irq_index, uint32_t channel_mask, bool enabled) [inline], [static]`

Enable multiple DMA channels' interrupt via either DMA_IRQ_0 or DMA_IRQ_1.

Parameters

`irq_index`	the IRQ index; either 0 or 1 for DMA_IRQ_0 or DMA_IRQ_1
`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.

4.1.7.10.31. dma_set_irq0_channel_mask_enabled

`static void dma_set_irq0_channel_mask_enabled (uint32_t channel_mask, bool enabled) [inline], [static]`

Enable multiple DMA channels' interrupts via DMA_IRQ_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.

4.1.7.10.32. dma_set_irq1_channel_mask_enabled

`static void dma_set_irq1_channel_mask_enabled (uint32_t channel_mask, bool enabled) [inline], [static]`

Enable multiple DMA channels' interrupts via DMA_IRQ_1.

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.

4.1.7.10.33. dma_sniffer_disable

`static void dma_sniffer_disable (void) [inline], [static]`

Disable the DMA sniffer.

4.1.7.10.34. dma_sniffer_enable

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

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).

4.1.7.10.35. dma_sniffer_get_data_accumulator

`static uint32_t dma_sniffer_get_data_accumulator (void) [inline], [static]`

Get the sniffer's data accumulator value.

Read value calculated by the hardware from sniffing the DMA stream

4.1.7.10.36. dma_sniffer_set_byte_swap_enabled

`static void dma_sniffer_set_byte_swap_enabled (bool swap) [inline], [static]`

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

4.1.7.10.37. dma_sniffer_set_data_accumulator

`static void dma_sniffer_set_data_accumulator (uint32_t seed_value) [inline], [static]`

Set the sniffer's data accumulator with initial value.

Generally, CRC algorithms are used with the data accumulator initially seeded with 0xFFFF or 0xFFFFFFFF (for crc16 and crc32 algorithms)

Parameters

`seed_value`	value to set data accumulator

4.1.7.10.38. dma_sniffer_set_output_invert_enabled

`static void dma_sniffer_set_output_invert_enabled (bool invert) [inline], [static]`

Enable the Sniffer output invert function.

If enabled, the sniff data result appears bit-inverted when read. This does not affect the way the checksum is calculated.

Parameters

`invert`	Set true to enable output bit inversion

4.1.7.10.39. dma_sniffer_set_output_reverse_enabled

`static void dma_sniffer_set_output_reverse_enabled (bool reverse) [inline], [static]`

Enable the Sniffer output bit reversal function.

If enabled, the sniff data result appears bit-reversed when read. This does not affect the way the checksum is calculated.

Parameters

`reverse`	Set true to enable output bit reversal

4.1.7.10.40. dma_start_channel_mask

`static void dma_start_channel_mask (uint32_t chan_mask) [inline], [static]`

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.

4.1.7.10.41. dma_timer_claim

`void dma_timer_claim (uint timer)`

Mark a dma timer as used.

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

Parameters

`timer`	the dma timer

4.1.7.10.42. dma_timer_is_claimed

`bool dma_timer_is_claimed (uint timer)`

Determine if a dma timer is claimed.

Parameters

`timer`	the dma timer

Returns

true if the timer is claimed, false otherwise

See also

dma_timer_claim

4.1.7.10.43. dma_timer_set_fraction

`static void dma_timer_set_fraction (uint timer, uint16_t numerator, uint16_t denominator) [inline], [static]`

Set the multiplier for the given DMA timer.

The timer will run at the system_clock_freq * numerator / denominator, so this is the speed that data elements will be transferred at via a DMA channel using this timer as a DREQ. The multiplier must be less than or equal to one.

Parameters

`timer`	the dma timer
`numerator`	the fraction's numerator
`denominator`	the fraction's denominator

4.1.7.10.44. dma_timer_unclaim

`void dma_timer_unclaim (uint timer)`

Mark a dma timer as no longer used.

Method for cooperative claiming of hardware.

Parameters

`timer`	the dma timer to release

4.1.7.10.45. dma_unclaim_mask

`void dma_unclaim_mask (uint32_t channel_mask)`

Mark multiple dma channels as no longer used.

Parameters

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

4.1.7.11. channel_config

DMA channel configuration.

4.1.7.11.1. Detailed Description

A DMA channel needs to be configured, these functions provide handy helpers to set up configuration structures. See dma_channel_config

4.1.7.11.2. Functions

`static void channel_config_set_read_increment (dma_channel_config *c, bool incr)`

Set DMA channel read increment in a channel configuration object.

`static void channel_config_set_write_increment (dma_channel_config *c, bool incr)`

Set DMA channel write increment in a channel configuration object.

`static void channel_config_set_dreq (dma_channel_config *c, uint dreq)`

Select a transfer request signal in a channel configuration object.

`static void channel_config_set_chain_to (dma_channel_config *c, uint chain_to)`

Set DMA channel chain_to channel in a channel configuration object.

`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 in a channel configuration object.

`static void channel_config_set_ring (dma_channel_config *c, bool write, uint size_bits)`

Set address wrapping parameters in a channel configuration object.

`static void channel_config_set_bswap (dma_channel_config *c, bool bswap)`

Set DMA byte swapping config in a channel configuration object.

`static void channel_config_set_irq_quiet (dma_channel_config *c, bool irq_quiet)`

Set IRQ quiet mode in a channel configuration object.

`static void channel_config_set_high_priority (dma_channel_config *c, bool high_priority)`

Set the channel priority in a channel configuration object.

`static void channel_config_set_enable (dma_channel_config *c, bool enable)`

Enable/Disable the DMA channel in a channel configuration object.

`static void channel_config_set_sniff_enable (dma_channel_config *c, bool sniff_enable)`

Enable access to channel by sniff hardware in a channel configuration object.

`static dma_channel_config dma_channel_get_default_config (uint channel)`

Get the default channel configuration for a given channel.

`static dma_channel_config dma_get_channel_config (uint channel)`

Get the current configuration for the specified channel.

`static uint32_t channel_config_get_ctrl_value (const dma_channel_config *config)`

Get the raw configuration register from a channel configuration.

4.1.7.11.3. Function Documentation

channel_config_get_ctrl_value

`static uint32_t channel_config_get_ctrl_value (const dma_channel_config * config) [inline], [static]`

Get the raw configuration register from a channel configuration.

Parameters

`config`	Pointer to a config structure.

Returns

Register content

channel_config_set_bswap

`static void channel_config_set_bswap (dma_channel_config * c, bool bswap) [inline], [static]`

Set DMA byte swapping config in a channel configuration object.

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 object
`bswap`	True to enable byte swapping

channel_config_set_chain_to

`static void channel_config_set_chain_to (dma_channel_config * c, uint chain_to) [inline], [static]`

Set DMA channel chain_to channel in a channel configuration object.

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 object
`chain_to`	Channel to trigger when this channel completes.

channel_config_set_dreq

`static void channel_config_set_dreq (dma_channel_config * c, uint dreq) [inline], [static]`

Select a transfer request signal in a channel configuration object.

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)

channel_config_set_enable

`static void channel_config_set_enable (dma_channel_config * c, bool enable) [inline], [static]`

Enable/Disable the DMA channel in a channel configuration object.

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 object
`enable`	True to enable the DMA channel. When enabled, the channel will respond to triggering events, and start transferring data.

channel_config_set_high_priority

`static void channel_config_set_high_priority (dma_channel_config * c, bool high_priority) [inline], [static]`

Set the channel priority in a channel configuration object.

When true, gives a channel preferential treatment in issue scheduling: in each scheduling round, all high priority channels are considered first, and then only a single low priority channel, before returning to the high priority channels.

This only affects the order in which the DMA schedules channels. The DMA's bus priority is not changed. If the DMA is not saturated then a low priority channel will see no loss of throughput.

Parameters

`c`	Pointer to channel configuration object
`high_priority`	True to enable high priority

channel_config_set_irq_quiet

`static void channel_config_set_irq_quiet (dma_channel_config * c, bool irq_quiet) [inline], [static]`

Set IRQ quiet mode in a channel configuration object.

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 object
`irq_quiet`	True to enable quiet mode, false to disable.

channel_config_set_read_increment

`static void channel_config_set_read_increment (dma_channel_config * c, bool incr) [inline], [static]`

Set DMA channel read increment in a channel configuration object.

Parameters

`c`	Pointer to channel configuration object
`incr`	True to enable read address increments, if false, each read will be from the same address Usually disabled for peripheral to memory transfers

channel_config_set_ring

`static void channel_config_set_ring (dma_channel_config * c, bool write, uint size_bits) [inline], [static]`

Set address wrapping parameters in a channel configuration object.

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 object
`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.

channel_config_set_sniff_enable

`static void channel_config_set_sniff_enable (dma_channel_config * c, bool sniff_enable) [inline], [static]`

Enable access to channel by sniff hardware in a channel configuration object.

Sniff HW must be enabled and have this channel selected.

Parameters

`c`	Pointer to channel configuration object
`sniff_enable`	True to enable the Sniff HW access to this DMA channel.

channel_config_set_transfer_data_size

`static void channel_config_set_transfer_data_size (dma_channel_config * c, enum dma_channel_transfer_size size) [inline], [static]`

Set the size of each DMA bus transfer in a channel configuration object.

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 object
`size`	See enum for possible values.

channel_config_set_write_increment

`static void channel_config_set_write_increment (dma_channel_config * c, bool incr) [inline], [static]`

Set DMA channel write increment in a channel configuration object.

Parameters

`c`	Pointer to channel configuration object

`incr` True to enable write address increments, if false, each write will be to the same address Usually disabled for memory to peripheral transfers

dma_channel_get_default_config

`static dma_channel_config dma_channel_get_default_config (uint channel) [inline], [static]`

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
Quiet IRQs	false
High Priority	false
Channel Enable	true
Sniff Enable	false

Parameters

`channel` DMA channel

Returns

the default configuration which can then be modified.

dma_get_channel_config

`static dma_channel_config dma_get_channel_config (uint channel) [inline], [static]`

Get the current configuration for the specified channel.

Parameters

`channel` DMA channel

Returns

The current configuration as read from the HW register (not cached)

4.1.8. hardware_exception

Methods for setting processor exception handlers.

4.1.8.1. Detailed Description

Exceptions are identified by a exception_number which is a number from -15 to -1; these are the numbers relative to the index of the first IRQ vector in the vector table. (i.e. vector table index is exception_num plus 16)

There is one set of exception handlers per core, so the exception handlers for each core as set by these methods are independent.

NOTE

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

4.1.8.2. Typedefs

`typedef void(* exception_handler_t)(void)`

Exception handler function type.

4.1.8.3. Enumerations

`enum exception_number { MIN_EXCEPTION_NUM = 2, NMI_EXCEPTION = 2, HARDFAULT_EXCEPTION = 3, MEMMANAGE_EXCEPTION = 4, BUSFAULT_EXCEPTION = 5, USAGEFAULT_EXCEPTION = 6, SECUREFAULT_EXCEPTION = 7, SVCALL_EXCEPTION = 11, PENDSV_EXCEPTION = 14, SYSTICK_EXCEPTION = 15, MAX_EXCEPTION_NUM = 15 }`

Exception number definitions.

4.1.8.4. Functions

`exception_handler_t exception_set_exclusive_handler (enum exception_number num, exception_handler_t handler)`

Set the exception handler for an exception on the executing core.

`void exception_restore_handler (enum exception_number num, exception_handler_t original_handler)`

Restore the original exception handler for an exception on this core.

`exception_handler_t exception_get_vtable_handler (enum exception_number num)`

Get the current exception handler for the specified exception from the currently installed vector table of the execution core.

`bool exception_set_priority (uint num, uint8_t hardware_priority)`

Set specified exception's priority.

`uint exception_get_priority (uint num)`

Get specified exception's priority.

4.1.8.5. Typedef Documentation

4.1.8.5.1. exception_handler_t

```
typedef void(* exception_handler_t) (void)
```

Exception handler function type.

All exception handlers should be of this type, and follow normal ARM EABI register saving conventions

4.1.8.6. Enumeration Type Documentation

4.1.8.6.1. exception_number

```
enum exception_number
```

Exception number definitions.

On Arm these are vector table indices:

Name	Value	Exception
NMI_EXCEPTION	2	Non Maskable Interrupt
HARDFAULT_EXCEPTION	3	HardFault

MEMMANAGE_EXCEPTION | 4 | MemManage BUSFAULT_EXCEPTION | 5 | BusFault USAGEFAULT_EXCEPTION | 6 | UsageFault SECUREFAULT_EXCEPTION | 7 | SecureFault SVCALL_EXCEPTION | 11 | SV Call PENDSV_EXCEPTION | 14 | Pend SV SYSTICK_EXCEPTION | 15 | System Tick

On RISC-V these are exception cause numbers:

Name	Value	Exception
INSTR_ALIGN_EXCEPTION	0	Instruction fetch misaligned
INSTR_FAULT_EXCEPTION	1	Instruction fetch bus fault
INSTR_ILLEGAL_EXCEPTION	2	Invalid or illegal instruction
EBREAK_EXCEPTION	3	ebreak was not caught by an ex
LOAD_ALIGN_EXCEPTION	4	Load address not naturally ali
LOAD_FAULT_EXCEPTION	5	Load bus fault
STORE_ALIGN_EXCEPTION	6	Store or AMO address not natur
STORE_FAULT_EXCEPTION	7	Store or AMO bus fault
ECALL_UMODE_EXCEPTION	8	ecall was executed in U-mode
ECALL_SMODE_EXCEPTION	9	ecall was executed in S-mode
ECALL_MMODE_EXCEPTION	11	ecall was executed in M-mode

Table 16. Enumerator

NMI_EXCEPTION	Non Maskable Interrupt.
HARDFAULT_EXCEPTION	HardFault Interrupt.
MEMMANAGE_EXCEPTION	MemManage Interrupt.
BUSFAULT_EXCEPTION	BusFault Interrupt.
USAGEFAULT_EXCEPTION	UsageFault Interrupt.
SECUREFAULT_EXCEPTION	SecureFault Interrupt.
SVCALL_EXCEPTION	SV Call Interrupt.
PENDSV_EXCEPTION	Pend SV Interrupt.
SYSTICK_EXCEPTION	System Tick Interrupt.

4.1.8.7. Function Documentation

4.1.8.7.1. exception_get_priority

```
uint exception_get_priority (uint num)
```

Get specified exception's priority.

Numerically-lower values indicate a higher priority. Hardware priorities range from 0 (highest priority) to 255 (lowest priority).

Only the top 2 bits are significant on ARM Cortex-M0+ on RP2040.

Only the top 4 bits are significant on ARM Cortex-M33 on RP2350, and exception priorities are not supported on RISC-V

Parameters

`num`	Exception number exception_number

Returns

the exception priority

4.1.8.7.2. exception_get_vtable_handler

`exception_handler_t exception_get_vtable_handler (enum exception_number num)`

Get the current exception handler for the specified exception from the currently installed vector table of the execution core.

Parameters

`num`	Exception number

Returns

the address stored in the VTABLE for the given exception number

4.1.8.7.3. exception_restore_handler

`void exception_restore_handler (enum exception_number num, exception_handler_t original_handler)`

Restore the original exception handler for an exception on this core.

This method may be used to restore the exception handler for an exception on this core to the state prior to the call to exception_set_exclusive_handler(), so that exception_set_exclusive_handler() may be called again in the future.

Parameters

`num`	Exception number exception_number
`original_handler`	The original handler returned from exception_set_exclusive_handler

See also

exception_set_exclusive_handler()

4.1.8.7.4. exception_set_exclusive_handler

`exception_handler_t exception_set_exclusive_handler (enum exception_number num, exception_handler_t handler)`

Set the exception handler for an exception on the executing core.

This method will assert if an exception handler has been set for this exception number on this core via this method, without an intervening restore via exception_restore_handler.

NOTE

this method may not be used to override an exception handler that was specified at link time by providing a strong replacement for the weakly defined stub exception handlers. It will assert in this case too.

Parameters

`num`	Exception number
`handler`	The handler to set

See also

exception_number

4.1.8.7.5. exception_set_priority

`bool exception_set_priority (uint num, uint8_t hardware_priority)`

Set specified exception's priority.

Parameters

`num`	Exception number exception_number
`hardware_priority`	Priority to set.

Numerically-lower values indicate a higher priority. Hardware priorities range from 0 (highest priority) to 255 (lowest priority).

Only the top 2 bits are significant on ARM Cortex-M0+ on RP2040.

Only the top 4 bits are significant on ARM Cortex-M33 on RP2350, and exception priorities are not supported on RISC-V

4.1.9. hardware_flash

Low level flash programming and erase API.

4.1.9.1. Detailed Description

Note these functions are *unsafe* if you are using both cores, and the other is executing from flash concurrently with the operation. In this case, you must perform your own synchronisation to make sure that no XIP accesses take place during flash programming. One option is to use the lockout functions.

Likewise they are *unsafe* if you have interrupt handlers or an interrupt vector table in flash, so you must disable interrupts before calling in this case.

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

```

```c
// 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)

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(" ");
    }
}

// This function will be called when it's safe to call flash_range_erase
static void call_flash_range_erase(void *param) {
    uint32_t offset = (uint32_t)param;
    flash_range_erase(offset, FLASH_SECTOR_SIZE);
}

// This function will be called when it's safe to call flash_range_program
static void call_flash_range_program(void *param) {
    uint32_t offset = ((uintptr_t*)param)[0];
    const uint8_t *data = (const uint8_t *)((uintptr_t*)param)[1];
    flash_range_program(offset, data, FLASH_PAGE_SIZE);
}

int main() {
    stdio_init_all();
    uint8_t random_data[FLASH_PAGE_SIZE];
    for (uint 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 is "execute in place" and so will be in use when any code that is stored in flash runs, e.g. an interrupt handler
    // or code running on a different core.
    // Calling flash_range_erase or flash_range_program at the same time as flash is running code would cause a crash.
    // flash_safe_execute disables interrupts and tries to cooperate with the other core to ensure flash is not in use
    // See the documentation for flash_safe_execute and its assumptions and limitations
    int rc = flash_safe_execute(call_flash_range_erase, (void*)FLASH_TARGET_OFFSET, UINT32_MAX);
    hard_assert(rc == PICO_OK);

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

    printf("\nProgramming target region...\n");
    uintptr_t params[] = { FLASH_TARGET_OFFSET, (uintptr_t)random_data};
    rc = flash_safe_execute(call_flash_range_program, params, UINT32_MAX);
    hard_assert(rc == PICO_OK);
    printf("Done. Read back target region:\n");
    print_buf(flash_target_contents, FLASH_PAGE_SIZE);

    bool mismatch = false;
```

```
    for (uint 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");
}
```

4.1.9.2. Functions

`void flash_range_erase (uint32_t flash_offs, size_t count)`

Erase areas of flash.

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

Program flash.

`void flash_get_unique_id (uint8_t *id_out)`

Get flash unique 64 bit identifier.

`void flash_do_cmd (const uint8_t *txbuf, uint8_t *rxbuf, size_t count)`

Execute bidirectional flash command.

4.1.9.3. Function Documentation

4.1.9.3.1. flash_do_cmd

`void flash_do_cmd (const uint8_t * txbuf, uint8_t * rxbuf, size_t count)`

Execute bidirectional flash command.

Low-level function to execute a serial command on a flash device attached to the QSPI interface. Bytes are simultaneously transmitted and received from txbuf and to rxbuf. Therefore, both buffers must be the same length, count, which is the length of the overall transaction. This is useful for reading metadata from the flash chip, such as device ID or SFDP parameters.

The XIP cache is flushed following each command, in case flash state has been modified. Like other hardware_flash functions, the flash is not accessible for execute-in-place transfers whilst the command is in progress, so entering a flash-resident interrupt handler or executing flash code on the second core concurrently will be fatal. To avoid these pitfalls it is recommended that this function only be used to extract flash metadata during startup, before the main application begins to run: see the implementation of pico_get_unique_id() for an example of this.

Parameters

`txbuf`	Pointer to a byte buffer which will be transmitted to the flash
`rxbuf`	Pointer to a byte buffer where data received from the flash will be written. txbuf and rxbuf may be the same buffer.
`count`	Length in bytes of txbuf and of rxbuf

4.1.9.3.2. flash_get_unique_id

`void flash_get_unique_id (uint8_t * id_out)`

Get flash unique 64 bit identifier.

Use a standard 4Bh RUID instruction to retrieve the 64 bit unique identifier from a flash device attached to the QSPI interface. Since there is a 1:1 association between the MCU and this flash, this also serves as a unique identifier for the board.

Parameters

`id_out`	Pointer to an 8-byte buffer to which the ID will be written

4.1.9.3.3. 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).

> **NOTE**
>
> Erasing a flash sector sets all the bits in all the pages in that sector to one. You can then "program" flash pages in the sector to turn some of the bits to zero. Once a bit is set to zero it can only be changed back to one by erasing the whole sector again.

4.1.9.3.4. 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).

> **NOTE**
>
> : Programming a flash page effectively changes some of the bits from one to zero. The only way to change a zero bit back to one is to "erase" the whole sector that the page resides in. So you may need to make sure you have called flash_range_erase before calling flash_range_program.

4.1.10. hardware_gpio

General Purpose Input/Output (GPIO) API.

4.1.10.1. Detailed Description

RP-series microcontrollers have two banks of General Purpose Input / Output (GPIO) pins, which are assigned as follows:

RP2040 has 30 user GPIO pins in bank 0, and 6 QSPI pins in the QSPI bank 1 (QSPI_SS, QSPI_SCLK and QSPI_SD0 to QSPI_SD3). The QSPI pins are used to execute code from an external flash device, leaving the User bank (GPIO0 to GPIO29) for the programmer to use.

The number of GPIO pins available depends on the package. There are 30 user GPIOs in bank 0 in the QFN-60 package (RP2350A), or 48 user GPIOs in the QFN-80 package. Bank 1 contains the 6 QSPI pins and the USB DP/DM pins.

All GPIOs support digital input and output, but a subset can also be used as inputs to the chip's Analogue to Digital Converter (ADC). The allocation of GPIO pins to the ADC depends on the packaging.

RP2040 and RP2350 QFN-60 GPIO, ADC pins are 26-29. RP2350 QFN-80, ADC pins are 40-47.

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.

NOTE

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 datasheet for more information on GPIO function select.

Function Select Table

On RP2040 the function selects are:

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

GPIO	F1	F2	F3	F4	F5	F6	F7	F8	F9
12	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1		USB OVCUR DET
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

On RP2350 the function selects are:

GPIO	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
0		SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1	PIO2	XIP_CS 1n	USB OVCUR DET	
1		SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO	PIO0	PIO1	PIO2	TRACE CLK	USB VBUS DET	
2		SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO	PIO0	PIO1	PIO2	TRACE DATA0	USB VBUS EN	UART0 TX
3		SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO	PIO0	PIO1	PIO2	TRACE DATA1	USB OVCUR DET	UART0 RX
4		SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1	PIO2	TRACE DATA2	USB VBUS DET	
5		SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO	PIO0	PIO1	PIO2	TRACE DATA3	USB VBUS EN	
6		SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	UART1 TX
7		SPI0 TX	UART1 RTS	I2C1 SCL	PWM3 B	SIO	PIO0	PIO1	PIO2		USB VBUS DET	UART1 RX
8		SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1	PIO2	XIP_CS 1n	USB VBUS EN	
9		SPI1 CSn	UART1 RX	I2C0 SCL	PWM4 B	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	
10		SPI1 SCK	UART1 CTS	I2C1 SDA	PWM5 A	SIO	PIO0	PIO1	PIO2		USB VBUS DET	UART1 TX
11		SPI1 TX	UART1 RTS	I2C1 SCL	PWM5 B	SIO	PIO0	PIO1	PIO2		USB VBUS EN	UART1 RX
12	HSTX	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1	PIO2	CLOCK GPIN0	USB OVCUR DET	
13	HSTX	SPI1 CSn	UART0 RX	I2C0 SCL	PWM6 B	SIO	PIO0	PIO1	PIO2	CLOCK GPOUT 0	USB VBUS DET	
14	HSTX	SPI1 SCK	UART0 CTS	I2C1 SDA	PWM7 A	SIO	PIO0	PIO1	PIO2	CLOCK GPIN1	USB VBUS EN	UART0 TX

GPIO	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
15	HSTX	SPI1 TX	UART0 RTS	I2C1 SCL	PWM7 B	SIO	PIO0	PIO1	PIO2	CLOCK GPOUT 1	USB OVCUR DET	UART0 RX
16	HSTX	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1	PIO2		USB VBUS DET	
17	HSTX	SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO	PIO0	PIO1	PIO2		USB VBUS EN	
18	HSTX	SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	UART0 TX
19	HSTX	SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO	PIO0	PIO1	PIO2	XIP_CS1n	USB VBUS DET	UART0 RX
20		SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1	PIO2	CLOCK GPIN0	USB VBUS EN	
21		SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO	PIO0	PIO1	PIO2	CLOCK GPOUT 0	USB OVCUR DET	
22		SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO	PIO0	PIO1	PIO2	CLOCK GPIN1	USB VBUS DET	UART1 TX
23		SPI0 TX	UART1 RTS	I2C1 SCL	PWM3 B	SIO	PIO0	PIO1	PIO2	CLOCK GPOUT 1	USB VBUS EN	UART1 RX
24		SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1	PIO2	CLOCK GPOUT 2	USB OVCUR DET	
25		SPI1 CSn	UART1 RX	I2C0 SCL	PWM4 B	SIO	PIO0	PIO1	PIO2	CLOCK GPOUT 3	USB VBUS DET	
26		SPI1 SCK	UART1 CTS	I2C1 SDA	PWM5 A	SIO	PIO0	PIO1	PIO2		USB VBUS EN	UART1 TX
27		SPI1 TX	UART1 RTS	I2C1 SCL	PWM5 B	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	UART1 RX
28		SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1	PIO2		USB VBUS DET	
29		SPI1 CSn	UART0 RX	I2C0 SCL	PWM6 B	SIO	PIO0	PIO1	PIO2		USB VBUS EN	

GPIOs 30 through 47 are QFN-80 only:

GPIO	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
30		SPI1 SCK	UART0 CTS	I2C1 SDA	PWM7 A	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	UART0 TX
31		SPI1 TX	UART0 RTS	I2C1 SCL	PWM7 B	SIO	PIO0	PIO1	PIO2		USB VBUS DET	UART0 RX
32		SPI0 RX	UART0 TX	I2C0 SDA	PWM8 A	SIO	PIO0	PIO1	PIO2		USB VBUS EN	
33		SPI0 CSn	UART0 RX	I2C0 SCL	PWM8 B	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	
34		SPI0 SCK	UART0 CTS	I2C1 SDA	PWM9 A	SIO	PIO0	PIO1	PIO2		USB VBUS DET	UART0 TX
35		SPI0 TX	UART0 RTS	I2C1 SCL	PWM9 B	SIO	PIO0	PIO1	PIO2		USB VBUS EN	UART0 RX
36		SPI0 RX	UART1 TX	I2C0 SDA	PWM1 0 A	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	
37		SPI0 CSn	UART1 RX	I2C0 SCL	PWM1 0 B	SIO	PIO0	PIO1	PIO2		USB VBUS DET	
38		SPI0 SCK	UART1 CTS	I2C1 SDA	PWM1 1 A	SIO	PIO0	PIO1	PIO2		USB VBUS EN	UART1 TX
39		SPI0 TX	UART1 RTS	I2C1 SCL	PWM1 1 B	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	UART1 RX
40		SPI1 RX	UART1 TX	I2C0 SDA	PWM8 A	SIO	PIO0	PIO1	PIO2		USB VBUS DET	
41		SPI1 CSn	UART1 RX	I2C0 SCL	PWM8 B	SIO	PIO0	PIO1	PIO2		USB VBUS EN	
42		SPI1 SCK	UART1 CTS	I2C1 SDA	PWM9 A	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	UART1 TX
43		SPI1 TX	UART1 RTS	I2C1 SCL	PWM9 B	SIO	PIO0	PIO1	PIO2		USB VBUS DET	UART1 RX
44		SPI1 RX	UART0 TX	I2C0 SDA	PWM1 0 A	SIO	PIO0	PIO1	PIO2		USB VBUS EN	

GPIO	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
45		SPI1 CSn	UART0 RX	I2C0 SCL	PWM1 0 B	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	
46		SPI1 SCK	UART0 CTS	I2C1 SDA	PWM1 1 A	SIO	PIO0	PIO1	PIO2		USB VBUS DET	UART0 TX
47		SPI1 TX	UART0 RTS	I2C1 SCL	PWM1 1 B	SIO	PIO0	PIO1	PIO2	XIP_CS 1n	USB VBUS EN	UART0 RX

4.1.10.2. Typedefs

`typedef enum gpio_function_rp2040 gpio_function_t` RP2040

GPIO pin function selectors on RP2040 (used as typedef gpio_function_t)

`typedef enum gpio_function_rp2350 gpio_function_t` RP2350

GPIO pin function selectors on RP2350 (used as typedef gpio_function_t)

`typedef void(* gpio_irq_callback_t)(uint gpio, uint32_t event_mask)`

4.1.10.3. Enumerations

`enum gpio_function_rp2040 { 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 = 0x1f }` RP2040

GPIO pin function selectors on RP2040 (used as typedef gpio_function_t)

`enum gpio_function_rp2350 { GPIO_FUNC_HSTX = 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_PIO2 = 8, GPIO_FUNC_GPCK = 9, GPIO_FUNC_XIP_CS1 = 9, GPIO_FUNC_CORESIGHT_TRACE = 9, GPIO_FUNC_USB = 10, GPIO_FUNC_UART_AUX = 11, GPIO_FUNC_NULL = 0x1f }` RP2350

GPIO pin function selectors on RP2350 (used as typedef gpio_function_t)

`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 (GPIO events)

`enum gpio_slew_rate { GPIO_SLEW_RATE_SLOW = 0, GPIO_SLEW_RATE_FAST = 1 }`

Slew rate limiting levels for GPIO outputs.

`enum gpio_drive_strength { GPIO_DRIVE_STRENGTH_2MA = 0, GPIO_DRIVE_STRENGTH_4MA = 1, GPIO_DRIVE_STRENGTH_8MA = 2, GPIO_DRIVE_STRENGTH_12MA = 3 }`

Drive strength levels for GPIO outputs.

4.1.10.4. Functions

`void gpio_set_function (uint gpio, gpio_function_t fn)`

Select GPIO function.

`void gpio_set_function_masked (uint32_t gpio_mask, gpio_function_t fn)`

Select the function for multiple GPIOs.

`void gpio_set_function_masked64 (uint64_t gpio_mask, gpio_function_t fn)`

Select the function for multiple GPIOs.

`gpio_function_t gpio_get_function (uint gpio)` RP2040

Determine current GPIO function.

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

Select up and down pulls on specific GPIO.

`static void gpio_pull_up (uint gpio)`

Set specified GPIO to be pulled up.

`static bool gpio_is_pulled_up (uint gpio)`

Determine if the specified GPIO is pulled up.

`static void gpio_pull_down (uint gpio)`

Set specified GPIO to be pulled down.

`static bool gpio_is_pulled_down (uint gpio)`

Determine if the specified GPIO is pulled down.

`static void gpio_disable_pulls (uint gpio)`

Disable pulls on specified GPIO.

`void gpio_set_irqover (uint gpio, uint value)`

Set GPIO IRQ override.

`void gpio_set_outover (uint gpio, uint value)`

Set GPIO output override.

`void gpio_set_inover (uint gpio, uint value)`

Select GPIO input override.

`void gpio_set_oeover (uint gpio, uint value)`

Select GPIO output enable override.

`void gpio_set_input_enabled (uint gpio, bool enabled)`

Enable GPIO input.

`void gpio_set_input_hysteresis_enabled (uint gpio, bool enabled)`

Enable/disable GPIO input hysteresis (Schmitt trigger)

`bool gpio_is_input_hysteresis_enabled (uint gpio)`

Determine whether input hysteresis is enabled on a specified GPIO.

`void gpio_set_slew_rate (uint gpio, enum gpio_slew_rate slew)`

Set slew rate for a specified GPIO.

`enum gpio_slew_rate gpio_get_slew_rate (uint gpio)`

Determine current slew rate for a specified GPIO.

`void gpio_set_drive_strength (uint gpio, enum gpio_drive_strength drive)`

Set drive strength for a specified GPIO.

`enum gpio_drive_strength gpio_get_drive_strength (uint gpio)`

Determine current drive strength for a specified GPIO.

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

Enable or disable specific interrupt events for specified GPIO.

`void gpio_set_irq_callback (gpio_irq_callback_t callback)`

Set the generic callback used for GPIO IRQ events for the current core.

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

Convenience function which performs multiple GPIO IRQ related initializations.

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

Enable dormant wake up interrupt for specified GPIO and events.

`static uint32_t gpio_get_irq_event_mask (uint gpio)`

Return the current interrupt status (pending events) for the given GPIO.

`void gpio_acknowledge_irq (uint gpio, uint32_t event_mask)`

Acknowledge a GPIO interrupt for the specified events on the calling core.

`void gpio_add_raw_irq_handler_with_order_priority_masked (uint32_t gpio_mask, irq_handler_t handler, uint8_t order_priority)`

Adds a raw GPIO IRQ handler for the specified GPIOs on the current core.

`void gpio_add_raw_irq_handler_with_order_priority_masked64 (uint64_t gpio_mask, irq_handler_t handler, uint8_t order_priority)`

Adds a raw GPIO IRQ handler for the specified GPIOs on the current core.

`static void gpio_add_raw_irq_handler_with_order_priority (uint gpio, irq_handler_t handler, uint8_t order_priority)`

Adds a raw GPIO IRQ handler for a specific GPIO on the current core.

`void gpio_add_raw_irq_handler_masked (uint32_t gpio_mask, irq_handler_t handler)`

Adds a raw GPIO IRQ handler for the specified GPIOs on the current core.

`void gpio_add_raw_irq_handler_masked64 (uint64_t gpio_mask, irq_handler_t handler)`

Adds a raw GPIO IRQ handler for the specified GPIOs on the current core.

`static void gpio_add_raw_irq_handler (uint gpio, irq_handler_t handler)`

Adds a raw GPIO IRQ handler for a specific GPIO on the current core.

`void gpio_remove_raw_irq_handler_masked (uint32_t gpio_mask, irq_handler_t handler)`

Removes a raw GPIO IRQ handler for the specified GPIOs on the current core.

`void gpio_remove_raw_irq_handler_masked64 (uint64_t gpio_mask, irq_handler_t handler)`

Removes a raw GPIO IRQ handler for the specified GPIOs on the current core.

`static void gpio_remove_raw_irq_handler (uint gpio, irq_handler_t handler)`

Removes a raw GPIO IRQ handler for the specified GPIO on the current core.

`void gpio_init (uint gpio)`

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

`void gpio_deinit (uint gpio)`

Resets a GPIO back to the NULL function, i.e. disables it.

`void gpio_init_mask (uint gpio_mask)`

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

`static bool gpio_get (uint gpio)`

Get state of a single specified GPIO.

`static uint32_t gpio_get_all (void)`

Get raw value of all GPIOs.

`static uint64_t gpio_get_all64 (void)`

Get raw value of all GPIOs.

`static void gpio_set_mask (uint32_t mask)`

Drive high every GPIO appearing in mask.

`static void gpio_set_mask64 (uint64_t mask)`

Drive high every GPIO appearing in mask.

`static void gpio_set_mask_n (uint n, uint32_t mask)`

Drive high every GPIO appearing in mask.

`static void gpio_clr_mask (uint32_t mask)`

Drive low every GPIO appearing in mask.

`static void gpio_clr_mask64 (uint64_t mask)`

Drive low every GPIO appearing in mask.

`static void gpio_clr_mask_n (uint n, uint32_t mask)`

Drive low every GPIO appearing in mask.

`static void gpio_xor_mask (uint32_t mask)`

Toggle every GPIO appearing in mask.

`static void gpio_xor_mask64 (uint64_t mask)`

Toggle every GPIO appearing in mask.

`static void gpio_xor_mask_n (uint n, uint32_t mask)`

Toggle every GPIO appearing in mask.

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

Drive GPIOs high/low depending on parameters.

`static void gpio_put_masked64 (uint64_t mask, uint64_t value)`

Drive GPIOs high/low depending on parameters.

`static void gpio_put_masked_n (uint n, uint32_t mask, uint32_t value)`

Drive GPIOs high/low depending on parameters.

`static void gpio_put_all (uint32_t value)`

Drive all pins simultaneously.

`static void gpio_put_all64 (uint64_t value)`

Drive all pins simultaneously.

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

Drive a single GPIO high/low.

`static bool gpio_get_out_level (uint gpio)`

Determine whether a GPIO is currently driven high or low.

`static void gpio_set_dir_out_masked (uint32_t mask)`

Set a number of GPIOs to output.

`static void gpio_set_dir_out_masked64 (uint64_t mask)`

Set a number of GPIOs to output.

`static void gpio_set_dir_in_masked (uint32_t mask)`

Set a number of GPIOs to input.

`static void gpio_set_dir_in_masked64 (uint64_t mask)`

Set a number of GPIOs to input.

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

Set multiple GPIO directions.

`static void gpio_set_dir_masked64 (uint64_t mask, uint64_t value)`

Set multiple GPIO directions.

`static void gpio_set_dir_all_bits (uint32_t values)`

Set direction of all pins simultaneously.

`static void gpio_set_dir_all_bits64 (uint64_t values)`

Set direction of all pins simultaneously.

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

Set a single GPIO direction.

`static bool gpio_is_dir_out (uint gpio)`

Check if a specific GPIO direction is OUT.

`static uint gpio_get_dir (uint gpio)`

Get a specific GPIO direction.

4.1.10.5. Typedef Documentation

4.1.10.5.1. gpio_function_t RP2040

```
typedef enum gpio_function_rp2040 gpio_function_t
```

GPIO pin function selectors on RP2040 (used as typedef gpio_function_t)

4.1.10.5.2. gpio_function_t RP2350

```
typedef enum gpio_function_rp2350 gpio_function_t
```

GPIO pin function selectors on RP2350 (used as typedef gpio_function_t)

4.1.10.5.3. gpio_irq_callback_t

```
typedef void(* gpio_irq_callback_t) (uint gpio, uint32_t event_mask)
```

Callback function type for GPIO events

Parameters

`gpio`	Which GPIO caused this interrupt
`event_mask`	Which events caused this interrupt. See gpio_irq_level for details.

See also

gpio_set_irq_enabled_with_callback()

gpio_set_irq_callback()

4.1.10.6. Enumeration Type Documentation

4.1.10.6.1. gpio_function_rp2040 RP2040

`enum gpio_function_rp2040`

GPIO pin function selectors on RP2040 (used as typedef gpio_function_t)

Table 17. Enumerator

GPIO_FUNC_XIP	Select XIP as GPIO pin function.
GPIO_FUNC_SPI	Select SPI as GPIO pin function.
GPIO_FUNC_UART	Select UART as GPIO pin function.
GPIO_FUNC_I2C	Select I2C as GPIO pin function.
GPIO_FUNC_PWM	Select PWM as GPIO pin function.
GPIO_FUNC_SIO	Select SIO as GPIO pin function.
GPIO_FUNC_PIO0	Select PIO0 as GPIO pin function.
GPIO_FUNC_PIO1	Select PIO1 as GPIO pin function.
GPIO_FUNC_GPCK	Select GPCK as GPIO pin function.
GPIO_FUNC_USB	Select USB as GPIO pin function.
GPIO_FUNC_NULL	Select NULL as GPIO pin function.

4.1.10.6.2. gpio_function_rp2350 RP2350

`enum gpio_function_rp2350`

GPIO pin function selectors on RP2350 (used as typedef gpio_function_t)

Table 18. Enumerator

GPIO_FUNC_HSTX	Select HSTX as GPIO pin function.
GPIO_FUNC_SPI	Select SPI as GPIO pin function.
GPIO_FUNC_UART	Select UART as GPIO pin function.
GPIO_FUNC_I2C	Select I2C as GPIO pin function.
GPIO_FUNC_PWM	Select PWM as GPIO pin function.
GPIO_FUNC_SIO	Select SIO as GPIO pin function.
GPIO_FUNC_PIO0	Select PIO0 as GPIO pin function.
GPIO_FUNC_PIO1	Select PIO1 as GPIO pin function.
GPIO_FUNC_PIO2	Select PIO2 as GPIO pin function.
GPIO_FUNC_GPCK	Select GPCK as GPIO pin function.
GPIO_FUNC_XIP_CS1	Select XIP CS1 as GPIO pin function.
GPIO_FUNC_CORESIGHT_TRACE	Select CORESIGHT TRACE as GPIO pin function.
GPIO_FUNC_USB	Select USB as GPIO pin function.
GPIO_FUNC_UART_AUX	Select UART_AUX as GPIO pin function.
GPIO_FUNC_NULL	Select NULL as GPIO pin function.

4.1.10.6.3. gpio_irq_level

`enum gpio_irq_level`

GPIO Interrupt level definitions (GPIO events)

GPIO Interrupt levels

An interrupt can be generated for every GPIO pin in 4 scenarios:

- Level High: the GPIO pin is a logical 1
- Level Low: the GPIO pin is a logical 0
- Edge High: the GPIO has transitioned from a logical 0 to a logical 1
- Edge Low: the GPIO has transitioned from a logical 1 to a logical 0

The level interrupts are not latched. This means that if the pin is a logical 1 and the level high interrupt is active, it will become inactive as soon as the pin changes to a logical 0. The edge interrupts are stored in the INTR register and can be cleared by writing to the INTR register.

Table 19. Enumerator

GPIO_IRQ_LEVEL_LOW	IRQ when the GPIO pin is a logical 0.
GPIO_IRQ_LEVEL_HIGH	IRQ when the GPIO pin is a logical 1.
GPIO_IRQ_EDGE_FALL	IRQ when the GPIO has transitioned from a logical 1 to a logical 0.
GPIO_IRQ_EDGE_RISE	IRQ when the GPIO has transitioned from a logical 0 to a logical 1.

4.1.10.6.4. gpio_slew_rate

`enum gpio_slew_rate`

Slew rate limiting levels for GPIO outputs.

Slew rate limiting increases the minimum rise/fall time when a GPIO output is lightly loaded, which can help to reduce electromagnetic emissions.

See also

gpio_set_slew_rate

Table 20. Enumerator

GPIO_SLEW_RATE_SLOW	Slew rate limiting enabled.
GPIO_SLEW_RATE_FAST	Slew rate limiting disabled.

4.1.10.6.5. gpio_drive_strength

`enum gpio_drive_strength`

Drive strength levels for GPIO outputs.

Drive strength levels for GPIO outputs.

See also

gpio_set_drive_strength

Table 21. Enumerator

GPIO_DRIVE_STRENGTH_2MA	2 mA nominal drive strength
GPIO_DRIVE_STRENGTH_4MA	4 mA nominal drive strength
GPIO_DRIVE_STRENGTH_8MA	8 mA nominal drive strength
GPIO_DRIVE_STRENGTH_12MA	12 mA nominal drive strength

4.1.10.7. Function Documentation

4.1.10.7.1. gpio_acknowledge_irq

`void gpio_acknowledge_irq (uint gpio, uint32_t event_mask)`

Acknowledge a GPIO interrupt for the specified events on the calling core.

NOTE

This may be called with a mask of any of valid bits specified in gpio_irq_level, however it has no effect on *level* sensitive interrupts which remain pending while the GPIO is at the specified level. When handling *level* sensitive interrupts, you should generally disable the interrupt (see gpio_set_irq_enabled) and then set it up again later once the GPIO level has changed (or to catch the opposite level).

Parameters

`gpio` GPIO number

NOTE

For callbacks set with gpio_set_irq_enabled_with_callback, or gpio_set_irq_callback, this function is called automatically.

Parameters

`event_mask` Bitmask of events to clear. See gpio_irq_level for details.

4.1.10.7.2. gpio_add_raw_irq_handler

`static void gpio_add_raw_irq_handler (uint gpio, irq_handler_t handler) [inline], [static]`

Adds a raw GPIO IRQ handler for a specific GPIO on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see gpio_set_irq_callback), it is possible to add explicit GPIO IRQ handlers which are called independent of the default event callback.

This method adds such a callback, and disables the "default" callback for the specified GPIO.

NOTE

Multiple raw handlers should not be added for the same GPIO, and this method will assert if you attempt to. Internally, this function calls irq_add_shared_handler, which will assert if the maximum number of shared handlers (configurable via PICO_MAX_IRQ_SHARED_HANDLERS) would be exceeded.

A raw handler should check for whichever GPIOs and events it handles, and acknowledge them itself; it might look something like:

```
void my_irq_handler(void) {
    if (gpio_get_irq_event_mask(my_gpio_num) & my_gpio_event_mask) {
       gpio_acknowledge_irq(my_gpio_num, my_gpio_event_mask);
      // handle the IRQ
    }
}
```

Parameters

`gpio`	the GPIO number that will no longer be passed to the default callback for this core
`handler`	the handler to add to the list of GPIO IRQ handlers for this core

4.1.10.7.3. gpio_add_raw_irq_handler_masked

`void gpio_add_raw_irq_handler_masked (uint32_t gpio_mask, irq_handler_t handler)`

Adds a raw GPIO IRQ handler for the specified GPIOs on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see gpio_set_irq_callback), it is possible to add explicit GPIO IRQ handlers which are called independent of the default event callback.

This method adds such a callback, and disables the "default" callback for the specified GPIOs.

> **NOTE**
>
> Multiple raw handlers should not be added for the same GPIOs, and this method will assert if you attempt to. Internally, this function calls irq_add_shared_handler, which will assert if the maximum number of shared handlers (configurable via PICO_MAX_IRQ_SHARED_HANDLERS) would be exceeded.

A raw handler should check for whichever GPIOs and events it handles, and acknowledge them itself; it might look something like:

```c
void my_irq_handler(void) {
    if (gpio_get_irq_event_mask(my_gpio_num) & my_gpio_event_mask) {
       gpio_acknowledge_irq(my_gpio_num, my_gpio_event_mask);
      // handle the IRQ
    }
    if (gpio_get_irq_event_mask(my_gpio_num2) & my_gpio_event_mask2) {
       gpio_acknowledge_irq(my_gpio_num2, my_gpio_event_mask2);
      // handle the IRQ
    }
}
```

Parameters

`gpio_mask`	a bit mask of the GPIO numbers that will no longer be passed to the default callback for this core
`handler`	the handler to add to the list of GPIO IRQ handlers for this core

4.1.10.7.4. gpio_add_raw_irq_handler_masked64

`void gpio_add_raw_irq_handler_masked64 (uint64_t gpio_mask, irq_handler_t handler)`

Adds a raw GPIO IRQ handler for the specified GPIOs on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see gpio_set_irq_callback), it is possible to add explicit GPIO IRQ handlers which are called independent of the default event callback.

This method adds such a callback, and disables the "default" callback for the specified GPIOs.

NOTE

Multiple raw handlers should not be added for the same GPIOs, and this method will assert if you attempt to. Internally, this function calls irq_add_shared_handler, which will assert if the maximum number of shared handlers (configurable via PICO_MAX_IRQ_SHARED_HANDLERS) would be exceeded.

A raw handler should check for whichever GPIOs and events it handles, and acknowledge them itself; it might look something like:

```
void my_irq_handler(void) {
    if (gpio_get_irq_event_mask(my_gpio_num) & my_gpio_event_mask) {
       gpio_acknowledge_irq(my_gpio_num, my_gpio_event_mask);
      // handle the IRQ
    }
    if (gpio_get_irq_event_mask(my_gpio_num2) & my_gpio_event_mask2) {
       gpio_acknowledge_irq(my_gpio_num2, my_gpio_event_mask2);
      // handle the IRQ
    }
}
```

Parameters

`gpio_mask`	a 64 bit mask of the GPIO numbers that will no longer be passed to the default callback for this core
`handler`	the handler to add to the list of GPIO IRQ handlers for this core

4.1.10.7.5. gpio_add_raw_irq_handler_with_order_priority

`static void gpio_add_raw_irq_handler_with_order_priority (uint gpio, irq_handler_t handler, uint8_t order_priority) [inline], [static]`

Adds a raw GPIO IRQ handler for a specific GPIO on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see gpio_set_irq_callback), it is possible to add explicit GPIO IRQ handlers which are called independent of the default callback. The order relative to the default callback can be controlled via the order_priority parameter(the default callback has the priority GPIO_IRQ_CALLBACK_ORDER_PRIORITY which defaults to the lowest priority with the intention of it running last).

This method adds such a callback, and disables the "default" callback for the specified GPIO.

NOTE

Multiple raw handlers should not be added for the same GPIO, and this method will assert if you attempt to. Internally, this function calls irq_add_shared_handler, which will assert if the maximum number of shared handlers (configurable via PICO_MAX_IRQ_SHARED_HANDLERS) would be exceeded.

A raw handler should check for whichever GPIOs and events it handles, and acknowledge them itself; it might look something like:

```
void my_irq_handler(void) {
    if (gpio_get_irq_event_mask(my_gpio_num) & my_gpio_event_mask) {
       gpio_acknowledge_irq(my_gpio_num, my_gpio_event_mask);
      // handle the IRQ
    }
}
```

Parameters

`gpio`	the GPIO number that will no longer be passed to the default callback for this core
`handler`	the handler to add to the list of GPIO IRQ handlers for this core
`order_priority`	the priority order to determine the relative position of the handler in the list of GPIO IRQ handlers for this core.

4.1.10.7.6. gpio_add_raw_irq_handler_with_order_priority_masked

`void gpio_add_raw_irq_handler_with_order_priority_masked (uint32_t gpio_mask, irq_handler_t handler, uint8_t order_priority)`

Adds a raw GPIO IRQ handler for the specified GPIOs on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see gpio_set_irq_callback), it is possible to add explicit GPIO IRQ handlers which are called independent of the default callback. The order relative to the default callback can be controlled via the order_priority parameter (the default callback has the priority GPIO_IRQ_CALLBACK_ORDER_PRIORITY which defaults to the lowest priority with the intention of it running last).

This method adds such an explicit GPIO IRQ handler, and disables the "default" callback for the specified GPIOs.

> **NOTE**
>
> Multiple raw handlers should not be added for the same GPIOs, and this method will assert if you attempt to. Internally, this function calls irq_add_shared_handler, which will assert if the maximum number of shared handlers (configurable via PICO_MAX_IRQ_SHARED_HANDLERS) would be exceeded.

A raw handler should check for whichever GPIOs and events it handles, and acknowledge them itself; it might look something like:

```
void my_irq_handler(void) {
    if (gpio_get_irq_event_mask(my_gpio_num) & my_gpio_event_mask) {
       gpio_acknowledge_irq(my_gpio_num, my_gpio_event_mask);
      // handle the IRQ
    }
    if (gpio_get_irq_event_mask(my_gpio_num2) & my_gpio_event_mask2) {
       gpio_acknowledge_irq(my_gpio_num2, my_gpio_event_mask2);
      // handle the IRQ
    }
}
```

Parameters

`gpio_mask`	a bit mask of the GPIO numbers that will no longer be passed to the default callback for this core
`handler`	the handler to add to the list of GPIO IRQ handlers for this core
`order_priority`	the priority order to determine the relative position of the handler in the list of GPIO IRQ handlers for this core.

4.1.10.7.7. gpio_add_raw_irq_handler_with_order_priority_masked64

`void gpio_add_raw_irq_handler_with_order_priority_masked64 (uint64_t gpio_mask, irq_handler_t handler, uint8_t order_priority)`

Adds a raw GPIO IRQ handler for the specified GPIOs on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see gpio_set_irq_callback), it is possible to add explicit GPIO IRQ handlers which are called independent of the default callback. The order relative to the default callback can be controlled via the order_priority parameter (the default callback has the priority GPIO_IRQ_CALLBACK_ORDER_PRIORITY which defaults to the lowest priority with the intention of it running last).

This method adds such an explicit GPIO IRQ handler, and disables the "default" callback for the specified GPIOs.

NOTE

> Multiple raw handlers should not be added for the same GPIOs, and this method will assert if you attempt to. Internally, this function calls irq_add_shared_handler, which will assert if the maximum number of shared handlers (configurable via PICO_MAX_IRQ_SHARED_HANDLERS) would be exceeded.

A raw handler should check for whichever GPIOs and events it handles, and acknowledge them itself; it might look something like:

```
void my_irq_handler(void) {
    if (gpio_get_irq_event_mask(my_gpio_num) & my_gpio_event_mask) {
       gpio_acknowledge_irq(my_gpio_num, my_gpio_event_mask);
      // handle the IRQ
    }
    if (gpio_get_irq_event_mask(my_gpio_num2) & my_gpio_event_mask2) {
       gpio_acknowledge_irq(my_gpio_num2, my_gpio_event_mask2);
      // handle the IRQ
    }
}
```

Parameters

`gpio_mask`	a bit mask of the GPIO numbers that will no longer be passed to the default callback for this core
`handler`	the handler to add to the list of GPIO IRQ handlers for this core
`order_priority`	the priority order to determine the relative position of the handler in the list of GPIO IRQ handlers for this core.

4.1.10.7.8. gpio_clr_mask

`static void gpio_clr_mask (uint32_t mask) [inline], [static]`

Drive low every GPIO appearing in mask.

Parameters

`mask`	Bitmask of GPIO values to clear

4.1.10.7.9. gpio_clr_mask64

`static void gpio_clr_mask64 (uint64_t mask) [inline], [static]`

Drive low every GPIO appearing in mask.

Parameters

`mask`	Bitmask of GPIO values to clear

4.1.10.7.10. gpio_clr_mask_n

`static void gpio_clr_mask_n (uint n, uint32_t mask) [inline], [static]`

Drive low every GPIO appearing in mask.

Parameters

`n`	the base GPIO index of the mask to update. n == 0 means 0->31, n == 1 mean 32->63 etc.
`mask`	Bitmask of 32 GPIO values to clear

4.1.10.7.11. gpio_deinit

`void gpio_deinit (uint gpio)`

Resets a GPIO back to the NULL function, i.e. disables it.

Parameters

`gpio`	GPIO number

4.1.10.7.12. gpio_disable_pulls

`static void gpio_disable_pulls (uint gpio) [inline], [static]`

Disable pulls on specified GPIO.

Parameters

`gpio`	GPIO number

4.1.10.7.13. gpio_get

`static bool gpio_get (uint gpio) [inline], [static]`

Get state of a single specified GPIO.

Parameters

`gpio`	GPIO number

Returns

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

4.1.10.7.14. gpio_get_all

`static uint32_t gpio_get_all (void) [inline], [static]`

Get raw value of all GPIOs.

Returns

Bitmask of raw GPIO values

4.1.10.7.15. gpio_get_all64

`static uint64_t gpio_get_all64 (void) [inline], [static]`

Get raw value of all GPIOs.

Returns

Bitmask of raw GPIO values

4.1.10.7.16. gpio_get_dir

`static uint gpio_get_dir (uint gpio) [inline], [static]`

Get a specific GPIO direction.

Parameters

`gpio`	GPIO number

Returns

1 for out, 0 for in

4.1.10.7.17. gpio_get_drive_strength

`enum gpio_drive_strength gpio_get_drive_strength (uint gpio)`

Determine current drive strength for a specified GPIO.

See also

gpio_set_drive_strength

Parameters

`gpio`	GPIO number

Returns

Current drive strength of that GPIO

4.1.10.7.18. gpio_get_function

`gpio_function_t gpio_get_function (uint gpio)`

Determine current GPIO function.

Parameters

`gpio`	GPIO number

Returns

Which GPIO function is currently selected from list gpio_function_t

4.1.10.7.19. gpio_get_irq_event_mask

`static uint32_t gpio_get_irq_event_mask (uint gpio) [inline], [static]`

Return the current interrupt status (pending events) for the given GPIO.

Parameters

`gpio`	GPIO number

Returns

Bitmask of events that are currently pending for the GPIO. See gpio_irq_level for details.

See also

gpio_acknowledge_irq

4.1.10.7.20. gpio_get_out_level

`static bool gpio_get_out_level (uint gpio) [inline], [static]`

Determine whether a GPIO is currently driven high or low.

This function returns the high/low output level most recently assigned to a GPIO via gpio_put() or similar. This is the value that is presented outward to the IO muxing, *not* the input level back from the pad (which can be read using gpio_get()).

To avoid races, this function must not be used for read-modify-write sequences when driving GPIOs – instead functions like gpio_put() should be used to atomically update GPIOs. This accessor is intended for debug use only.

Parameters

`gpio` GPIO number

Returns

true if the GPIO output level is high, false if low.

4.1.10.7.21. gpio_get_slew_rate

`enum gpio_slew_rate gpio_get_slew_rate (uint gpio)`

Determine current slew rate for a specified GPIO.

See also

gpio_set_slew_rate

Parameters

`gpio` GPIO number

Returns

Current slew rate of that GPIO

4.1.10.7.22. 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

4.1.10.7.23. 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

4.1.10.7.24. gpio_is_dir_out

`static bool gpio_is_dir_out (uint gpio) [inline], [static]`

Check if a specific GPIO direction is OUT.

Parameters

`gpio`	GPIO number

Returns

true if the direction for the pin is OUT

4.1.10.7.25. gpio_is_input_hysteresis_enabled

`bool gpio_is_input_hysteresis_enabled (uint gpio)`

Determine whether input hysteresis is enabled on a specified GPIO.

See also

gpio_set_input_hysteresis_enabled

Parameters

`gpio`	GPIO number

4.1.10.7.26. gpio_is_pulled_down

`static bool gpio_is_pulled_down (uint gpio) [inline], [static]`

Determine if the specified GPIO is pulled down.

Parameters

`gpio`	GPIO number

Returns

true if the GPIO is pulled down

4.1.10.7.27. gpio_is_pulled_up

`static bool gpio_is_pulled_up (uint gpio) [inline], [static]`

Determine if the specified GPIO is pulled up.

Parameters

`gpio`	GPIO number

Returns

true if the GPIO is pulled up

4.1.10.7.28. gpio_pull_down

`static void gpio_pull_down (uint gpio) [inline], [static]`

Set specified GPIO to be pulled down.

Parameters

`gpio`	GPIO number

4.1.10.7.29. gpio_pull_up

`static void gpio_pull_up (uint gpio) [inline], [static]`

Set specified GPIO to be pulled up.

Parameters

`gpio`	GPIO number

4.1.10.7.30. gpio_put

`static void gpio_put (uint gpio, bool value) [inline], [static]`

Drive a single GPIO high/low.

Parameters

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

4.1.10.7.31. gpio_put_all

`static void gpio_put_all (uint32_t value) [inline], [static]`

Drive all pins simultaneously.

Parameters

`value`	Bitmask of GPIO values to change

4.1.10.7.32. gpio_put_all64

`static void gpio_put_all64 (uint64_t value) [inline], [static]`

Drive all pins simultaneously.

Parameters

`value`	Bitmask of GPIO values to change

4.1.10.7.33. gpio_put_masked

`static void gpio_put_masked (uint32_t mask, uint32_t value) [inline], [static]`

Drive GPIOs high/low depending on parameters.

Parameters

`mask`	Bitmask of GPIO values to change
`value`	Value to set

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.

4.1.10.7.34. gpio_put_masked64

`static void gpio_put_masked64 (uint64_t mask, uint64_t value) [inline], [static]`

Drive GPIOs high/low depending on parameters.

Parameters

`mask`	Bitmask of GPIO values to change
`value`	Value to set

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.

4.1.10.7.35. gpio_put_masked_n

```
static void gpio_put_masked_n (uint n, uint32_t mask, uint32_t value) [inline], [static]
```

Drive GPIOs high/low depending on parameters.

Parameters

`n`	the base GPIO index of the mask to update. n == 0 means 0->31, n == 1 mean 32->63 etc.
`mask`	Bitmask of GPIO values to change
`value`	Value to set

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.

4.1.10.7.36. gpio_remove_raw_irq_handler

```
static void gpio_remove_raw_irq_handler (uint gpio, irq_handler_t handler) [inline], [static]
```

Removes a raw GPIO IRQ handler for the specified GPIO on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see gpio_set_irq_callback), it is possible to add explicit GPIO IRQ handlers which are called independent of the default event callback.

This method removes such a callback, and enables the "default" callback for the specified GPIO.

Parameters

`gpio`	the GPIO number that will now be passed to the default callback for this core
`handler`	the handler to remove from the list of GPIO IRQ handlers for this core

4.1.10.7.37. gpio_remove_raw_irq_handler_masked

```
void gpio_remove_raw_irq_handler_masked (uint32_t gpio_mask, irq_handler_t handler)
```

Removes a raw GPIO IRQ handler for the specified GPIOs on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see gpio_set_irq_callback), it is possible to add explicit GPIO IRQ handlers which are called independent of the default event callback.

This method removes such a callback, and enables the "default" callback for the specified GPIOs.

Parameters

`gpio_mask`	a bit mask of the GPIO numbers that will now be passed to the default callback for this core
`handler`	the handler to remove from the list of GPIO IRQ handlers for this core

4.1.10.7.38. gpio_remove_raw_irq_handler_masked64

```
void gpio_remove_raw_irq_handler_masked64 (uint64_t gpio_mask, irq_handler_t handler)
```

Removes a raw GPIO IRQ handler for the specified GPIOs on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see gpio_set_irq_callback), it is possible to add explicit GPIO IRQ handlers which are called independent of the default event callback.

This method removes such a callback, and enables the "default" callback for the specified GPIOs.

Parameters

`gpio_mask`	a bit mask of the GPIO numbers that will now be passed to the default callback for this core
`handler`	the handler to remove from the list of GPIO IRQ handlers for this core

4.1.10.7.39. gpio_set_dir

`static void gpio_set_dir (uint gpio, bool out) [inline], [static]`

Set a single GPIO direction.

Parameters

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

4.1.10.7.40. gpio_set_dir_all_bits

`static void gpio_set_dir_all_bits (uint32_t values) [inline], [static]`

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

4.1.10.7.41. gpio_set_dir_all_bits64

`static void gpio_set_dir_all_bits64 (uint64_t values) [inline], [static]`

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

4.1.10.7.42. gpio_set_dir_in_masked

`static void gpio_set_dir_in_masked (uint32_t mask) [inline], [static]`

Set a number of GPIOs to input.

Parameters

`mask`	Bitmask of GPIO to set to input

4.1.10.7.43. gpio_set_dir_in_masked64

`static void gpio_set_dir_in_masked64 (uint64_t mask) [inline], [static]`

Set a number of GPIOs to input.

Parameters

`mask` Bitmask of GPIO to set to input

4.1.10.7.44. gpio_set_dir_masked

`static void gpio_set_dir_masked (uint32_t mask, uint32_t value) [inline], [static]`

Set multiple GPIO directions.

Parameters

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

`value` Values to set

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.

4.1.10.7.45. gpio_set_dir_masked64

`static void gpio_set_dir_masked64 (uint64_t mask, uint64_t value) [inline], [static]`

Set multiple GPIO directions.

Parameters

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

`value` Values to set

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.

4.1.10.7.46. gpio_set_dir_out_masked

`static void gpio_set_dir_out_masked (uint32_t mask) [inline], [static]`

Set a number of GPIOs to output.

Switch all GPIOs in "mask" to output

Parameters

`mask` Bitmask of GPIO to set to output

4.1.10.7.47. gpio_set_dir_out_masked64

`static void gpio_set_dir_out_masked64 (uint64_t mask) [inline], [static]`

Set a number of GPIOs to output.

Switch all GPIOs in "mask" to output

Parameters

`mask` Bitmask of GPIO to set to output

4.1.10.7.48. gpio_set_dormant_irq_enabled

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

Enable dormant wake up interrupt for specified GPIO and events.

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

Parameters

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

4.1.10.7.49. gpio_set_drive_strength

```
void gpio_set_drive_strength (uint gpio, enum gpio_drive_strength drive)
```

Set drive strength for a specified GPIO.

See also

gpio_get_drive_strength

Parameters

`gpio`	GPIO number
`drive`	GPIO output drive strength

4.1.10.7.50. gpio_set_function

```
void gpio_set_function (uint gpio, gpio_function_t fn)
```

Select GPIO function.

Parameters

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

4.1.10.7.51. gpio_set_function_masked

```
void gpio_set_function_masked (uint32_t gpio_mask, gpio_function_t fn)
```

Select the function for multiple GPIOs.

See also

gpio_set_function

Parameters

`gpio_mask`	Mask with 1 bit per GPIO number to set the function for
`fn`	Which GPIO function select to use from list gpio_function_t

4.1.10.7.52. gpio_set_function_masked64

```
void gpio_set_function_masked64 (uint64_t gpio_mask, gpio_function_t fn)
```

Select the function for multiple GPIOs.

See also

gpio_set_function

Parameters

`gpio_mask`	Mask with 1 bit per GPIO number to set the function for

`fn`	Which GPIO function select to use from list gpio_function_t

4.1.10.7.53. gpio_set_inover

`void gpio_set_inover (uint gpio, uint value)`

Select GPIO input override.

Parameters

`gpio`	GPIO number
`value`	See gpio_override

4.1.10.7.54. 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

4.1.10.7.55. gpio_set_input_hysteresis_enabled

`void gpio_set_input_hysteresis_enabled (uint gpio, bool enabled)`

Enable/disable GPIO input hysteresis (Schmitt trigger)

Enable or disable the Schmitt trigger hysteresis on a given GPIO. This is enabled on all GPIOs by default. Disabling input hysteresis can lead to inconsistent readings when the input signal has very long rise or fall times, but slightly reduces the GPIO's input delay.

See also

gpio_is_input_hysteresis_enabled

Parameters

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

4.1.10.7.56. gpio_set_irq_callback

`void gpio_set_irq_callback (gpio_irq_callback_t callback)`

Set the generic callback used for GPIO IRQ events for the current core.

This function sets the callback used for all GPIO IRQs on the current core that are not explicitly hooked via gpio_add_raw_irq_handler or other gpio_add_raw_irq_handler_ functions.

This function is called with the GPIO number and event mask for each of the (not explicitly hooked) GPIOs that have events enabled and that are pending (see gpio_get_irq_event_mask).

NOTE

The IO IRQs are independent per-processor. This function affects the processor that calls the function.

Parameters

`callback` default user function to call on GPIO irq. Note only one of these can be set per processor.

4.1.10.7.57. gpio_set_irq_enabled

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

Enable or disable specific interrupt events for specified GPIO.

This function sets which GPIO events cause a GPIO interrupt on the calling core. See gpio_set_irq_callback, gpio_set_irq_enabled_with_callback and gpio_add_raw_irq_handler to set up a GPIO interrupt handler to handle the events.

NOTE

The IO IRQs are independent per-processor. This configures the interrupt events for the processor that calls the function.

Parameters

`gpio` GPIO number

`event_mask` Which events will cause an interrupt

`enabled` Enable or disable flag

Events is a bitmask of the following gpio_irq_level values:

bit	constant	interrupt
0	GPIO_IRQ_LEVEL_LOW	Continuously while level is low
1	GPIO_IRQ_LEVEL_HIGH	Continuously while level is high
2	GPIO_IRQ_EDGE_FALL	On each transition from high to low
3	GPIO_IRQ_EDGE_RISE	On each transition from low to high

which are specified in gpio_irq_level

4.1.10.7.58. gpio_set_irq_enabled_with_callback

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

Convenience function which performs multiple GPIO IRQ related initializations.

This method is a slightly eclectic mix of initialization, that:

- Updates whether the specified events for the specified GPIO causes an interrupt on the calling core based on the enable flag.
- Sets the callback handler for the calling core to callback (or clears the handler if the callback is NULL).
- Enables GPIO IRQs on the current core if enabled is true.

This method is commonly used to perform a one time setup, and following that any additional IRQs/events are enabled via gpio_set_irq_enabled. All GPIOs/events added in this way on the same core share the same callback; for multiple independent handlers for different GPIOs you should use gpio_add_raw_irq_handler and related functions.

This method is equivalent to:

```
gpio_set_irq_enabled(gpio, event_mask, enabled);
gpio_set_irq_callback(callback);
if (enabled) irq_set_enabled(IO_IRQ_BANK0, true);
```

NOTE

The IO IRQs are independent per-processor. This method affects only the processor that calls the function.

Parameters

`gpio`	GPIO number
`event_mask`	Which events will cause an interrupt. See gpio_irq_level for details.
`enabled`	Enable or disable flag
`callback`	user function to call on GPIO irq. if NULL, the callback is removed

4.1.10.7.59. gpio_set_irqover

`void gpio_set_irqover (uint gpio, uint value)`

Set GPIO IRQ override.

Optionally invert a GPIO IRQ signal, or drive it high or low

Parameters

`gpio`	GPIO number
`value`	See gpio_override

4.1.10.7.60. gpio_set_mask

`static void gpio_set_mask (uint32_t mask) [inline], [static]`

Drive high every GPIO appearing in mask.

Parameters

`mask`	Bitmask of GPIO values to set

4.1.10.7.61. gpio_set_mask64

`static void gpio_set_mask64 (uint64_t mask) [inline], [static]`

Drive high every GPIO appearing in mask.

Parameters

`mask`	Bitmask of GPIO values to set

4.1.10.7.62. gpio_set_mask_n

`static void gpio_set_mask_n (uint n, uint32_t mask) [inline], [static]`

Drive high every GPIO appearing in mask.

Parameters

`n`	the base GPIO index of the mask to update. n == 0 means 0->31, n == 1 mean 32->63 etc.
`mask`	Bitmask of 32 GPIO values to set

4.1.10.7.63. gpio_set_oeover

`void gpio_set_oeover (uint gpio, uint value)`

Select GPIO output enable override.

Parameters

`gpio`	GPIO number
`value`	See gpio_override

4.1.10.7.64. gpio_set_outover

`void gpio_set_outover (uint gpio, uint value)`

Set GPIO output override.

Parameters

`gpio`	GPIO number
`value`	See gpio_override

4.1.10.7.65. 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

NOTE

> On the RP2040, setting both pulls enables a "bus keep" function, i.e. a weak pull to whatever is current high/low state of GPIO.

4.1.10.7.66. gpio_set_slew_rate

`void gpio_set_slew_rate (uint gpio, enum gpio_slew_rate slew)`

Set slew rate for a specified GPIO.

See also

gpio_get_slew_rate

Parameters

`gpio`	GPIO number
`slew`	GPIO output slew rate

4.1.10.7.67. gpio_xor_mask

```
static void gpio_xor_mask (uint32_t mask) [inline], [static]
```

Toggle every GPIO appearing in mask.

Parameters

`mask`	Bitmask of GPIO values to toggle

4.1.10.7.68. gpio_xor_mask64

```
static void gpio_xor_mask64 (uint64_t mask) [inline], [static]
```

Toggle every GPIO appearing in mask.

Parameters

`mask`	Bitmask of GPIO values to toggle

4.1.10.7.69. gpio_xor_mask_n

```
static void gpio_xor_mask_n (uint n, uint32_t mask) [inline], [static]
```

Toggle every GPIO appearing in mask.

Parameters

`n`	the base GPIO index of the mask to update. n == 0 means 0->31, n == 1 mean 32->63 etc.
`mask`	Bitmask of 32 GPIO values to toggle

4.1.11. hardware_hazard3 RP2350

Accessors for Hazard3-specific RISC-V CSRs, and intrinsics for Hazard3 custom instructions.

4.1.11.1. Detailed Description

Intrinsics and asm macros for Hazard3 custom instructions.

Sets macros for supported Hazard3 custom extensions (features) based on PICO_PLATFORM macros.

The implementation of these intrinsics depends on the feature macros defined in hardware/hazard3/features.h. When the relevant feature is not present, the intrinsics fall back on an RV32I equivalent if possible.

4.1.12. hardware_i2c

I2C Controller API.

4.1.12.1. Detailed Description

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 7-bit 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. The first byte in the data transfer always contains the 7-bit address and a read/write bit in the LSB position. This API takes care of toggling the read/write bit. After this, 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
// 00 . . . . . . . . . . . . . . . .
// 10 . . @ . . . . . . . . . . . . .
// 20 . . . . . . . . . . . . . . . .
// 30 . . . . @ . . . . . . . . . . .
// 40 . . . . . . . . . . . . . . . .
// 50 . . . . . . . . . . . . . . . .
// 60 . . . . . . . . . . . . . . . .
// 70 . . . . . . . . . . . . . . . .
// E.g. if addresses 0x12 and 0x34 were acknowledged.

#include <stdio.h>
#include "pico/stdlib.h"
#include "pico/binary_info.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();
#if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) || !defined(PICO_DEFAULT_I2C_SCL_PIN)
#warning i2c/bus_scan example requires a board with I2C pins
    puts("Default I2C pins were not defined");
#else
    // This example will use I2C0 on the default SDA and SCL pins (GP4, GP5 on a Pico)
    i2c_init(i2c_default, 100 * 1000);
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
    gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);
    // Make the I2C pins available to picotool
    bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C));

    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 ret;
        uint8_t rxdata;
        if (reserved_addr(addr))
            ret = PICO_ERROR_GENERIC;
        else
            ret = i2c_read_blocking(i2c_default, addr, &rxdata, 1, false);

        printf(ret < 0 ? "." : "@");
        printf(addr % 16 == 15 ? "\n" : "  ");
    }
    printf("Done.\n");
    return 0;
#endif
}
```

4.1.12.2. Macros

- `#define I2C_NUM(i2c)`
- `#define I2C_INSTANCE(num)`
- `#define I2C_DREQ_NUM(i2c, is_tx)`

4.1.12.3. Functions

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

Initialise the I2C HW block.

`void i2c_deinit (i2c_inst_t *i2c)`

Disable the I2C HW block.

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

Set I2C baudrate.

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

Set I2C port to slave mode.

`static uint i2c_get_index (i2c_inst_t *i2c)`

Convert I2C instance to hardware instance number.

`static i2c_hw_t * i2c_get_hw (i2c_inst_t *i2c)`

Return pointer to structure containing i2c hardware registers.

`static i2c_inst_t * i2c_get_instance (uint num)`

Convert I2C hardware instance number to I2C instance.

`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.

`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.

`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.

`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.

`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.

`int i2c_write_burst_blocking (i2c_inst_t *i2c, uint8_t addr, const uint8_t *src, size_t len)`

Attempt to write specified number of bytes to address, blocking in burst mode.

`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.

`int i2c_read_burst_blocking (i2c_inst_t *i2c, uint8_t addr, uint8_t *dst, size_t len)`

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

`static size_t i2c_get_write_available (i2c_inst_t *i2c)`

Determine non-blocking write space available.

`static size_t i2c_get_read_available (i2c_inst_t *i2c)`

Determine number of bytes received.

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

Write direct to TX FIFO.

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

Read direct from RX FIFO.

`static uint8_t i2c_read_byte_raw (i2c_inst_t *i2c)`

Pop a byte from I2C Rx FIFO.

`static void i2c_write_byte_raw (i2c_inst_t *i2c, uint8_t value)`

Push a byte into I2C Tx FIFO.

`static uint i2c_get_dreq (i2c_inst_t *i2c, bool is_tx)`

Return the DREQ to use for pacing transfers to/from a particular I2C instance.

4.1.12.3.1. i2c0_inst

`i2c_inst_t i2c0_inst`

The I2C identifiers for use in I2C functions.

e.g. i2c_init(i2c0, 48000)

4.1.12.4. Macro Definition Documentation

4.1.12.4.1. I2C_NUM

`#define I2C_NUM(i2c)`

Returns the I2C number for a I2C instance.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.12.4.2. I2C_INSTANCE

```
#define I2C_INSTANCE(num)
```

Returns the I2C instance with the given I2C number.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.12.4.3. I2C_DREQ_NUM

```
#define I2C_DREQ_NUM(i2c, is_tx)
```

Returns the dreq_num_t used for pacing DMA transfers to or from this I2C instance. If is_tx is true, then it is for transfers to the I2C instance else for transfers from the I2C instance.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.12.5. Function Documentation

4.1.12.5.1. i2c_deinit

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

Disable the I2C HW block.

Parameters

`i2c`	Either i2c0 or i2c1

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

4.1.12.5.2. i2c_get_dreq

```
static uint i2c_get_dreq (i2c_inst_t * i2c, bool is_tx) [inline], [static]
```

Return the DREQ to use for pacing transfers to/from a particular I2C instance.

Parameters

`i2c`	Either i2c0 or i2c1
`is_tx`	true for sending data to the I2C instance, false for receiving data from the I2C instance

4.1.12.5.3. i2c_get_hw

```
static i2c_hw_t * i2c_get_hw (i2c_inst_t * i2c) [inline], [static]
```

Return pointer to structure containing i2c hardware registers.

Parameters

`i2c`	I2C instance

Returns

pointer to i2c_hw_t

4.1.12.5.4. i2c_get_index

`static uint i2c_get_index (i2c_inst_t * i2c) [inline], [static]`

Convert I2C instance to hardware instance number.

Parameters

`i2c`	I2C instance

Returns

Number of I2C, 0 or 1.

4.1.12.5.5. i2c_get_instance

`static i2c_inst_t * i2c_get_instance (uint num) [inline], [static]`

Convert I2C hardware instance number to I2C instance.

Parameters

`num`	Number of I2C, 0 or 1

Returns

I2C hardware instance

4.1.12.5.6. i2c_get_read_available

`static size_t i2c_get_read_available (i2c_inst_t * i2c) [inline], [static]`

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.

4.1.12.5.7. i2c_get_write_available

`static size_t i2c_get_write_available (i2c_inst_t * i2c) [inline], [static]`

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.

4.1.12.5.8. 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 actual rate set is returned

Parameters

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

Returns

Actual set baudrate

4.1.12.5.9. 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`	7-bit 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 or no device present.

4.1.12.5.10. 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`	7-bit 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 occurred.

4.1.12.5.11. i2c_read_burst_blocking

```
int i2c_read_burst_blocking (i2c_inst_t * i2c, uint8_t addr, uint8_t * dst, size_t len)
```

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

This version of the function will not issue a stop and will not restart on the next read. This allows you to read consecutive bytes of data without having to resend a stop bit and (for example) without having to send address byte(s) repeatedly

Parameters

`i2c`	Either i2c0 or i2c1
`addr`	7-bit address of device to read from
`dst`	Pointer to buffer to receive data
`len`	Length of data in bytes to receive

Returns

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

4.1.12.5.12. i2c_read_byte_raw

`static uint8_t i2c_read_byte_raw (i2c_inst_t * i2c) [inline], [static]`

Pop a byte from I2C Rx FIFO.

This function is non-blocking and assumes the Rx FIFO isn't empty.

Parameters

`i2c`	I2C instance.

Returns

uint8_t Byte value.

4.1.12.5.13. i2c_read_raw_blocking

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

Read direct from RX FIFO.

Parameters

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

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

4.1.12.5.14. 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) [inline], [static]`

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

Parameters

`i2c`	Either i2c0 or i2c1
`addr`	7-bit 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 occurred.

4.1.12.5.15. 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

4.1.12.5.16. 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

4.1.12.5.17. 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`	7-bit 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.

4.1.12.5.18. 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`	7-bit 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 occurred.

4.1.12.5.19. i2c_write_burst_blocking

`int i2c_write_burst_blocking (i2c_inst_t * i2c, uint8_t addr, const uint8_t * src, size_t len)`

Attempt to write specified number of bytes to address, blocking in burst mode.

This version of the function will not issue a stop and will not restart on the next write. This allows you to write consecutive bytes of data without having to resend a stop bit and (for example) without having to send address byte(s) repeatedly

Parameters

`i2c`	Either i2c0 or i2c1
`addr`	7-bit address of device to read from
`dst`	Pointer to buffer to receive data
`len`	Length of data in bytes to receive

Returns

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

4.1.12.5.20. i2c_write_byte_raw

`static void i2c_write_byte_raw (i2c_inst_t * i2c, uint8_t value) [inline], [static]`

Push a byte into I2C Tx FIFO.

This function is non-blocking and assumes the Tx FIFO isn't full.

Parameters

`i2c`	I2C instance.
`value`	Byte value.

4.1.12.5.21. i2c_write_raw_blocking

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

Write direct to TX FIFO.

Parameters

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

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

4.1.12.5.22. 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) [inline], [static]`

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

Parameters

`i2c`	Either i2c0 or i2c1
`addr`	7-bit 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 occurred.

4.1.13. hardware_interp

Hardware Interpolator API.

4.1.13.1. Detailed Description

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 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 appropriate RP-series microcontroller datasheet for more information on the HW interpolators and how they work.

4.1.13.2. Modules

interp_config

Interpolator configuration.

4.1.13.3. Functions

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

Claim the interpolator lane specified.

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

Claim the interpolator lanes specified in the mask.

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

Release a previously claimed interpolator lane.

`bool interp_lane_is_claimed (interp_hw_t *interp, uint lane)`

Determine if an interpolator lane is claimed.

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

Release previously claimed interpolator lanes.

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

Directly set the force bits on a specified lane.

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

Save the specified interpolator state.

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

Restore an interpolator state.

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

Sets the interpolator base register by lane.

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

Gets the content of interpolator base register by lane.

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

Sets the interpolator base registers simultaneously.

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

Sets the interpolator accumulator register by lane.

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

Gets the content of the interpolator accumulator register by lane.

`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.

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

Read lane 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.

`static uint32_t interp_peek_full_result (interp_hw_t *interp)`

Read lane result.

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

Add to accumulator.

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

Get raw lane value.

4.1.13.4. Function Documentation

4.1.13.4.1. interp_add_accumulator

`static void interp_add_accumulator (interp_hw_t * interp, uint lane, uint32_t val) [inline], [static]`

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

4.1.13.4.2. 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 specified 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.

4.1.13.4.3. 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)

4.1.13.4.4. interp_get_accumulator

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

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

4.1.13.4.5. interp_get_base

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

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

4.1.13.4.6. interp_get_raw

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

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

4.1.13.4.7. interp_lane_is_claimed

`bool interp_lane_is_claimed (interp_hw_t * interp, uint lane)`

Determine if an interpolator lane is claimed.

Parameters

`interp` Interpolator whose lane to check

`lane` The lane number, 0 or 1

Returns

true if claimed, false otherwise

See also

interp_claim_lane

interp_claim_lane_mask

4.1.13.4.8. interp_peek_full_result

`static uint32_t interp_peek_full_result (interp_hw_t * interp) [inline], [static]` Read lane result.

Parameters

`interp` Interpolator instance, interp0 or interp1.

Returns

The content of the FULL register

4.1.13.4.9. interp_peek_lane_result

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

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

4.1.13.4.10. interp_pop_full_result

`static uint32_t interp_pop_full_result (interp_hw_t * interp) [inline], [static]`

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

4.1.13.4.11. interp_pop_lane_result

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

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

4.1.13.4.12. 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

4.1.13.4.13. 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

4.1.13.4.14. interp_set_accumulator

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

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

4.1.13.4.15. interp_set_base

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

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

4.1.13.4.16. interp_set_base_both

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

Sets the interpolator base registers simultaneously.

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

4.1.13.4.17. interp_set_force_bits

`static void interp_set_force_bits (interp_hw_t * interp, uint lane, uint bits) [inline], [static]`

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)

4.1.13.4.18. 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

4.1.13.4.19. interp_unclaim_lane_mask

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

Release previously claimed interpolator lanes.

See also

interp_claim_lane_mask

Parameters

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

4.1.13.5. interp_config

Interpolator configuration.

4.1.13.5.1. Detailed Description

Each interpolator needs to be configured, these functions provide handy helpers to set up configuration structures.

4.1.13.5.2. Functions

`static void interp_config_set_shift (interp_config *c, uint shift)`

Set the interpolator shift value.

`static void interp_config_set_mask (interp_config *c, uint mask_lsb, uint mask_msb)`

Set the interpolator mask range.

`static void interp_config_set_cross_input (interp_config *c, bool cross_input)`

Enable cross input.

`static void interp_config_set_cross_result (interp_config *c, bool cross_result)`

Enable cross results.

`static void interp_config_set_signed (interp_config *c, bool _signed)`

Set sign extension.

`static void interp_config_set_add_raw (interp_config *c, bool add_raw)`

Set raw add option.

`static void interp_config_set_blend (interp_config *c, bool blend)`

Set blend mode.

`static void interp_config_set_clamp (interp_config *c, bool clamp)`

Set interpolator clamp mode (Interpolator 1 only)

`static void interp_config_set_force_bits (interp_config *c, uint bits)`

Set interpolator Force bits.

`static interp_config interp_default_config (void)`

Get a default configuration.

`static void interp_set_config (interp_hw_t *interp, uint lane, interp_config *config)`

Send configuration to a lane.

4.1.13.5.3. Function Documentation

interp_config_set_add_raw

`static void interp_config_set_add_raw (interp_config * c, bool add_raw) [inline], [static]`

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.

interp_config_set_blend

`static void interp_config_set_blend (interp_config * c, bool blend) [inline], [static]`

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.

interp_config_set_clamp

`static void interp_config_set_clamp (interp_config * c, bool clamp) [inline], [static]`

Set interpolator clamp mode (Interpolator 1 only)

Only present on INTERP1 on each core. If CLAMP mode is enabled:

- LANE0 result is a shifted and masked ACCUM0, clamped by a lower bound of BASE0 and an upper bound of BASE1.
- Signedness of these comparisons is determined by LANE0_CTRL_SIGNED

Parameters

`c` Pointer to interpolation config

`clamp` Set true to enable clamp mode

interp_config_set_cross_input

`static void interp_config_set_cross_input (interp_config * c, bool cross_input) [inline], [static]`

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_set_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.

interp_config_set_cross_result

`static void interp_config_set_cross_result (interp_config * c, bool cross_result) [inline], [static]`

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

interp_config_set_force_bits

`static void interp_config_set_force_bits (interp_config * c, uint bits) [inline], [static]`

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)

interp_config_set_mask

`static void interp_config_set_mask (interp_config * c, uint mask_lsb, uint mask_msb) [inline], [static]`

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

interp_config_set_shift

`static void interp_config_set_shift (interp_config * c, uint shift) [inline], [static]`

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

interp_config_set_signed

`static void interp_config_set_signed (interp_config * c, bool _signed) [inline], [static]`

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

interp_default_config

`static interp_config interp_default_config (void) [inline], [static]`

Get a default configuration.

Returns

A default interpolation configuration

interp_set_config

`static void interp_set_config (interp_hw_t * interp, uint lane, interp_config * config) [inline], [static]`

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

4.1.14. hardware_irq

Hardware interrupt handling API.

4.1.14.1. Detailed Description

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.

> **NOTE**
>
> 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).

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

4.1.14.1.1. Interrupt Numbers

A set of defines is available (intctrl.h) with these names to avoid using the numbers directly.

On RP2040 the interrupt numbers are as follows:

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

IRQ	Interrupt Source
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

On RP2350 the interrupt numbers are as follows:

IRQ	Interrupt Source
0	TIMER0_IRQ_0
1	TIMER0_IRQ_1
2	TIMER0_IRQ_2
3	TIMER0_IRQ_3
4	TIMER1_IRQ_0
5	TIMER1_IRQ_1
6	TIMER1_IRQ_2
7	TIMER1_IRQ_3
8	PWM_IRQ_WRAP_0
9	PWM_IRQ_WRAP_1
10	DMA_IRQ_0
11	DMA_IRQ_1
12	DMA_IRQ_2
13	DMA_IRQ_3
14	USBCTRL_IRQ
15	PIO0_IRQ_0
16	PIO0_IRQ_1
17	PIO1_IRQ_0

IRQ	Interrupt Source
18	PIO1_IRQ_1
19	PIO2_IRQ_0
20	PIO2_IRQ_1
21	IO_IRQ_BANK0
22	IO_IRQ_BANK0_NS
23	IO_IRQ_QSPI
24	IO_IRQ_QSPI_NS
25	SIO_IRQ_FIFO
26	SIO_IRQ_BELL
27	SIO_IRQ_FIFO_NS
28	SIO_IRQ_BELL_NS
29	SIO_IRQ_MTIMECMP
30	CLOCKS_IRQ
31	SPI0_IRQ
32	SPI1_IRQ
33	UART0_IRQ
34	UART1_IRQ
35	ADC_IRQ_FIFO
36	I2C0_IRQ
37	I2C1_IRQ
38	OTP_IRQ
39	TRNG_IRQ
40	PROC0_IRQ_CTI
41	PROC1_IRQ_CTI
42	PLL_SYS_IRQ
43	PLL_USB_IRQ
44	POWMAN_IRQ_POW
45	POWMAN_IRQ_TIMER
46	SPAREIRQ_IRQ_0
47	SPAREIRQ_IRQ_1
48	SPAREIRQ_IRQ_2
49	SPAREIRQ_IRQ_3
50	SPAREIRQ_IRQ_4
51	SPAREIRQ_IRQ_5

4.1.14.2. Typedefs

`typedef enum irq_num_rp2350 irq_num_t` RP2350

Interrupt numbers on RP2350 (used as typedef irq_num_t)

`typedef enum irq_num_rp2040 irq_num_t` RP2040

Interrupt numbers on RP2040 (used as typedef irq_num_t)

`typedef void(* irq_handler_t)(void)`

Interrupt handler function type.

4.1.14.3. Enumerations

`enum irq_num_rp2350 { TIMER0_IRQ_0 = 0, TIMER0_IRQ_1 = 1, TIMER0_IRQ_2 = 2, TIMER0_IRQ_3 = 3, TIMER1_IRQ_0 = 4, TIMER1_IRQ_1 = 5, TIMER1_IRQ_2 = 6, TIMER1_IRQ_3 = 7, PWM_IRQ_WRAP_0 = 8, PWM_IRQ_WRAP_1 = 9, DMA_IRQ_0 = 10, DMA_IRQ_1 = 11, DMA_IRQ_2 = 12, DMA_IRQ_3 = 13, USBCTRL_IRQ = 14, PIO0_IRQ_0 = 15, PIO0_IRQ_1 = 16, PIO1_IRQ_0 = 17, PIO1_IRQ_1 = 18, PIO2_IRQ_0 = 19, PIO2_IRQ_1 = 20, IO_IRQ_BANK0 = 21, IO_IRQ_BANK0_NS = 22, IO_IRQ_QSPI = 23, IO_IRQ_QSPI_NS = 24, SIO_IRQ_FIFO = 25, SIO_IRQ_BELL = 26, SIO_IRQ_FIFO_NS = 27, SIO_IRQ_BELL_NS = 28, SIO_IRQ_MTIMECMP = 29, CLOCKS_IRQ = 30, SPI0_IRQ = 31, SPI1_IRQ = 32, UART0_IRQ = 33, UART1_IRQ = 34, ADC_IRQ_FIFO = 35, I2C0_IRQ = 36, I2C1_IRQ = 37, OTP_IRQ = 38, TRNG_IRQ = 39, PROC0_IRQ_CTI = 40, PROC1_IRQ_CTI = 41, PLL_SYS_IRQ = 42, PLL_USB_IRQ = 43, POWMAN_IRQ_POW = 44, POWMAN_IRQ_TIMER = 45, SPARE_IRQ_0 = 46, SPARE_IRQ_1 = 47, SPARE_IRQ_2 = 48, SPARE_IRQ_3 = 49, SPARE_IRQ_4 = 50, SPARE_IRQ_5 = 51, IRQ_COUNT }` RP2350

Interrupt numbers on RP2350 (used as typedef irq_num_t)

`enum irq_num_rp2040 { TIMER_IRQ_0 = 0, TIMER_IRQ_1 = 1, TIMER_IRQ_2 = 2, TIMER_IRQ_3 = 3, PWM_IRQ_WRAP = 4, USBCTRL_IRQ = 5, XIP_IRQ = 6, PIO0_IRQ_0 = 7, PIO0_IRQ_1 = 8, PIO1_IRQ_0 = 9, PIO1_IRQ_1 = 10, DMA_IRQ_0 = 11, DMA_IRQ_1 = 12, IO_IRQ_BANK0 = 13, IO_IRQ_QSPI = 14, SIO_IRQ_PROC0 = 15, SIO_IRQ_PROC1 = 16, CLOCKS_IRQ = 17, SPI0_IRQ = 18, SPI1_IRQ = 19, UART0_IRQ = 20, UART1_IRQ = 21, ADC_IRQ_FIFO = 22, I2C0_IRQ = 23, I2C1_IRQ = 24, RTC_IRQ = 25, IRQ_COUNT }` RP2040

Interrupt numbers on RP2040 (used as typedef irq_num_t)

4.1.14.4. Functions

`void irq_set_priority (uint num, uint8_t hardware_priority)`

Set specified interrupt's priority.

`uint irq_get_priority (uint num)`

Get specified interrupt's priority.

`void irq_set_enabled (uint num, bool enabled)`

Enable or disable a specific interrupt on the executing core.

`bool irq_is_enabled (uint num)`

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

`void irq_set_mask_enabled (uint32_t mask, bool enabled)`

Enable/disable multiple interrupts on the executing core.

`void irq_set_mask_n_enabled (uint n, uint32_t mask, bool enabled)`

Enable/disable multiple interrupts on the executing core.

`void irq_set_exclusive_handler (uint num, irq_handler_t handler)`

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

`irq_handler_t irq_get_exclusive_handler (uint num)`

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

`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.

`void irq_remove_handler (uint num, irq_handler_t handler)`

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

`bool irq_has_handler (uint num)`

Determine if there is an installed IRQ handler for the given interrupt number.

`bool irq_has_shared_handler (uint num)`

Determine if the current IRQ andler for the given interrupt number is shared.

`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.

`static void irq_clear (uint int_num)`

Clear a specific interrupt on the executing core.

`void irq_set_pending (uint num)`

Force an interrupt to be pending on the executing core.

`void user_irq_claim (uint irq_num)`

Claim ownership of a user IRQ on the calling core.

`void user_irq_unclaim (uint irq_num)`

Mark a user IRQ as no longer used on the calling core.

`int user_irq_claim_unused (bool required)`

Claim ownership of a free user IRQ on the calling core.

4.1.14.5. Typedef Documentation

4.1.14.5.1. irq_num_t RP2350

```
typedef enum irq_num_rp2350 irq_num_t
```

Interrupt numbers on RP2350 (used as typedef irq_num_t)

4.1.14.5.2. irq_num_t RP2040

```
typedef enum irq_num_rp2040 irq_num_t
```

Interrupt numbers on RP2040 (used as typedef irq_num_t)

4.1.14.5.3. irq_handler_t

```
typedef void(* irq_handler_t) (void)
```

Interrupt handler function type.

All interrupts handlers should be of this type, and follow normal ARM EABI register saving conventions

4.1.14.6. Enumeration Type Documentation

4.1.14.6.1. irq_num_rp2350 RP2350

`enum irq_num_rp2350`

Interrupt numbers on RP2350 (used as typedef irq_num_t)

Table 22. Enumerator

TIMER0_IRQ_0	Select TIMER0's IRQ 0 output.
TIMER0_IRQ_1	Select TIMER0's IRQ 1 output.
TIMER0_IRQ_2	Select TIMER0's IRQ 2 output.
TIMER0_IRQ_3	Select TIMER0's IRQ 3 output.
TIMER1_IRQ_0	Select TIMER1's IRQ 0 output.
TIMER1_IRQ_1	Select TIMER1's IRQ 1 output.
TIMER1_IRQ_2	Select TIMER1's IRQ 2 output.
TIMER1_IRQ_3	Select TIMER1's IRQ 3 output.
PWM_IRQ_WRAP_0	Select PWM's IRQ_WRAP 0 output.
PWM_IRQ_WRAP_1	Select PWM's IRQ_WRAP 1 output.
DMA_IRQ_0	Select DMA's IRQ 0 output.
DMA_IRQ_1	Select DMA's IRQ 1 output.
DMA_IRQ_2	Select DMA's IRQ 2 output.
DMA_IRQ_3	Select DMA's IRQ 3 output.
USBCTRL_IRQ	Select USBCTRL's IRQ output.
PIO0_IRQ_0	Select PIO0's IRQ 0 output.
PIO0_IRQ_1	Select PIO0's IRQ 1 output.
PIO1_IRQ_0	Select PIO1's IRQ 0 output.
PIO1_IRQ_1	Select PIO1's IRQ 1 output.
PIO2_IRQ_0	Select PIO2's IRQ 0 output.
PIO2_IRQ_1	Select PIO2's IRQ 1 output.
IO_IRQ_BANK0	Select IO_BANK0's IRQ output.
IO_IRQ_BANK0_NS	Select IO_BANK0_NS's IRQ output.
IO_IRQ_QSPI	Select IO_QSPI's IRQ output.
IO_IRQ_QSPI_NS	Select IO_QSPI_NS's IRQ output.
SIO_IRQ_FIFO	Select SIO's IRQ_FIFO output.
SIO_IRQ_BELL	Select SIO's IRQ_BELL output.
SIO_IRQ_FIFO_NS	Select SIO_NS's IRQ_FIFO output.
SIO_IRQ_BELL_NS	Select SIO_NS's IRQ_BELL output.
SIO_IRQ_MTIMECMP	Select SIO_IRQ_MTIMECMP's IRQ output.
CLOCKS_IRQ	Select CLOCKS's IRQ output.
SPI0_IRQ	Select SPI0's IRQ output.

SPI1_IRQ	Select SPI1's IRQ output.
UART0_IRQ	Select UART0's IRQ output.
UART1_IRQ	Select UART1's IRQ output.
ADC_IRQ_FIFO	Select ADC's IRQ_FIFO output.
I2C0_IRQ	Select I2C0's IRQ output.
I2C1_IRQ	Select I2C1's IRQ output.
OTP_IRQ	Select OTP's IRQ output.
TRNG_IRQ	Select TRNG's IRQ output.
PROC0_IRQ_CTI	Select PROC0's IRQ_CTI output.
PROC1_IRQ_CTI	Select PROC1's IRQ_CTI output.
PLL_SYS_IRQ	Select PLL_SYS's IRQ output.
PLL_USB_IRQ	Select PLL_USB's IRQ output.
POWMAN_IRQ_POW	Select POWMAN's IRQ_POW output.
POWMAN_IRQ_TIMER	Select POWMAN's IRQ_TIMER output.
SPARE_IRQ_0	Select SPARE IRQ 0.
SPARE_IRQ_1	Select SPARE IRQ 1.
SPARE_IRQ_2	Select SPARE IRQ 2.
SPARE_IRQ_3	Select SPARE IRQ 3.
SPARE_IRQ_4	Select SPARE IRQ 4.
SPARE_IRQ_5	Select SPARE IRQ 5.

4.1.14.6.2. irq_num_rp2040 RP2040

`enum irq_num_rp2040`

Interrupt numbers on RP2040 (used as typedef irq_num_t)

Table 23. Enumerator

TIMER_IRQ_0	Select TIMER's IRQ 0 output.
TIMER_IRQ_1	Select TIMER's IRQ 1 output.
TIMER_IRQ_2	Select TIMER's IRQ 2 output.
TIMER_IRQ_3	Select TIMER's IRQ 3 output.
PWM_IRQ_WRAP	Select PWM's IRQ_WRAP output.
USBCTRL_IRQ	Select USBCTRL's IRQ output.
XIP_IRQ	Select XIP's IRQ output.
PIO0_IRQ_0	Select PIO0's IRQ 0 output.
PIO0_IRQ_1	Select PIO0's IRQ 1 output.
PIO1_IRQ_0	Select PIO1's IRQ 0 output.
PIO1_IRQ_1	Select PIO1's IRQ 1 output.
DMA_IRQ_0	Select DMA's IRQ 0 output.

DMA_IRQ_1	Select DMA's IRQ 1 output.
IO_IRQ_BANK0	Select IO_BANK0's IRQ output.
IO_IRQ_QSPI	Select IO_QSPI's IRQ output.
SIO_IRQ_PROC0	Select SIO_PROC0's IRQ output.
SIO_IRQ_PROC1	Select SIO_PROC1's IRQ output.
CLOCKS_IRQ	Select CLOCKS's IRQ output.
SPI0_IRQ	Select SPI0's IRQ output.
SPI1_IRQ	Select SPI1's IRQ output.
UART0_IRQ	Select UART0's IRQ output.
UART1_IRQ	Select UART1's IRQ output.
ADC_IRQ_FIFO	Select ADC's IRQ_FIFO output.
I2C0_IRQ	Select I2C0's IRQ output.
I2C1_IRQ	Select I2C1's IRQ output.
RTC_IRQ	Select RTC's IRQ output.

4.1.14.7. Function Documentation

4.1.14.7.1. 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.

> **NOTE**
>
> By default, the SDK uses a single shared vector table for both cores, and the currently installed IRQ handlers are effectively a linked list starting a vector table entry for a particular IRQ number. Therefore, this method (when using the same vector table for both cores) add the same interrupt handler for both cores.

On RP2040 this was never really a cause of any confusion, because it rarely made sense to enable the same interrupt number in the NVIC on both cores (see irq_set_enabled()), because the interrupt would then fire on both cores, and the interrupt handlers would race.

The problem *does* exist however when dealing with interrupts which are independent on the two cores.

This includes:

- the core local "spare" IRQs
- on RP2350 the SIO FIFO IRQ which is now the same irq number for both cores (vs RP2040 where it was two)

In the cases where you want to enable the same IRQ on both cores, and both cores are sharing the same vector table, you should install the IRQ handler once - it will be used on both cores - and check the core number (via get_core_num())

on each core.

NOTE

It is not thread safe to add/remove/handle IRQs for the same irq number in the same vector table from both cores concurrently.

NOTE

The SDK has a PICO_VTABLE_PER_CORE define indicating whether the two vector tables are separate, however as of version 2.1.1 the user cannot set this value, and expect the vector table duplication to be handled for them. This functionality will be added in a future SDK version

Parameters

`num`	Interrupt number Interrupt Numbers
`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.

NOTE

The order_priority uses *higher* values for higher priorities which is the *opposite* of the CPU interrupt priorities passed to irq_set_priority() which use lower values for higher priorities.

See also

irq_set_exclusive_handler()

4.1.14.7.2. irq_clear

`static void irq_clear (uint int_num) [inline], [static]`

Clear a specific interrupt on the executing core.

This method is only useful for "software" IRQs that are not connected to hardware (e.g. IRQs 26-31 on RP2040) as the the NVIC always reflects the current state of the IRQ state of the hardware for hardware IRQs, and clearing of the IRQ state of the hardware is performed via the hardware's registers instead.

Parameters

`int_num`	Interrupt number Interrupt Numbers

4.1.14.7.3. irq_get_exclusive_handler

`irq_handler_t irq_get_exclusive_handler (uint num)`

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

This method will return an exclusive IRQ handler set on this core by irq_set_exclusive_handler if there is one.

Parameters

`num`	Interrupt number Interrupt Numbers

See also

irq_set_exclusive_handler()

Returns

handler The handler if an exclusive handler is set for the IRQ, NULL if no handler is set or shared/shareable handlers are installed

4.1.14.7.4. irq_get_priority

`uint irq_get_priority (uint num)`

Get specified interrupt's priority.

Numerically-lower values indicate a higher priority. Hardware priorities range from 0 (highest priority) to 255 (lowest priority). 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

Only the top 2 bits are significant on ARM Cortex-M0+ on RP2040.

Only the top 4 bits are significant on ARM Cortex-M33 or Hazard3 (RISC-V) on RP2350. Note that this API uses the same (inverted) ordering as ARM on RISC-V

Parameters

`num` Interrupt number Interrupt Numbers

Returns

the IRQ priority

4.1.14.7.5. 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

4.1.14.7.6. irq_has_handler

`bool irq_has_handler (uint num)`

Determine if there is an installed IRQ handler for the given interrupt number.

See irq_set_exclusive_handler() for discussion on the scope of handlers when using both cores.

Parameters

`num` Interrupt number Interrupt Numbers

Returns

true if the specified IRQ has a handler

4.1.14.7.7. irq_has_shared_handler

`bool irq_has_shared_handler (uint num)`

Determine if the current IRQ andler for the given interrupt number is shared.

See irq_set_exclusive_handler() for discussion on the scope of handlers when using both cores.

Parameters

`num`	Interrupt number Interrupt Numbers

Returns

true if the specified IRQ has a shared handler

4.1.14.7.8. 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

4.1.14.7.9. 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

> **NOTE**
>
> This method may *only* be called from user (non IRQ code) or from within the handler itself (i.e. an IRQ handler may remove itself as part of handling the IRQ). Attempts to call from another IRQ will cause an assertion.

Parameters

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

See also

irq_set_exclusive_handler()

irq_add_shared_handler()

4.1.14.7.10. 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

4.1.14.7.11. 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 be 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.

> **NOTE**
>
> By default, the SDK uses a single shared vector table for both cores, and the currently installed IRQ handlers are effectively a linked list starting a vector table entry for a particular IRQ number. Therefore, this method (when using the same vector table for both cores) sets the same interrupt handler for both cores.

On RP2040 this was never really a cause of any confusion, because it rarely made sense to enable the same interrupt number in the NVIC on both cores (see irq_set_enabled()), because the interrupt would then fire on both cores, and the interrupt handlers would race.

The problem *does* exist however when dealing with interrupts which are independent on the two cores.

This includes:

- the core local "spare" IRQs
- on RP2350 the SIO FIFO IRQ which is now the same irq number for both cores (vs RP2040 where it was two)

In the cases where you want to enable the same IRQ on both cores, and both cores are sharing the same vector table, you should install the IRQ handler once - it will be used on both cores - and check the core number (via get_core_num()) on each core.

> **NOTE**
>
> It is not thread safe to add/remove/handle IRQs for the same irq number in the same vector table from both cores concurrently.

> **NOTE**
>
> The SDK has a PICO_VTABLE_PER_CORE define indicating whether the two vector tables are separate, however as of version 2.1.1 the user cannot set this value, and expect the vector table duplication to be handled for them. This functionality will be added in a future SDK version

Parameters

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

See also

irq_add_shared_handler()

4.1.14.7.12. 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 Interrupt Numbers
`enabled`	true to enable the interrupts, false to disable them.

4.1.14.7.13. irq_set_mask_n_enabled

```
void irq_set_mask_n_enabled (uint n, uint32_t mask, bool enabled)
```

Enable/disable multiple interrupts on the executing core.

Parameters

`n`	the index of the mask to update. n == 0 means 0->31, n == 1 mean 32->63 etc.
`mask`	32-bit mask with one bits set for the interrupts to enable/disable Interrupt Numbers
`enabled`	true to enable the interrupts, false to disable them.

4.1.14.7.14. irq_set_pending

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

Force an interrupt to be pending on the executing core.

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

Parameters

`num`	Interrupt number Interrupt Numbers

4.1.14.7.15. irq_set_priority

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

Set specified interrupt's priority.

Parameters

`num`	Interrupt number Interrupt Numbers
`hardware_priority`	Priority to set. Numerically-lower values indicate a higher priority. Hardware priorities range from 0 (highest priority) to 255 (lowest priority). 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

Only the top 2 bits are significant on ARM Cortex-M0+ on RP2040.

Only the top 4 bits are significant on ARM Cortex-M33 or Hazard3 (RISC-V) on RP2350. Note that this API uses the same (inverted) ordering as ARM on RISC-V

4.1.14.7.16. user_irq_claim

```
void user_irq_claim (uint irq_num)
```

Claim ownership of a user IRQ on the calling core.

User IRQs starting from FIRST_USER_IRQ are not connected to any hardware, but can be triggered by irq_set_pending.

NOTE

User IRQs are a core local feature; they cannot be used to communicate between cores. Therefore all functions dealing with Uer IRQs affect only the calling core

This method explicitly claims ownership of a user IRQ, so other code can know it is being used.

Parameters

`irq_num`	the user IRQ to claim

4.1.14.7.17. user_irq_claim_unused

`int user_irq_claim_unused (bool required)`

Claim ownership of a free user IRQ on the calling core.

User IRQs starting from FIRST_USER_IRQ are not connected to any hardware, but can be triggered by irq_set_pending.

NOTE

User IRQs are a core local feature; they cannot be used to communicate between cores. Therefore all functions dealing with Uer IRQs affect only the calling core

This method explicitly claims ownership of an unused user IRQ if there is one, so other code can know it is being used.

Parameters

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

Returns

the user IRQ number or -1 if required was false, and none were free

4.1.14.7.18. user_irq_unclaim

`void user_irq_unclaim (uint irq_num)`

Mark a user IRQ as no longer used on the calling core.

User IRQs starting from FIRST_USER_IRQ are not connected to any hardware, but can be triggered by irq_set_pending.

NOTE

User IRQs are a core local feature; they cannot be used to communicate between cores. Therefore all functions dealing with Uer IRQs affect only the calling core

This method explicitly releases ownership of a user IRQ, so other code can know it is free to use.

NOTE

it is customary to have disabled the irq and removed the handler prior to calling this method.

Parameters

`irq_num`	the irq irq_num to unclaim

4.1.15. hardware_pio

Programmable I/O (PIO) API.

4.1.15.1. Detailed Description

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.

There are three PIO blocks in the RP2350

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 appropriate RP-series datasheet. Note that there are additional features in the RP2350 PIO implementation that mean care should be taken when writing PIO code that needs to run on both the RP2040 and the RP2350.

On RP2040, pin numbers may always be specified from 0-31

On RP2350A, pin numbers may always be specified from 0-31.

On RP2350B, there are 48 pins but each PIO instance can only address 32 pins (the PIO instance either addresses pins 0-31 or 16-47 based on pio_set_gpio_base). The `pio_sm_` methods that directly affect the hardware always take *real* pin numbers in the full range, however:

- If `PICO_PIO_USE_GPIO_BASE != 1` then the 5th bit of the pin number is ignored. This is done so that programs compiled for boards with RP2350A do not incur the extra overhead of dealing with higher pins that don't exist. Effectively these functions behave exactly like RP2040 in this case. Note that `PICO_PIO_USE_GPIO_BASE` is defaulted to 0 if `PICO_RP2350A` is 1
- If `PICO_PIO_USE_GPIO_BASE == 1` then the passed pin numbers are adjusted internally by subtracting the GPIO base to give a pin number in the range 0-31 from the PIO's perspective

You can set `PARAM_ASSERTIONS_ENABLED_HARDWARE_PIO = 1` to enable parameter checking to debug pin (or other) issues with hardware_pio methods.

Note that pin masks follow the same rules as individual pins; bit N of a 32-bit or 64-bit mask always refers to pin N.

4.1.15.2. Modules

sm_config

PIO state machine configuration.

pio_instructions

PIO instruction encoding.

4.1.15.3. Macros

- `#define pio0 pio0_hw`
- `#define pio1 pio1_hw`
- `#define PIO_NUM(pio)`
- `#define PIO_INSTANCE(instance)`
- `#define PIO_FUNCSEL_NUM(pio, gpio)`
- `#define PIO_DREQ_NUM(pio, sm, is_tx)`
- `#define PIO_IRQ_NUM(pio, irqn)`

4.1.15.4. Typedefs

`typedef enum pio_interrupt_source pio_interrupt_source_t`

PIO interrupt source numbers for pio related IRQs.

4.1.15.5. Enumerations

`enum pio_fifo_join { PIO_FIFO_JOIN_NONE = 0, PIO_FIFO_JOIN_TX = 1, PIO_FIFO_JOIN_RX = 2 }`

FIFO join states.

`enum pio_mov_status_type { STATUS_TX_LESSTHAN = 0, STATUS_RX_LESSTHAN = 1 }`

MOV status types.

`enum pio_interrupt_source { pis_interrupt0 = PIO_INTR_SM0_LSB, pis_interrupt1 = PIO_INTR_SM1_LSB, pis_interrupt2 = PIO_INTR_SM2_LSB, pis_interrupt3 = PIO_INTR_SM3_LSB, pis_sm0_tx_fifo_not_full = PIO_INTR_SM0_TXNFULL_LSB, pis_sm1_tx_fifo_not_full = PIO_INTR_SM1_TXNFULL_LSB, pis_sm2_tx_fifo_not_full = PIO_INTR_SM2_TXNFULL_LSB, pis_sm3_tx_fifo_not_full = PIO_INTR_SM3_TXNFULL_LSB, pis_sm0_rx_fifo_not_empty = PIO_INTR_SM0_RXNEMPTY_LSB, pis_sm1_rx_fifo_not_empty = PIO_INTR_SM1_RXNEMPTY_LSB, pis_sm2_rx_fifo_not_empty = PIO_INTR_SM2_RXNEMPTY_LSB, pis_sm3_rx_fifo_not_empty = PIO_INTR_SM3_RXNEMPTY_LSB }`

PIO interrupt source numbers for pio related IRQs.

4.1.15.6. Functions

`static uint pio_get_gpio_base (PIO pio)`

Return the base GPIO base for the PIO instance.

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

Apply a state machine configuration to a state machine.

`static uint pio_get_index (PIO pio)`

Return the instance number of a PIO instance.

`static uint pio_get_funcsel (PIO pio)`

Return the funcsel number of a PIO instance.

`static PIO pio_get_instance (uint instance)`

Convert PIO instance to hardware instance.

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

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

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

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

`int pio_set_gpio_base (PIO pio, uint gpio_base)`

Set the base GPIO base for the PIO instance.

`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.

`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.

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

Attempt to load the program.

`int 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.

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

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

`void pio_clear_instruction_memory (PIO pio)`

Clears all of a PIO instance's instruction memory.

`static void pio_sm_set_enabled (PIO pio, uint sm, bool enabled)`

Enable or disable a PIO state machine.

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

Enable or disable multiple PIO state machines.

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

Restart a state machine with a known state.

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

Restart multiple state machine with a known state.

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

Restart a state machine's clock divider from a phase of 0.

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

Restart multiple state machines' clock dividers from a phase of 0.

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

Enable multiple PIO state machines synchronizing their clock dividers.

`static void pio_set_irq0_source_enabled (PIO pio, pio_interrupt_source_t source, bool enabled)`

Enable/Disable a single source on a PIO's IRQ 0.

`static void pio_set_irq1_source_enabled (PIO pio, pio_interrupt_source_t source, bool enabled)`

Enable/Disable a single source on a PIO's IRQ 1.

`static void pio_set_irq0_source_mask_enabled (PIO pio, uint32_t source_mask, bool enabled)`

Enable/Disable multiple sources on a PIO's IRQ 0.

`static void pio_set_irq1_source_mask_enabled (PIO pio, uint32_t source_mask, bool enabled)`

Enable/Disable multiple sources on a PIO's IRQ 1.

`static void pio_set_irqn_source_enabled (PIO pio, uint irq_index, pio_interrupt_source_t source, bool enabled)`

Enable/Disable a single source on a PIO's specified (0/1) IRQ index.

`static void pio_set_irqn_source_mask_enabled (PIO pio, uint irq_index, uint32_t source_mask, bool enabled)`

Enable/Disable multiple sources on a PIO's specified (0/1) IRQ index.

`static bool pio_interrupt_get (PIO pio, uint pio_interrupt_num)`

Determine if a particular PIO interrupt is set.

`static void pio_interrupt_clear (PIO pio, uint pio_interrupt_num)`

Clear a particular PIO interrupt.

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

Return the current program counter for a state machine.

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

Immediately execute an instruction on a state machine.

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

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

`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.

`static void pio_sm_set_wrap (PIO pio, uint sm, uint wrap_target, uint wrap)`

Set the current wrap configuration for a state machine.

`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.

`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.

`static void pio_sm_set_in_pins (PIO pio, uint sm, uint in_base)`

Set the current 'in' pins for a state machine.

`static void pio_sm_set_sideset_pins (PIO pio, uint sm, uint sideset_base)`

Set the current 'sideset' pins for a state machine.

`static void pio_sm_set_jmp_pin (PIO pio, uint sm, uint pin)`

Set the 'jmp' pin for a state machine.

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

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

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

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

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

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

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

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

`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.

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

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

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

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

`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.

`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.

`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.

`void pio_sm_drain_tx_fifo (PIO pio, uint sm)`

Empty out a state machine's TX FIFO.

`static void pio_sm_set_clkdiv_int_frac8 (PIO pio, uint sm, uint32_t div_int, uint8_t div_frac8)`

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

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

set the current clock divider for a state machine

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

Clear a state machine's TX and RX FIFOs.

`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.

`void pio_sm_set_pins64 (PIO pio, uint sm, uint64_t pin_values)`

Use a state machine to set a value on all pins for the PIO instance.

`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.

`void pio_sm_set_pins_with_mask64 (PIO pio, uint sm, uint64_t pin_values, uint64_t pin_mask)`

Use a state machine to set a value on multiple pins for the PIO instance.

`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.

`void pio_sm_set_pindirs_with_mask64 (PIO pio, uint sm, uint64_t pin_dirs, uint64_t pin_mask)`

Use a state machine to set the pin directions for multiple pins for the PIO instance.

`int pio_sm_set_consecutive_pindirs (PIO pio, uint sm, uint pins_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.

`void pio_sm_claim (PIO pio, uint sm)`

Mark a state machine as used.

`void pio_claim_sm_mask (PIO pio, uint sm_mask)`

Mark multiple state machines as used.

`void pio_sm_unclaim (PIO pio, uint sm)`

Mark a state machine as no longer used.

`int pio_claim_unused_sm (PIO pio, bool required)`

Claim a free state machine on a PIO instance.

`bool pio_sm_is_claimed (PIO pio, uint sm)`

Determine if a PIO state machine is claimed.

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

Finds a PIO and statemachine and adds a program into PIO memory.

`bool pio_claim_free_sm_and_add_program_for_gpio_range (const pio_program_t *program, PIO *pio, uint *sm, uint *offset, uint gpio_base, uint gpio_count, bool set_gpio_base)`

Finds a PIO and statemachine and adds a program into PIO memory.

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

Removes a program from PIO memory and unclaims the state machine.

`static int pio_get_irq_num (PIO pio, uint irqn)`

Return an IRQ for a PIO hardware instance.

`static pio_interrupt_source_t pio_get_tx_fifo_not_full_interrupt_source (uint sm)`

Return the interrupt source for a state machines TX FIFO not full interrupt.

`static pio_interrupt_source_t pio_get_rx_fifo_not_empty_interrupt_source (uint sm)`

Return the interrupt source for a state machines RX FIFO not empty interrupt.

4.1.15.7. Macro Definition Documentation

4.1.15.7.1. pio0

```
#define pio0 pio0_hw
```

Identifier for the first (PIO 0) hardware PIO instance (for use in PIO functions).

e.g. pio_gpio_init(pio0, 5)

4.1.15.7.2. pio1

```
#define pio1 pio1_hw
```

Identifier for the second (PIO 1) hardware PIO instance (for use in PIO functions).

e.g. pio_gpio_init(pio1, 5)

4.1.15.7.3. PIO_NUM

```
#define PIO_NUM(pio)
```

Returns the PIO number for a PIO instance.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.15.7.4. PIO_INSTANCE

```
#define PIO_INSTANCE(instance)
```

Returns the PIO instance with the given PIO number.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.15.7.5. PIO_FUNCSEL_NUM

```
#define PIO_FUNCSEL_NUM(pio, gpio)
```

Returns gpio_function_t needed to select the PIO function for the given PIO instance on the given GPIO.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.15.7.6. PIO_DREQ_NUM

```
#define PIO_DREQ_NUM(pio, sm, is_tx)
```

Returns the dreq_num_t used for pacing DMA transfers to or from a given state machine's FIFOs on this PIO instance. If is_tx is true, then it is for transfers to the PIO state machine TX FIFO else for transfers from the PIO state machine RX FIFO.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.15.7.7. PIO_IRQ_NUM

```
#define PIO_IRQ_NUM(pio, irqn)
```

Returns the irq_num_t for processor interrupts from the given PIO instance.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.15.8. Typedef Documentation

4.1.15.8.1. pio_interrupt_source_t

```
typedef enum pio_interrupt_source pio_interrupt_source_t
```

PIO interrupt source numbers for pio related IRQs.

4.1.15.9. Enumeration Type Documentation

4.1.15.9.1. pio_fifo_join

```
enum pio_fifo_join
```

FIFO join states.

Table 24. Enumerator

PIO_FIFO_JOIN_NONE	TX FIFO length=4 is used for transmit, RX FIFO length=4 is used for receive.
PIO_FIFO_JOIN_TX	TX FIFO length=8 is used for transmit, RX FIFO is disabled.
PIO_FIFO_JOIN_RX	RX FIFO length=8 is used for receive, TX FIFO is disabled.

4.1.15.9.2. pio_mov_status_type

```
enum pio_mov_status_type
```

MOV status types.

4.1.15.9.3. pio_interrupt_source

```
enum pio_interrupt_source
```

PIO interrupt source numbers for pio related IRQs.

Table 25. Enumerator

pis_interrupt0	PIO interrupt 0 is raised.
pis_interrupt1	PIO interrupt 1 is raised.

pis_interrupt2	PIO interrupt 2 is raised.
pis_interrupt3	PIO interrupt 3 is raised.
pis_sm0_tx_fifo_not_full	State machine 0 TX FIFO is not full.
pis_sm1_tx_fifo_not_full	State machine 1 TX FIFO is not full.
pis_sm2_tx_fifo_not_full	State machine 2 TX FIFO is not full.
pis_sm3_tx_fifo_not_full	State machine 3 TX FIFO is not full.
pis_sm0_rx_fifo_not_empty	State machine 0 RX FIFO is not empty.
pis_sm1_rx_fifo_not_empty	State machine 1 RX FIFO is not empty.
pis_sm2_rx_fifo_not_empty	State machine 2 RX FIFO is not empty.
pis_sm3_rx_fifo_not_empty	State machine 3 RX FIFO is not empty.

4.1.15.10. Function Documentation

4.1.15.10.1. pio_add_program

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

Attempt to load the program.

See also

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

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`program`	the program definition

Returns

the instruction memory offset the program is loaded at, or negative for error (for backwards compatibility with prior SDK the error value is -1 i.e. PICO_ERROR_GENERIC)

4.1.15.10.2. pio_add_program_at_offset

`int 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.

See also

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

Parameters

`pio`	The PIO instance; e.g. 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, or negative for error (for backwards compatibility with prior SDK the error value is -1 i.e. PICO_ERROR_GENERIC)

4.1.15.10.3. 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; e.g. pio0 or pio1
`program`	the program definition

Returns

true if the program can be loaded; false if not, e.g. if there is not suitable space in the instruction memory

4.1.15.10.4. 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; e.g. 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 not, e.g. if there is not space in the instruction memory

4.1.15.10.5. pio_claim_free_sm_and_add_program

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

Finds a PIO and statemachine and adds a program into PIO memory.

Parameters

`program`	PIO program to add
`pio`	Returns the PIO hardware instance or NULL if no PIO is available
`sm`	Returns the index of the PIO state machine that was claimed
`offset`	Returns the instruction memory offset of the start of the program

Returns

true on success, false otherwise

See also

pio_remove_program_unclaim_sm

4.1.15.10.6. pio_claim_free_sm_and_add_program_for_gpio_range

`bool pio_claim_free_sm_and_add_program_for_gpio_range (const pio_program_t * program, PIO * pio, uint * sm, uint * offset, uint gpio_base, uint gpio_count, bool set_gpio_base)`

Finds a PIO and statemachine and adds a program into PIO memory.

This variation of pio_claim_free_sm_and_add_program is useful on RP2350 QFN80 where the "GPIO Base" must be set per PIO instance to either address the 32 GPIOs (0->31) or the 32 GPIOS (16-47). No single PIO instance can interact with both pins 0->15 or 32->47 at the same time.

This method takes additional information about the GPIO pins needed (via gpio_base and gpio_count), and optionally will set the GPIO base (

See also

pio_set_gpio_base) of an unused PIO instance if necessary

Parameters

`program`	PIO program to add
`pio`	Returns the PIO hardware instance or NULL if no PIO is available
`sm`	Returns the index of the PIO state machine that was claimed
`offset`	Returns the instruction memory offset of the start of the program
`gpio_base`	the lowest GPIO number required (0-47 on RP2350B, 0-31 otherwise)
`gpio_count`	the count of GPIOs required
`set_gpio_base`	if there is no free SM on a PIO instance with the right GPIO base, and there IS an unused PIO instance, then that PIO will be reconfigured so that this method can succeed

Returns

true on success, false otherwise

See also

pio_remove_program_unclaim_sm

4.1.15.10.7. 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; e.g. pio0 or pio1
`sm_mask`	Mask of state machine indexes

4.1.15.10.8. 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; e.g. pio0 or pio1
`required`	if true the function will panic if none are available

Returns

the state machine index or negative if required was false, and none were free (for backwards compatibility with prior SDK the error value is -1 i.e. PICO_ERROR_GENERIC)

4.1.15.10.9. 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; e.g. pio0 or pio1

4.1.15.10.10. pio_clkdiv_restart_sm_mask

```
static void pio_clkdiv_restart_sm_mask (PIO pio, uint32_t mask) [inline], [static]
```

Restart multiple state machines' clock dividers from a phase of 0.

Each state machine's clock divider is a free-running piece of hardware, that generates a pattern of clock enable pulses for the state machine, based *only* on the configured integer/fractional divisor. The pattern of running/halted cycles slows the state machine's execution to some controlled rate.

This function simultaneously clears the integer and fractional phase accumulators of multiple state machines' clock dividers. If these state machines all have the same integer and fractional divisors configured, their clock dividers will run in precise deterministic lockstep from this point.

With their execution clocks synchronised in this way, it is then safe to e.g. have multiple state machines performing a 'wait irq' on the same flag, and all clear it on the same cycle.

Also note that this function can be called whilst state machines are running (e.g. if you have just changed the clock divisors of some state machines and wish to resynchronise them), and that disabling a state machine does not halt its clock divider: that is, if multiple state machines have their clocks synchronised, you can safely disable and re-enable one of the state machines without losing synchronisation.

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`mask`	bit mask of state machine indexes to modify the enabled state of

4.1.15.10.11. pio_enable_sm_mask_in_sync

```
static void pio_enable_sm_mask_in_sync (PIO pio, uint32_t mask) [inline], [static]
```

Enable multiple PIO state machines synchronizing their clock dividers.

This is equivalent to calling both pio_set_sm_mask_enabled() and pio_clkdiv_restart_sm_mask() on the *same* clock cycle. All state machines specified by 'mask' are started simultaneously and, assuming they have the same clock divisors, their divided clocks will stay precisely synchronised.

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`mask`	bit mask of state machine indexes to modify the enabled state of

4.1.15.10.12. pio_get_dreq

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

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

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1

`sm` State machine index (0..3)

`is_tx` true for sending data to the state machine, false for receiving data from the state machine

4.1.15.10.13. pio_get_funcsel

`static uint pio_get_funcsel (PIO pio) [inline], [static]`

Return the funcsel number of a PIO instance.

Parameters

`pio` The PIO instance; e.g. pio0 or pio1

Returns

the PIO instance number (0, 1, …)

See also

gpio_function

4.1.15.10.14. pio_get_gpio_base

`static uint pio_get_gpio_base (PIO pio) [inline], [static]`

Return the base GPIO base for the PIO instance.

This method always return 0 in RP2040

Parameters

`pio` The PIO instance; e.g. pio0 or pio1

Returns

the current GPIO base for the PIO instance

4.1.15.10.15. pio_get_index

`static uint pio_get_index (PIO pio) [inline], [static]`

Return the instance number of a PIO instance.

Parameters

`pio` The PIO instance; e.g. pio0 or pio1

Returns

the PIO instance number (0, 1, …)

4.1.15.10.16. pio_get_instance

`static PIO pio_get_instance (uint instance) [inline], [static]`

Convert PIO instance to hardware instance.

Parameters

`instance` Instance of PIO, 0 or 1

Returns

the PIO hardware instance

4.1.15.10.17. pio_get_irq_num

`static int pio_get_irq_num (PIO pio, uint irqn) [inline], [static]`

Return an IRQ for a PIO hardware instance.

Parameters

- `pio` PIO hardware instance
- `irqn` 0 for PIOx_IRQ_0 or 1 for PIOx_IRQ_1 etc where x is the PIO number

Returns

The IRQ number to use for the PIO

4.1.15.10.18. pio_get_rx_fifo_not_empty_interrupt_source

`static pio_interrupt_source_t pio_get_rx_fifo_not_empty_interrupt_source (uint sm) [inline], [static]`

Return the interrupt source for a state machines RX FIFO not empty interrupt.

Parameters

- `sm` State machine index (0..3)

Returns

The interrupt source number for use in pio_set_irqn_source_enabled or similar functions

4.1.15.10.19. pio_get_tx_fifo_not_full_interrupt_source

`static pio_interrupt_source_t pio_get_tx_fifo_not_full_interrupt_source (uint sm) [inline], [static]`

Return the interrupt source for a state machines TX FIFO not full interrupt.

Parameters

- `sm` State machine index (0..3)

Returns

The interrupt source number for use in pio_set_irqn_source_enabled or similar functions

4.1.15.10.20. pio_gpio_init

`static void pio_gpio_init (PIO pio, uint pin) [inline], [static]`

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

PIO appears as an alternate function in the GPIO muxing, just like an SPI or UART. This function configures that multiplexing to connect a given PIO instance to a GPIO. Note that this is not necessary for a state machine to be able to read the *input* value from a GPIO, but only for it to set the output value or output enable.

Parameters

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

4.1.15.10.21. pio_interrupt_clear

`static void pio_interrupt_clear (PIO pio, uint pio_interrupt_num) [inline], [static]`

Clear a particular PIO interrupt.

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`pio_interrupt_num`	the PIO interrupt number 0-7

4.1.15.10.22. pio_interrupt_get

`static bool pio_interrupt_get (PIO pio, uint pio_interrupt_num) [inline], [static]`

Determine if a particular PIO interrupt is set.

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`pio_interrupt_num`	the PIO interrupt number 0-7

Returns

true if corresponding PIO interrupt is currently set

4.1.15.10.23. 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; e.g. pio0 or pio1
`program`	the program definition
`loaded_offset`	the loaded offset returned when the program was added

4.1.15.10.24. pio_remove_program_and_unclaim_sm

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

Removes a program from PIO memory and unclaims the state machine.

Parameters

`program`	PIO program to remove from memory
`pio`	PIO hardware instance being used
`sm`	PIO state machine that was claimed
`offset`	offset of the program in PIO memory

See also

pio_claim_free_sm_and_add_program

4.1.15.10.25. pio_restart_sm_mask

`static void pio_restart_sm_mask (PIO pio, uint32_t mask) [inline], [static]`

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; e.g. pio0 or pio1
`mask`	bit mask of state machine indexes to modify the enabled state of

4.1.15.10.26. pio_set_gpio_base

```
int pio_set_gpio_base (PIO pio, uint gpio_base)
```

Set the base GPIO base for the PIO instance.

Since an individual PIO accesses only 32 pins, to be able to access more pins, the PIO instance must specify a base GPIO where the instance's "pin 0" maps. For RP2350 the valid values are 0 and 16, indicating the PIO instance has access to pins 0-31, or 16-47 respectively.

> **NOTE**
>
> This method simply changes the underlying PIO register, it does not detect or attempt to prevent any side effects this change will have on in use state machines on this PIO.

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`gpio_base`	the GPIO base (either 0 or 16)

Returns

PICO_OK (0) on success, error code otherwise

4.1.15.10.27. pio_set_irq0_source_enabled

```
static void pio_set_irq0_source_enabled (PIO pio, pio_interrupt_source_t source, bool enabled) [inline], [static]
```

Enable/Disable a single source on a PIO's IRQ 0.

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`source`	the source number (see pio_interrupt_source)
`enabled`	true to enable IRQ 0 for the source, false to disable.

4.1.15.10.28. pio_set_irq0_source_mask_enabled

```
static void pio_set_irq0_source_mask_enabled (PIO pio, uint32_t source_mask, bool enabled) [inline], [static]
```

Enable/Disable multiple sources on a PIO's IRQ 0.

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`source_mask`	Mask of bits, one for each source number (see pio_interrupt_source) to affect
`enabled`	true to enable all the sources specified in the mask on IRQ 0, false to disable all the sources specified in the mask on IRQ 0

4.1.15.10.29. pio_set_irq1_source_enabled

`static void pio_set_irq1_source_enabled (PIO pio, pio_interrupt_source_t source, bool enabled) [inline], [static]`

Enable/Disable a single source on a PIO's IRQ 1.

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`source`	the source number (see pio_interrupt_source)
`enabled`	true to enable IRQ 0 for the source, false to disable.

4.1.15.10.30. pio_set_irq1_source_mask_enabled

`static void pio_set_irq1_source_mask_enabled (PIO pio, uint32_t source_mask, bool enabled) [inline], [static]`

Enable/Disable multiple sources on a PIO's IRQ 1.

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`source_mask`	Mask of bits, one for each source number (see pio_interrupt_source) to affect
`enabled`	true to enable all the sources specified in the mask on IRQ 1, false to disable all the source specified in the mask on IRQ 1

4.1.15.10.31. pio_set_irqn_source_enabled

`static void pio_set_irqn_source_enabled (PIO pio, uint irq_index, pio_interrupt_source_t source, bool enabled) [inline], [static]`

Enable/Disable a single source on a PIO's specified (0/1) IRQ index.

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`irq_index`	the IRQ index; either 0 or 1
`source`	the source number (see pio_interrupt_source)
`enabled`	true to enable the source on the specified IRQ, false to disable.

4.1.15.10.32. pio_set_irqn_source_mask_enabled

`static void pio_set_irqn_source_mask_enabled (PIO pio, uint irq_index, uint32_t source_mask, bool enabled) [inline], [static]`

Enable/Disable multiple sources on a PIO's specified (0/1) IRQ index.

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`irq_index`	the IRQ index; either 0 or 1
`source_mask`	Mask of bits, one for each source number (see pio_interrupt_source) to affect
`enabled`	true to enable all the sources specified in the mask on the specified IRQ, false to disable all the sources specified in the mask on the specified IRQ

4.1.15.10.33. pio_set_sm_mask_enabled

`static void pio_set_sm_mask_enabled (PIO pio, uint32_t mask, bool enabled) [inline], [static]`

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.

See also

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

Parameters

`pio`	The PIO instance; e.g. 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

4.1.15.10.34. 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; e.g. pio0 or pio1
`sm`	State machine index (0..3)

4.1.15.10.35. pio_sm_clear_fifos

`static void pio_sm_clear_fifos (PIO pio, uint sm) [inline], [static]`

Clear a state machine's TX and RX FIFOs.

Parameters

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

4.1.15.10.36. pio_sm_clkdiv_restart

`static void pio_sm_clkdiv_restart (PIO pio, uint sm) [inline], [static]`

Restart a state machine's clock divider from a phase of 0.

Each state machine's clock divider is a free-running piece of hardware, that generates a pattern of clock enable pulses for the state machine, based *only* on the configured integer/fractional divisor. The pattern of running/halted cycles slows the state machine's execution to some controlled rate.

This function clears the divider's integer and fractional phase accumulators so that it restarts this pattern from the beginning. It is called automatically by pio_sm_init() but can also be called at a later time, when you enable the state machine, to ensure precisely consistent timing each time you load and run a given PIO program.

More commonly this hardware mechanism is used to synchronise the execution clocks of multiple state machines – see pio_clkdiv_restart_sm_mask().

Parameters

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

4.1.15.10.37. 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. This disturbs the contents of the OSR, so see also pio_sm_clear_fifos() which clears both FIFOs but leaves the state machine's internal state undisturbed.

Parameters

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

See also

pio_sm_clear_fifos()

4.1.15.10.38. pio_sm_exec

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

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.

See also

pio_sm_is_exec_stalled() to see if an executed instruction is still running (i.e. it is stalled on some condition)

Parameters

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

4.1.15.10.39. pio_sm_exec_wait_blocking

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

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.

See also

pio_sm_is_exec_stalled() to see if an executed instruction is still running (i.e. it is stalled on some condition)

Parameters

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

`instr` the encoded PIO instruction

4.1.15.10.40. pio_sm_get

`static uint32_t pio_sm_get (PIO pio, uint sm) [inline], [static]`

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

This is a raw FIFO access that does not check for emptiness. If the FIFO is empty, the hardware ignores the attempt to read from the FIFO (the FIFO remains in an empty state following the read) and the sticky RXUNDER flag for this FIFO is set in FDEBUG to indicate that the system tried to read from this FIFO when empty. The data returned by this function is undefined when the FIFO is empty.

Parameters

`pio` The PIO instance; e.g. pio0 or pio1

`sm` State machine index (0..3)

See also

pio_sm_get_blocking()

4.1.15.10.41. pio_sm_get_blocking

`static uint32_t pio_sm_get_blocking (PIO pio, uint sm) [inline], [static]`

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

Parameters

`pio` The PIO instance; e.g. pio0 or pio1

`sm` State machine index (0..3)

4.1.15.10.42. pio_sm_get_pc

`static uint8_t pio_sm_get_pc (PIO pio, uint sm) [inline], [static]`

Return the current program counter for a state machine.

Parameters

`pio` The PIO instance; e.g. pio0 or pio1

`sm` State machine index (0..3)

Returns

the program counter

4.1.15.10.43. pio_sm_get_rx_fifo_level

`static uint pio_sm_get_rx_fifo_level (PIO pio, uint sm) [inline], [static]`

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

Parameters

`pio` The PIO instance; e.g. pio0 or pio1

`sm` State machine index (0..3)

Returns

the number of elements in the RX FIFO

4.1.15.10.44. pio_sm_get_tx_fifo_level

`static uint pio_sm_get_tx_fifo_level (PIO pio, uint sm) [inline], [static]`

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

Parameters

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

Returns

the number of elements in the TX FIFO

4.1.15.10.45. pio_sm_is_claimed

`bool pio_sm_is_claimed (PIO pio, uint sm)`

Determine if a PIO state machine is claimed.

Parameters

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

Returns

true if claimed, false otherwise

See also

pio_sm_claim

pio_claim_sm_mask

4.1.15.10.46. pio_sm_is_exec_stalled

`static bool pio_sm_is_exec_stalled (PIO pio, uint sm) [inline], [static]`

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

Parameters

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

Returns

true if the executed instruction is still running (stalled)

4.1.15.10.47. pio_sm_is_rx_fifo_empty

`static bool pio_sm_is_rx_fifo_empty (PIO pio, uint sm) [inline], [static]`

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

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1

`sm` State machine index (0..3)

Returns

true if the RX FIFO is empty

4.1.15.10.48. pio_sm_is_rx_fifo_full

`static bool pio_sm_is_rx_fifo_full (PIO pio, uint sm) [inline], [static]`

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

Parameters

`pio` The PIO instance; e.g. pio0 or pio1

`sm` State machine index (0..3)

Returns

true if the RX FIFO is full

4.1.15.10.49. pio_sm_is_tx_fifo_empty

`static bool pio_sm_is_tx_fifo_empty (PIO pio, uint sm) [inline], [static]`

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

Parameters

`pio` The PIO instance; e.g. pio0 or pio1

`sm` State machine index (0..3)

Returns

true if the TX FIFO is empty

4.1.15.10.50. pio_sm_is_tx_fifo_full

`static bool pio_sm_is_tx_fifo_full (PIO pio, uint sm) [inline], [static]`

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

Parameters

`pio` The PIO instance; e.g. pio0 or pio1

`sm` State machine index (0..3)

Returns

true if the TX FIFO is full

4.1.15.10.51. pio_sm_put

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

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

This is a raw FIFO access that does not check for fullness. If the FIFO is full, the FIFO contents and state are not affected by the write attempt. Hardware sets the TXOVER sticky flag for this FIFO in FDEBUG, to indicate that the system attempted to write to a full FIFO.

Parameters

`pio` The PIO instance; e.g. pio0 or pio1

`sm` State machine index (0..3)

`data` the 32 bit data value

See also

pio_sm_put_blocking()

4.1.15.10.52. pio_sm_put_blocking

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

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

Parameters

`pio` The PIO instance; e.g. pio0 or pio1

`sm` State machine index (0..3)

`data` the 32 bit data value

4.1.15.10.53. pio_sm_restart

`static void pio_sm_restart (PIO pio, uint sm) [inline], [static]`

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; e.g. pio0 or pio1

`sm` State machine index (0..3)

4.1.15.10.54. pio_sm_set_clkdiv

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

set the current clock divider for a state machine

Parameters

`pio` The PIO instance; e.g. pio0 or pio1

`sm` State machine index (0..3)

`div` the floating point clock divider

4.1.15.10.55. pio_sm_set_clkdiv_int_frac8

`static void pio_sm_set_clkdiv_int_frac8 (PIO pio, uint sm, uint32_t div_int, uint8_t div_frac8) [inline], [static]`

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

Parameters

`pio` The PIO instance; e.g. pio0 or pio1

`sm` State machine index (0..3)

`div_int`	the integer part of the clock divider
`div_frac8`	the fractional part of the clock divider in 1/256s

4.1.15.10.56. pio_sm_set_config

```
static int pio_sm_set_config (PIO pio, uint sm, const pio_sm_config * config) [inline], [static]
```

Apply a state machine configuration to a state machine.

See sm_config_ pins for more detail on why this method might fail on RP2350B

Parameters

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

Returns

PICO_OK (0) on success, negative error code otherwise

4.1.15.10.57. pio_sm_set_consecutive_pindirs

```
int pio_sm_set_consecutive_pindirs (PIO pio, uint sm, uint pins_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; e.g. pio0 or pio1
`sm`	State machine index (0..3) to use
`pins_base`	the first pin to set a direction for. See pio_sm_ pins for more detail on pin arguments
`pin_count`	the count of consecutive pins to set the direction for
`is_out`	the direction to set; true = out, false = in

Returns

PICO_OK (0) on success, error code otherwise

4.1.15.10.58. pio_sm_set_enabled

```
static void pio_sm_set_enabled (PIO pio, uint sm, bool enabled) [inline], [static]
```

Enable or disable a PIO state machine.

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`sm`	State machine index (0..3)
`enabled`	true to enable the state machine; false to disable

4.1.15.10.59. pio_sm_set_in_pins

`static void pio_sm_set_in_pins (PIO pio, uint sm, uint in_base) [inline], [static]`

Set the current 'in' pins for a state machine.

'in' pins can overlap with the 'out', 'set' and 'sideset' pins

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`sm`	State machine index (0..3)
`in_base`	First pin to use as input. See pio_sm_ pins for more detail on pin arguments

4.1.15.10.60. pio_sm_set_jmp_pin

`static void pio_sm_set_jmp_pin (PIO pio, uint sm, uint pin) [inline], [static]`

Set the 'jmp' pin for a state machine.

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`sm`	State machine index (0..3)
`pin`	The pin number to use as the source for a `jmp pin` instruction. See pio_sm_ pins for more detail on pin arguments

4.1.15.10.61. pio_sm_set_out_pins

`static void pio_sm_set_out_pins (PIO pio, uint sm, uint out_base, uint out_count) [inline], [static]`

Set the current 'out' pins for a state machine.

'out' pins can overlap with the 'in', 'set' and 'sideset' pins

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`sm`	State machine index (0..3)
`out_base`	First pin to set as output. See pio_sm_ pins for more detail on pin arguments
`out_count`	0-32 Number of pins to set.

4.1.15.10.62. 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. Note: This method only works for pins < 32. To use with pins >= 32 call pio_sm_set_pindirs_with_mask64

Parameters

`pio`	The PIO instance; e.g. 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.

4.1.15.10.63. pio_sm_set_pindirs_with_mask64

`void pio_sm_set_pindirs_with_mask64 (PIO pio, uint sm, uint64_t pin_dirs, uint64_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; e.g. 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.

4.1.15.10.64. 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. Note: This method only works for pins < 32. To use with pins >= 32 call pio_sm_set_pins64

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`sm`	State machine index (0..3) to use
`pin_values`	the pin values to set. See pio_sm_ pins for more detail on pin arguments

4.1.15.10.65. pio_sm_set_pins64

`void pio_sm_set_pins64 (PIO pio, uint sm, uint64_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; e.g. pio0 or pio1
`sm`	State machine index (0..3) to use

`pin_values`	the pin values to set. See pio_sm_ pins for more detail on pin arguments

4.1.15.10.66. 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. Note: This method only works for pins < 32. To use with pins >= 32 call pio_sm_set_pins_with_mask64

Parameters

`pio`	The PIO instance; e.g. 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. See pio_sm_ pins for more detail on pin arguments

4.1.15.10.67. pio_sm_set_pins_with_mask64

`void pio_sm_set_pins_with_mask64 (PIO pio, uint sm, uint64_t pin_values, uint64_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; e.g. 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. See pio_sm_ pins for more detail on pin arguments

4.1.15.10.68. pio_sm_set_set_pins

`static void pio_sm_set_set_pins (PIO pio, uint sm, uint set_base, uint set_count) [inline], [static]`

Set the current 'set' pins for a state machine.

'set' pins can overlap with the 'in', 'out' and 'sideset' pins

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`sm`	State machine index (0..3)
`set_base`	First pin to set as 'set'. See pio_sm_ pins for more detail on pin arguments
`set_count`	0-5 Number of pins to set.

4.1.15.10.69. pio_sm_set_sideset_pins

`static void pio_sm_set_sideset_pins (PIO pio, uint sm, uint sideset_base) [inline], [static]`

Set the current 'sideset' pins for a state machine.

'sideset' pins can overlap with the 'in', 'out' and 'set' pins

Parameters

`pio`	The PIO instance; e.g. pio0 or pio1
`sm`	State machine index (0..3)
`sideset_base`	Base pin for 'side set'. See pio_sm_ pins for more detail on pin arguments

4.1.15.10.70. pio_sm_set_wrap

`static void pio_sm_set_wrap (PIO pio, uint sm, uint wrap_target, uint wrap) [inline], [static]`

Set the current wrap configuration for a state machine.

Parameters

`pio`	The PIO instance; e.g. 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

4.1.15.10.71. 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; e.g. pio0 or pio1
`sm`	State machine index (0..3)

4.1.15.11. sm_config

PIO state machine configuration.

4.1.15.11.1. Detailed Description

A PIO block needs to be configured, these functions provide helpers to set up configuration structures. See pio_sm_set_config

On RP2040, pin numbers may always be specified from 0-31

On RP2350A, pin numbers may always be specified from 0-31.

On RP2350B, there are 48 pins but each PIO instance can only address 32 pins (the PIO instance either addresses pins 0-31 or 16-47 based on pio_set_gpio_base). The `sm_config_` state machine configuration always take *real* pin numbers in the full range, however:

- If `PICO_PIO_USE_GPIO_BASE != 1` then the 5th bit of the pin number is ignored. This is done so that programs compiled for boards with RP2350A do not incur the extra overhead of dealing with higher pins that don't exist. Effectively these functions behave exactly like RP2040 in this case. Note that `PICO_PIO_USE_GPIO_BASE` is defaulted to 0 if `PICO_RP2350A` is 1
- If `PICO_PIO_USE_GPIO_BASE == 1` then the state machine configuration stores the actual pin numbers in the range 0-47. Of course in this scenario, it is possible to make an invalid configuration (one which uses pins in both the ranges 0-15 and 32-47).

  pio_sm_set_config (or pio_sm_init which calls it) attempts to apply the configuration to a particular PIO's state machine, and will return PICO_ERROR_BAD_ALIGNMENT if the configuration cannot be applied due to the above problem, or if the PIO's GPIO base (see pio_set_gpio_base) does not allow access to the required pins.

  To be clear, pio_sm_set_config does not change the PIO's GPIO base for you; you must configre the PIO's GPIO base before calling the method, however you can use pio_claim_free_sm_and_add_program_for_gpio_range to find/configure a PIO instance suitable for a partiular GPIO range.

You can set `PARAM_ASSERTIONS_ENABLED_HARDWARE_PIO = 1` to enable parameter checking to debug pin (or other) issues with hardware_pio methods.

4.1.15.11.2. Functions

`static void sm_config_set_out_pin_base (pio_sm_config *c, uint out_base)`

Set the base of the 'out' pins in a state machine configuration.

`static void sm_config_set_out_pin_count (pio_sm_config *c, uint out_count)`

Set the number of 'out' pins in a state machine configuration.

`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.

`static void sm_config_set_set_pin_base (pio_sm_config *c, uint set_base)`

Set the base of the 'set' pins in a state machine configuration.

`static void sm_config_set_set_pin_count (pio_sm_config *c, uint set_count)`

Set the count of 'set' pins in a state machine configuration.

`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.

`static void sm_config_set_in_pin_base (pio_sm_config *c, uint in_base)`

Set the base of the 'in' pins in a state machine configuration.

`static void sm_config_set_in_pins (pio_sm_config *c, uint in_base)`

Set the base for the 'in' pins in a state machine configuration.

`static void sm_config_set_in_pin_count (pio_sm_config *c, uint in_count)`

Set the count of 'in' pins in a state machine configuration.

`static void sm_config_set_sideset_pin_base (pio_sm_config *c, uint sideset_base)`

Set the base of the 'sideset' pins in a state machine configuration.

`static void sm_config_set_sideset_pins (pio_sm_config *c, uint sideset_base)`

Set the 'sideset' pins in a state machine configuration.

`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.

`static void sm_config_set_clkdiv_int_frac8 (pio_sm_config *c, uint32_t div_int, uint8_t div_frac8)`

Set the state machine clock divider (from integer and fractional parts - 16:8) in a state machine configuration.

`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.

`static void sm_config_set_wrap (pio_sm_config *c, uint wrap_target, uint wrap)`

Set the wrap addresses in a state machine configuration.

`static void sm_config_set_jmp_pin (pio_sm_config *c, uint pin)`

Set the 'jmp' pin in a state machine configuration.

`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.

`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.

`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.

`static void sm_config_set_out_special (pio_sm_config *c, bool sticky, bool has_enable_pin, uint enable_bit_index)`

Set special 'out' operations in a state machine configuration.

`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.

`static pio_sm_config pio_get_default_sm_config (void)`

Get the default state machine configuration.

4.1.15.11.3. Function Documentation

pio_get_default_sm_config

`static pio_sm_config pio_get_default_sm_config (void) [inline], [static]`

Get the default state machine configuration.

Setting	Default
Out Pins	32 starting at 0
Set Pins	0 starting at 0
In Pins	32 starting at 0
Side Set Pins (base)	0
Side Set	disabled
Wrap	wrap=31, wrap_to=0
In Shift	shift_direction=right, autopush=false, push_threshold=32
Out Shift	shift_direction=right, autopull=false, pull_threshold=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.

sm_config_set_clkdiv

`static void sm_config_set_clkdiv (pio_sm_config * c, float div) [inline], [static]`

Set the state machine clock divider (from a floating point value) in a state machine configuration.

The clock divider slows the state machine's execution by masking the system clock on some cycles, in a repeating pattern, so that the state machine does not advance. Effectively this produces a slower clock for the state machine to run from, which can be used to generate e.g. a particular UART baud rate. See the datasheet for further detail.

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.

sm_config_set_clkdiv_int_frac8

`static void sm_config_set_clkdiv_int_frac8 (pio_sm_config * c, uint32_t div_int, uint8_t div_frac8) [inline], [static]`

Set the state machine clock divider (from integer and fractional parts - 16:8) in a state machine configuration.

The clock divider can slow the state machine's execution to some rate below the system clock frequency, by enabling the state machine on some cycles but not on others, in a regular pattern. This can be used to generate e.g. a given UART baud rate. See the datasheet for further detail.

Parameters

`c`	Pointer to the configuration structure to modify
`div_int`	Integer part of the divisor
`div_frac8`	Fractional part in 1/256ths

See also

sm_config_set_clkdiv()

sm_config_set_fifo_join

`static void sm_config_set_fifo_join (pio_sm_config * c, enum pio_fifo_join join) [inline], [static]`

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

sm_config_set_in_pin_base

`static void sm_config_set_in_pin_base (pio_sm_config * c, uint in_base) [inline], [static]`

Set the base of the 'in' pins in a state machine configuration.

'in' pins can overlap with the 'out', 'set' and 'sideset' pins

Parameters

`c`	Pointer to the configuration structure to modify
`in_base`	First pin to use as input. See sm_config_ pins for more detail on pin arguments

sm_config_set_in_pin_count

`static void sm_config_set_in_pin_count (pio_sm_config * c, uint in_count) [inline], [static]`

Set the count of 'in' pins in a state machine configuration.

When reading pins using the IN pin mapping, this many (low) bits will be read, with the rest taking the value zero.

RP2040 does not have the ability to mask unused input pins, so the in_count must be 32

Parameters

`c`	Pointer to the configuration structure to modify
`in_count`	1-32 The number of pins to include when reading via the IN pin mapping

sm_config_set_in_pins

`static void sm_config_set_in_pins (pio_sm_config * c, uint in_base) [inline], [static]`

Set the base for the 'in' pins in a state machine configuration.

'in' pins can overlap with the 'out', 'set' and 'sideset' pins

Parameters

`c`	Pointer to the configuration structure to modify
`in_base`	First pin to use as input. See sm_config_ pins for more detail on pin arguments

sm_config_set_in_shift

`static void sm_config_set_in_shift (pio_sm_config * c, bool shift_right, bool autopush, uint push_threshold) [inline], [static]`

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

sm_config_set_jmp_pin

`static void sm_config_set_jmp_pin (pio_sm_config * c, uint pin) [inline], [static]`

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. See sm_config_ pins for more detail on pin arguments

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) [inline], [static]`

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.

See also

enum pio_mov_status_type

Parameters

`status_n`	parameter for the mov status operation (currently a bit count)

sm_config_set_out_pin_base

`static void sm_config_set_out_pin_base (pio_sm_config * c, uint out_base) [inline], [static]`

Set the base of the 'out' pins in a state machine configuration.

'out' pins can overlap with the 'in', 'set' and 'sideset' pins

Parameters

`c`	Pointer to the configuration structure to modify
`out_base`	First pin to set as output. See sm_config_ pins for more detail on pin arguments

sm_config_set_out_pin_count

`static void sm_config_set_out_pin_count (pio_sm_config * c, uint out_count) [inline], [static]`

Set the number of 'out' pins in a state machine configuration.

'out' pins can overlap with the 'in', 'set' and 'sideset' pins

Parameters

`c`	Pointer to the configuration structure to modify
`out_count`	0-32 Number of pins to set.

sm_config_set_out_pins

`static void sm_config_set_out_pins (pio_sm_config * c, uint out_base, uint out_count) [inline], [static]`

Set the 'out' pins in a state machine configuration.

'out' pins can overlap with the 'in', 'set' and 'sideset' pins

Parameters

`c`	Pointer to the configuration structure to modify
`out_base`	First pin to set as output. See sm_config_ pins for more detail on pin arguments
`out_count`	0-32 Number of pins to set.

sm_config_set_out_shift

`static void sm_config_set_out_shift (pio_sm_config * c, bool shift_right, bool autopull, uint pull_threshold) [inline], [static]`

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

sm_config_set_out_special

`static void sm_config_set_out_special (pio_sm_config * c, bool sticky, bool has_enable_pin, uint enable_bit_index) [inline], [static]`

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_bit_index`	Data bit index for auxiliary OUT enable.

sm_config_set_set_pin_base

`static void sm_config_set_set_pin_base (pio_sm_config * c, uint set_base) [inline], [static]`

Set the base of the 'set' pins in a state machine configuration.

'set' pins can overlap with the 'in', 'out' and 'sideset' pins

Parameters

`c`	Pointer to the configuration structure to modify
`set_base`	First pin to use as 'set'. See sm_config_ pins for more detail on pin arguments

sm_config_set_set_pin_count

`static void sm_config_set_set_pin_count (pio_sm_config * c, uint set_count) [inline], [static]`

Set the count of 'set' pins in a state machine configuration.

'set' pins can overlap with the 'in', 'out' and 'sideset' pins

Parameters

`c`	Pointer to the configuration structure to modify
`set_count`	0-5 Number of pins to set.

sm_config_set_set_pins

`static void sm_config_set_set_pins (pio_sm_config * c, uint set_base, uint set_count) [inline], [static]`

Set the 'set' pins in a state machine configuration.

'set' pins can overlap with the 'in', 'out' and 'sideset' pins

Parameters

`c`	Pointer to the configuration structure to modify
`set_base`	First pin to use as 'set'. See sm_config_ pins for more detail on pin arguments
`set_count`	0-5 Number of pins to set.

sm_config_set_sideset

`static void sm_config_set_sideset (pio_sm_config * c, uint bit_count, bool optional, bool pindirs) [inline], [static]`

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 (max 5)
`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

sm_config_set_sideset_pin_base

`static void sm_config_set_sideset_pin_base (pio_sm_config * c, uint sideset_base) [inline], [static]`

Set the base of the 'sideset' pins in a state machine configuration.

'sideset' pins can overlap with the 'in', 'out' and 'set' pins

Parameters

`c`	Pointer to the configuration structure to modify
`sideset_base`	First pin to use for 'side set'. See sm_config_ pins for more detail on pin arguments

sm_config_set_sideset_pins

`static void sm_config_set_sideset_pins (pio_sm_config * c, uint sideset_base) [inline], [static]`

Set the 'sideset' pins in a state machine configuration.

This method is identical to sm_config_set_sideset_pin_base, and is provided for backwards compatibility

'sideset' pins can overlap with the 'in', 'out' and 'set' pins

Parameters

`c`	Pointer to the configuration structure to modify
`sideset_base`	First pin to use for 'side set'. See sm_config_ pins for more detail on pin arguments

sm_config_set_wrap

`static void sm_config_set_wrap (pio_sm_config * c, uint wrap_target, uint wrap) [inline], [static]`

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

4.1.15.12. pio_instructions

PIO instruction encoding.

4.1.15.12.1. Detailed Description

Functions for generating PIO instruction encodings programmatically. In debug builds `PARAM_ASSERTIONS_ENABLED_PIO_INSTRUCTIONS` can be set to 1 to enable validation of encoding function parameters.

For fuller descriptions of the instructions in question see the "RP2040 Datasheet"

4.1.15.12.2. Enumerations

`enum pio_src_dest { pio_pins = 0u, pio_x = 1u, pio_y = 2u, pio_null = 3u | 0x20u | 0x80u, pio_pindirs = 4u | 0x08u | 0x40u | 0x80u, pio_exec_mov = 4u | 0x08u | 0x10u | 0x20u | 0x40u, pio_status = 5u | 0x08u | 0x10u | 0x20u | 0x80u, pio_pc = 5u | 0x08u | 0x20u | 0x40u, pio_isr = 6u | 0x20u, pio_osr = 7u | 0x10u | 0x20u, pio_exec_out = 7u | 0x08u | 0x20u | 0x40u | 0x80u }`

Enumeration of values to pass for source/destination args for instruction encoding functions.

4.1.15.12.3. Functions

`static uint pio_encode_delay (uint cycles)`

Encode just the delay slot bits of an instruction.

`static uint pio_encode_sideset (uint sideset_bit_count, uint value)`

Encode just the side set bits of an instruction (in non optional side set mode)

`static uint pio_encode_sideset_opt (uint sideset_bit_count, uint value)`

Encode just the side set bits of an instruction (in optional -`opt` side set mode)

`static uint pio_encode_jmp (uint addr)`

Encode an unconditional JMP instruction.

`static uint pio_encode_jmp_not_x (uint addr)`

Encode a conditional JMP if scratch X zero instruction.

`static uint pio_encode_jmp_x_dec (uint addr)`

Encode a conditional JMP if scratch X non-zero (and post-decrement X) instruction.

`static uint pio_encode_jmp_not_y (uint addr)`

Encode a conditional JMP if scratch Y zero instruction.

`static uint pio_encode_jmp_y_dec (uint addr)`

Encode a conditional JMP if scratch Y non-zero (and post-decrement Y) instruction.

`static uint pio_encode_jmp_x_ne_y (uint addr)`

Encode a conditional JMP if scratch X not equal scratch Y instruction.

`static uint pio_encode_jmp_pin (uint addr)`

Encode a conditional JMP if input pin high instruction.

`static uint pio_encode_jmp_not_osre (uint addr)`

Encode a conditional JMP if output shift register not empty instruction.

`static uint pio_encode_wait_gpio (bool polarity, uint gpio)`

Encode a WAIT for GPIO pin instruction.

`static uint pio_encode_wait_pin (bool polarity, uint pin)`

Encode a WAIT for pin instruction.

`static uint pio_encode_wait_irq (bool polarity, bool relative, uint irq)`

Encode a WAIT for IRQ instruction.

`static uint pio_encode_in (enum pio_src_dest src, uint count)`

Encode an IN instruction.

`static uint pio_encode_out (enum pio_src_dest dest, uint count)`

Encode an OUT instruction.

`static uint pio_encode_push (bool if_full, bool block)`

Encode a PUSH instruction.

`static uint pio_encode_pull (bool if_empty, bool block)`

Encode a PULL instruction.

`static uint pio_encode_mov (enum pio_src_dest dest, enum pio_src_dest src)`

Encode a MOV instruction.

`static uint pio_encode_mov_not (enum pio_src_dest dest, enum pio_src_dest src)`

Encode a MOV instruction with bit invert.

`static uint pio_encode_mov_reverse (enum pio_src_dest dest, enum pio_src_dest src)`

Encode a MOV instruction with bit reverse.

`static uint pio_encode_irq_set (bool relative, uint irq)`

Encode a IRQ SET instruction.

`static uint pio_encode_irq_wait (bool relative, uint irq)`

Encode a IRQ WAIT instruction.

`static uint pio_encode_irq_clear (bool relative, uint irq)`

Encode a IRQ CLEAR instruction.

`static uint pio_encode_set (enum pio_src_dest dest, uint value)`

Encode a SET instruction.

`static uint pio_encode_nop (void)`

Encode a NOP instruction.

4.1.15.12.4. Enumeration Type Documentation

pio_src_dest

`enum pio_src_dest`

Enumeration of values to pass for source/destination args for instruction encoding functions.

> **NOTE**
>
> Not all values are suitable for all functions. Validity is only checked in debug mode when `PARAM_ASSERTIONS_ENABLED_PIO_INSTRUCTIONS` is 1

4.1.15.12.5. Function Documentation

pio_encode_delay

`static uint pio_encode_delay (uint cycles) [inline], [static]`

Encode just the delay slot bits of an instruction.

> **NOTE**
>
> This function does not return a valid instruction encoding; instead it returns an encoding of the delay slot suitable for `OR`ing with the result of an encoding function for an actual instruction. Care should be taken when combining the results of this function with the results of pio_encode_sideset and pio_encode_sideset_opt as they share the same bits within the instruction encoding.

Parameters

`cycles` the number of cycles 0-31 (or less if side set is being used)

Returns

the delay slot bits to be ORed with an instruction encoding

pio_encode_in

`static uint pio_encode_in (enum pio_src_dest src, uint count) [inline], [static]`

Encode an IN instruction.

This is the equivalent of `IN <src>, <count>`

Parameters

`src`	The source to take data from
`count`	The number of bits 1-32

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_irq_clear

`static uint pio_encode_irq_clear (bool relative, uint irq) [inline], [static]`

Encode a IRQ CLEAR instruction.

This is the equivalent of `IRQ CLEAR <irq> <relative>`

Parameters

`relative`	true for a `IRQ CLEAR <irq> REL`, false for regular `IRQ CLEAR <irq>`
`irq`	the irq number 0-7

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_irq_set

`static uint pio_encode_irq_set (bool relative, uint irq) [inline], [static]`

Encode a IRQ SET instruction.

This is the equivalent of `IRQ SET <irq> <relative>`

Parameters

`relative`	true for a `IRQ SET <irq> REL`, false for regular `IRQ SET <irq>`
`irq`	the irq number 0-7

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_irq_wait

`static uint pio_encode_irq_wait (bool relative, uint irq) [inline], [static]`

Encode a IRQ WAIT instruction.

This is the equivalent of `IRQ WAIT <irq> <relative>`

Parameters

`relative`	true for a `IRQ WAIT <irq> REL`, false for regular `IRQ WAIT <irq>`
`irq`	the irq number 0-7

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_jmp

`static uint pio_encode_jmp (uint addr) [inline], [static]`

Encode an unconditional JMP instruction.

This is the equivalent of `JMP <addr>`

Parameters

`addr`	The target address 0-31 (an absolute address within the PIO instruction memory)

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_jmp_not_osre

`static uint pio_encode_jmp_not_osre (uint addr) [inline], [static]`

Encode a conditional JMP if output shift register not empty instruction.

This is the equivalent of `JMP !OSRE <addr>`

Parameters

`addr`	The target address 0-31 (an absolute address within the PIO instruction memory)

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_jmp_not_x

`static uint pio_encode_jmp_not_x (uint addr) [inline], [static]`

Encode a conditional JMP if scratch X zero instruction.

This is the equivalent of `JMP !X <addr>`

Parameters

`addr`	The target address 0-31 (an absolute address within the PIO instruction memory)

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_jmp_not_y

`static uint pio_encode_jmp_not_y (uint addr) [inline], [static]`

Encode a conditional JMP if scratch Y zero instruction.

This is the equivalent of `JMP !Y <addr>`

Parameters

`addr`	The target address 0-31 (an absolute address within the PIO instruction memory)

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_jmp_pin

`static uint pio_encode_jmp_pin (uint addr) [inline], [static]`

Encode a conditional JMP if input pin high instruction.

This is the equivalent of `JMP PIN <addr>`

Parameters

`addr`	The target address 0-31 (an absolute address within the PIO instruction memory)

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_jmp_x_dec

`static uint pio_encode_jmp_x_dec (uint addr) [inline], [static]`

Encode a conditional JMP if scratch X non-zero (and post-decrement X) instruction.

This is the equivalent of `JMP X-- <addr>`

Parameters

`addr`	The target address 0-31 (an absolute address within the PIO instruction memory)

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_jmp_x_ne_y

`static uint pio_encode_jmp_x_ne_y (uint addr) [inline], [static]`

Encode a conditional JMP if scratch X not equal scratch Y instruction.

This is the equivalent of `JMP X!=Y <addr>`

Parameters

`addr`	The target address 0-31 (an absolute address within the PIO instruction memory)

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_jmp_y_dec

`static uint pio_encode_jmp_y_dec (uint addr) [inline], [static]`

Encode a conditional JMP if scratch Y non-zero (and post-decrement Y) instruction.

This is the equivalent of `JMP Y-- <addr>`

Parameters

`addr`	The target address 0-31 (an absolute address within the PIO instruction memory)

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_mov

`static uint pio_encode_mov (enum pio_src_dest dest, enum pio_src_dest src) [inline], [static]`

Encode a MOV instruction.

This is the equivalent of `MOV <dest>, <src>`

Parameters

`dest`	The destination to write data to
`src`	The source to take data from

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_mov_not

`static uint pio_encode_mov_not (enum pio_src_dest dest, enum pio_src_dest src) [inline], [static]`

Encode a MOV instruction with bit invert.

This is the equivalent of `MOV <dest>, ~<src>`

Parameters

`dest`	The destination to write inverted data to
`src`	The source to take data from

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_mov_reverse

`static uint pio_encode_mov_reverse (enum pio_src_dest dest, enum pio_src_dest src) [inline], [static]`

Encode a MOV instruction with bit reverse.

This is the equivalent of `MOV <dest>, ::<src>`

Parameters

`dest`	The destination to write bit reversed data to
`src`	The source to take data from

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_nop

`static uint pio_encode_nop (void) [inline], [static]`

Encode a NOP instruction.

This is the equivalent of `NOP` which is itself encoded as `MOV y, y`

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_out

`static uint pio_encode_out (enum pio_src_dest dest, uint count) [inline], [static]`

Encode an OUT instruction.

This is the equivalent of `OUT <src>, <count>`

Parameters

`dest`	The destination to write data to
`count`	The number of bits 1-32

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_pull

`static uint pio_encode_pull (bool if_empty, bool block) [inline], [static]`

Encode a PULL instruction.

This is the equivalent of `PULL <if_empty>, <block>`

Parameters

`if_empty`	true for `PULL IF_EMPTY …`, false for `PULL …`
`block`	true for `PULL … BLOCK`, false for `PULL …`

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_push

`static uint pio_encode_push (bool if_full, bool block) [inline], [static]`

Encode a PUSH instruction.

This is the equivalent of `PUSH <if_full>, <block>`

Parameters

`if_full`	true for `PUSH IF_FULL …`, false for `PUSH …`
`block`	true for `PUSH … BLOCK`, false for `PUSH …`

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_set

```
static uint pio_encode_set (enum pio_src_dest dest, uint value) [inline], [static]
```

Encode a SET instruction.

This is the equivalent of `SET <dest>, <value>`

Parameters

`dest`	The destination to apply the value to
`value`	The value 0-31

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_sideset

```
static uint pio_encode_sideset (uint sideset_bit_count, uint value) [inline], [static]
```

Encode just the side set bits of an instruction (in non optional side set mode)

NOTE

> This function does not return a valid instruction encoding; instead it returns an encoding of the side set bits suitable for `OR`ing with the result of an encoding function for an actual instruction. Care should be taken when combining the results of this function with the results of pio_encode_delay as they share the same bits within the instruction encoding.

Parameters

`sideset_bit_count`	number of side set bits as would be specified via `.sideset` in pioasm
`value`	the value to sideset on the pins

Returns

the side set bits to be ORed with an instruction encoding

pio_encode_sideset_opt

```
static uint pio_encode_sideset_opt (uint sideset_bit_count, uint value) [inline], [static]
```

Encode just the side set bits of an instruction (in optional `-opt` side set mode)

> **NOTE**
>
> This function does not return a valid instruction encoding; instead it returns an encoding of the side set bits suitable for `OR`ing with the result of an encoding function for an actual instruction. Care should be taken when combining the results of this function with the results of pio_encode_delay as they share the same bits within the instruction encoding.

Parameters

`sideset_bit_count`	number of side set bits as would be specified via `.sideset <n> opt` in pioasm
`value`	the value to sideset on the pins

Returns

the side set bits to be ORed with an instruction encoding

pio_encode_wait_gpio

`static uint pio_encode_wait_gpio (bool polarity, uint gpio) [inline], [static]`

Encode a WAIT for GPIO pin instruction.

This is the equivalent of `WAIT <polarity> GPIO <gpio>`

Parameters

`polarity`	true for `WAIT 1`, false for `WAIT 0`
`gpio`	The real GPIO number 0-31

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_wait_irq

`static uint pio_encode_wait_irq (bool polarity, bool relative, uint irq) [inline], [static]`

Encode a WAIT for IRQ instruction.

This is the equivalent of `WAIT <polarity> IRQ <irq> <relative>`

Parameters

`polarity`	true for `WAIT 1`, false for `WAIT 0`
`relative`	true for a `WAIT IRQ <irq> REL`, false for regular `WAIT IRQ <irq>`
`irq`	the irq number 0-7

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_wait_pin

`static uint pio_encode_wait_pin (bool polarity, uint pin) [inline], [static]`

Encode a WAIT for pin instruction.

This is the equivalent of `WAIT <polarity> PIN <pin>`

Parameters

`polarity`	true for `WAIT 1`, false for `WAIT 0`
`pin`	The pin number 0-31 relative to the executing SM's input pin mapping

Returns

The instruction encoding with 0 delay and no side set value

See also

pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

4.1.16. hardware_pll

Phase Locked Loop control APIs.

4.1.16.1. Detailed Description

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 PLLs are calculated, please refer to the RP2040 datasheet.

4.1.16.2. Macros

- `#define PLL_RESET_NUM(pll)`

4.1.16.3. Functions

`void pll_init (PLL pll, uint ref_div, uint vco_freq, uint post_div1, uint post_div2)`

Initialise specified PLL.

`void pll_deinit (PLL pll)`

Release/uninitialise specified PLL.

4.1.16.4. Macro Definition Documentation

4.1.16.4.1. PLL_RESET_NUM

```
#define PLL_RESET_NUM(pll)
```

Returns the reset_num_t used to reset a given PLL instance.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.16.5. Function Documentation

4.1.16.5.1. 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

4.1.16.5.2. pll_init

`void pll_init (PLL pll, uint ref_div, uint vco_freq, uint post_div1, uint 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

4.1.17. hardware_powman RP2350

Power Management API.

4.1.17.1. Enumerations

`enum powman_power_domains { POWMAN_POWER_DOMAIN_SRAM_BANK1 = 0, POWMAN_POWER_DOMAIN_SRAM_BANK0 = 1, POWMAN_POWER_DOMAIN_XIP_CACHE = 2, POWMAN_POWER_DOMAIN_SWITCHED_CORE = 3, POWMAN_POWER_DOMAIN_COUNT = 4 }`

Power domains of powman.

4.1.17.2. Functions

`void powman_timer_set_1khz_tick_source_lposc (void)`

Use the ~32KHz low power oscillator as the powman timer source.

`void powman_timer_set_1khz_tick_source_lposc_with_hz (uint32_t lposc_freq_hz)`

Use the low power oscillator (specifying frequency) as the powman timer source.

`void powman_timer_set_1khz_tick_source_xosc (void)`

Use the crystal oscillator as the powman timer source.

`void powman_timer_set_1khz_tick_source_xosc_with_hz (uint32_t xosc_freq_hz)`

Use the crystal oscillator as the powman timer source.

`void powman_timer_set_1khz_tick_source_gpio (uint32_t gpio)`

Use a 1KHz external tick as the powman timer source.

`void powman_timer_enable_gpio_1hz_sync (uint32_t gpio)`

Use a 1Hz external signal as the powman timer source for seconds only.

`void powman_timer_disable_gpio_1hz_sync (void)`

Stop using 1Hz external signal as the powman timer source for seconds.

`uint64_t powman_timer_get_ms (void)`

Returns current time in ms.

`void powman_timer_set_ms (uint64_t time_ms)`

Set current time in ms.

`void powman_timer_enable_alarm_at_ms (uint64_t alarm_time_ms)`

Set an alarm at an absolute time in ms.

`void powman_timer_disable_alarm (void)`

Disable the alarm.

`static void powman_set_bits (volatile uint32_t *reg, uint32_t bits)`

hw_set_bits helper function

`static void powman_clear_bits (volatile uint32_t *reg, uint32_t bits)`

hw_clear_bits helper function

`static bool powman_timer_is_running (void)`

Determine if the powman timer is running.

`static void powman_timer_stop (void)`

Stop the powman timer.

`static void powman_timer_start (void)`

Start the powman timer.

`static void powman_clear_alarm (void)`

Clears the powman alarm.

`powman_power_state powman_get_power_state (void)`

Get the current power state.

`int powman_set_power_state (powman_power_state state)`

Set the power state.

`static powman_power_state powman_power_state_with_domain_on (powman_power_state orig, enum powman_power_domains domain)`

Helper function modify a powman_power_state to turn a domain on.

`static powman_power_state powman_power_state_with_domain_off (powman_power_state orig, enum powman_power_domains domain)`

Helper function modify a powman_power_state to turn a domain off.

`static bool powman_power_state_is_domain_on (powman_power_state state, enum powman_power_domains domain)`

Helper function to check if a domain is on in a given powman_power_state.

`void powman_enable_alarm_wakeup_at_ms (uint64_t alarm_time_ms)`

Wake up from an alarm at a given time.

`void powman_enable_gpio_wakeup (uint gpio_wakeup_num, uint32_t gpio, bool edge, bool high)`

Wake up from a gpio.

`void powman_disable_alarm_wakeup (void)`

Disable waking up from alarm.

`void powman_disable_gpio_wakeup (uint gpio_wakeup_num)`

Disable wake up from a gpio.

`void powman_disable_all_wakeups (void)`

Disable all wakeup sources.

`bool powman_configure_wakeup_state (powman_power_state sleep_state, powman_power_state wakeup_state)`

Configure sleep state and wakeup state.

`static void powman_set_debug_power_request_ignored (bool ignored)`

Ignore wake up when the debugger is attached.

4.1.17.3. Enumeration Type Documentation

4.1.17.3.1. powman_power_domains

`enum powman_power_domains`

Power domains of powman.

Table 26. Enumerator

POWMAN_POWER_DOMAIN_SRAM_BANK1	bank1 includes the top 256K of sram plus sram 8 and 9 (scratch x and scratch y)
POWMAN_POWER_DOMAIN_SRAM_BANK0	bank0 is bottom 256K of sSRAM
POWMAN_POWER_DOMAIN_XIP_CACHE	XIP cache is 2x8K instances.
POWMAN_POWER_DOMAIN_SWITCHED_CORE	Switched core logic (processors, busfabric, peris etc)

4.1.17.4. Function Documentation

4.1.17.4.1. powman_clear_alarm

`static void powman_clear_alarm (void) [inline], [static]`

Clears the powman alarm.

Note, the alarm must be disabled (see powman_timer_disable_alarm) before clearing the alarm, as the alarm fires if the time is greater than equal to the target, so once the time has passed the alarm will always fire while enabled.

4.1.17.4.2. powman_clear_bits

`static void powman_clear_bits (volatile uint32_t * reg, uint32_t bits) [inline], [static]`

hw_clear_bits helper function

Powman needs a password for writes, to prevent accidentally writing to it. This function implements hw_clear_bits with an appropriate password.

Parameters

`reg`	register to clear
`bits`	bits of register to clear

4.1.17.4.3. powman_configure_wakeup_state

`bool powman_configure_wakeup_state (powman_power_state sleep_state, powman_power_state wakeup_state)`

Configure sleep state and wakeup state.

Parameters

`sleep_state`	power state powman will go to when sleeping, used to validate the wakeup state
`wakeup_state`	power state powman will go to when waking up. Note switched core and xip always power up. SRAM bank0 and bank1 can be left powered off

Returns

true if the state is valid, false if not

4.1.17.4.4. powman_disable_alarm_wakeup

`void powman_disable_alarm_wakeup (void)`

Disable waking up from alarm.

4.1.17.4.5. powman_disable_all_wakeups

`void powman_disable_all_wakeups (void)`

Disable all wakeup sources.

4.1.17.4.6. powman_disable_gpio_wakeup

`void powman_disable_gpio_wakeup (uint gpio_wakeup_num)`

Disable wake up from a gpio.

Parameters

`gpio_wakeup_num`	hardware wakeup instance to use (0-3)

4.1.17.4.7. powman_enable_alarm_wakeup_at_ms

`void powman_enable_alarm_wakeup_at_ms (uint64_t alarm_time_ms)`

Wake up from an alarm at a given time.

Parameters

`alarm_time_ms`	time to wake up in ms

4.1.17.4.8. powman_enable_gpio_wakeup

`void powman_enable_gpio_wakeup (uint gpio_wakeup_num, uint32_t gpio, bool edge, bool high)`

Wake up from a gpio.

Parameters

`gpio_wakeup_num`	hardware wakeup instance to use (0-3)
`gpio`	gpio to wake up from (0-47)
`edge`	true for edge sensitive, false for level sensitive
`high`	true for active high, false active low

4.1.17.4.9. powman_get_power_state

`powman_power_state powman_get_power_state (void)`

Get the current power state.

4.1.17.4.10. powman_power_state_is_domain_on

`static bool powman_power_state_is_domain_on (powman_power_state state, enum powman_power_domains domain) [inline], [static]`

Helper function to check if a domain is on in a given powman_power_state.

Parameters

`state`	powman_power_state
`domain`	domain to check is on

4.1.17.4.11. powman_power_state_with_domain_off

`static powman_power_state powman_power_state_with_domain_off (powman_power_state orig, enum powman_power_domains domain) [inline], [static]`

Helper function modify a powman_power_state to turn a domain off.

Parameters

`orig`	original state
`domain`	domain to turn off

4.1.17.4.12. powman_power_state_with_domain_on

`static powman_power_state powman_power_state_with_domain_on (powman_power_state orig, enum powman_power_domains domain) [inline], [static]`

Helper function modify a powman_power_state to turn a domain on.

Parameters

`orig`	original state
`domain`	domain to turn on

4.1.17.4.13. powman_set_bits

`static void powman_set_bits (volatile uint32_t * reg, uint32_t bits) [inline], [static]`

hw_set_bits helper function

Parameters

`reg`	register to set
`bits`	bits of register to set Powman needs a password for writes, to prevent accidentally writing to it. This function implements hw_set_bits with an appropriate password.

4.1.17.4.14. powman_set_debug_power_request_ignored

`static void powman_set_debug_power_request_ignored (bool ignored) [inline], [static]` Ignore wake up when the debugger is attached.

Typically, when a debugger is attached it will assert the pwrupreq signal. OpenOCD does not clear this signal, even when you quit. This means once you have attached a debugger powman will never go to sleep. This function lets you ignore the debugger pwrupreq which means you can go to sleep with a debugger attached. The debugger will error out if you go to turn off the switch core with it attached, as the processors have been powered off.

Parameters

`ignored`	should the debugger power up request be ignored

4.1.17.4.15. powman_set_power_state

```
int powman_set_power_state (powman_power_state state)
```

Set the power state.

Check the desired state is valid. Powman will go to the state if it is valid and there are no pending power up requests.

Note that if you are turning off the switched core then this function will never return as the processor will have been turned off at the end.

Parameters

`state`	the power state to go to

Returns

PICO_OK if the state is valid. Misc PICO_ERRORs are returned if not

4.1.17.4.16. powman_timer_disable_alarm

```
void powman_timer_disable_alarm (void)
```

Disable the alarm.

Once an alarm has fired it must be disabled to stop firing as the alarm comparison is alarm = alarm_time >= current_time

4.1.17.4.17. powman_timer_disable_gpio_1hz_sync

```
void powman_timer_disable_gpio_1hz_sync (void)
```

Stop using 1Hz external signal as the powman timer source for seconds.

4.1.17.4.18. powman_timer_enable_alarm_at_ms

```
void powman_timer_enable_alarm_at_ms (uint64_t alarm_time_ms)
```

Set an alarm at an absolute time in ms.

Note, the timer is stopped and then restarted as part of this function. This only controls the alarm if you want to use the alarm to wake up powman then you should use powman_enable_alarm_wakeup_at_ms

Parameters

`alarm_time_ms`	time at which the alarm will fire

4.1.17.4.19. powman_timer_enable_gpio_1hz_sync

`void powman_timer_enable_gpio_1hz_sync (uint32_t gpio)`

Use a 1Hz external signal as the powman timer source for seconds only.

Use a 1hz sync signal, such as from a gps for the seconds component of the timer. The milliseconds will still come from another configured source such as xosc or lposc

Parameters

`gpio`	the gpio to use. must be 12, 14, 20, 22

4.1.17.4.20. powman_timer_get_ms

`uint64_t powman_timer_get_ms (void)`

Returns current time in ms.

4.1.17.4.21. powman_timer_is_running

`static bool powman_timer_is_running (void) [inline], [static]`

Determine if the powman timer is running.

4.1.17.4.22. powman_timer_set_1khz_tick_source_gpio

`void powman_timer_set_1khz_tick_source_gpio (uint32_t gpio)`

Use a 1KHz external tick as the powman timer source.

Parameters

`gpio`	the gpio to use. must be 12, 14, 20, 22

4.1.17.4.23. powman_timer_set_1khz_tick_source_lposc

`void powman_timer_set_1khz_tick_source_lposc (void)`

Use the ~32KHz low power oscillator as the powman timer source.

4.1.17.4.24. powman_timer_set_1khz_tick_source_lposc_with_hz

`void powman_timer_set_1khz_tick_source_lposc_with_hz (uint32_t lposc_freq_hz)`

Use the low power oscillator (specifying frequency) as the powman timer source.

Parameters

`lposc_freq_hz`	specify an exact lposc freq to trim it

4.1.17.4.25. powman_timer_set_1khz_tick_source_xosc

`void powman_timer_set_1khz_tick_source_xosc (void)`

Use the crystal oscillator as the powman timer source.

4.1.17.4.26. powman_timer_set_1khz_tick_source_xosc_with_hz

`void powman_timer_set_1khz_tick_source_xosc_with_hz (uint32_t xosc_freq_hz)`

Use the crystal oscillator as the powman timer source.

Parameters

`xosc_freq_hz`	specify a crystal frequency

4.1.17.4.27. powman_timer_set_ms

`void powman_timer_set_ms (uint64_t time_ms)`

Set current time in ms.

Parameters

`time_ms`	Current time in ms

4.1.17.4.28. powman_timer_start

`static void powman_timer_start (void) [inline], [static]`

Start the powman timer.

4.1.17.4.29. powman_timer_stop

`static void powman_timer_stop (void) [inline], [static]`

Stop the powman timer.

4.1.18. hardware_pwm

Hardware Pulse Width Modulation (PWM) API.

4.1.18.1. Detailed Description

The RP2040 PWM block has 8 identical slices, the RP2350 has 12. 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 16/24 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.
}
```

4.1.18.2. Macros

- `#define PWM_DREQ_NUM(slice_num)`
- `#define PWM_GPIO_SLICE_NUM(gpio)`
- `#define PWM_DEFAULT_IRQ_NUM()`

4.1.18.3. Enumerations

`enum pwm_clkdiv_mode { PWM_DIV_FREE_RUNNING = 0, PWM_DIV_B_HIGH = 1, PWM_DIV_B_RISING = 2, PWM_DIV_B_FALLING = 3 }`

PWM Divider mode settings.

4.1.18.4. Functions

`static uint pwm_gpio_to_slice_num (uint gpio)`

Determine the PWM slice that is attached to the specified GPIO.

`static uint pwm_gpio_to_channel (uint gpio)`

Determine the PWM channel that is attached to the specified GPIO.

`static void pwm_config_set_phase_correct (pwm_config *c, bool phase_correct)`

Set phase correction in a PWM configuration.

`static void pwm_config_set_clkdiv (pwm_config *c, float div)`

Set PWM clock divider in a PWM configuration.

`static void pwm_config_set_clkdiv_int_frac4 (pwm_config *c, uint32_t div_int, uint8_t div_frac4)`

Set PWM clock divider in a PWM configuration using an 8:4 fractional value.

`static void pwm_config_set_clkdiv_int (pwm_config *c, uint32_t div_int)`

Set PWM clock divider in a PWM configuration.

`static void pwm_config_set_clkdiv_mode (pwm_config *c, enum pwm_clkdiv_mode mode)`

Set PWM counting mode in a PWM configuration.

`static void pwm_config_set_output_polarity (pwm_config *c, bool a, bool b)`

Set output polarity in a PWM configuration.

`static void pwm_config_set_wrap (pwm_config *c, uint16_t wrap)`

Set PWM counter wrap value in a PWM configuration.

`static void pwm_init (uint slice_num, pwm_config *c, bool start)`

Initialise a PWM with settings from a configuration object.

`static pwm_config pwm_get_default_config (void)`

Get a set of default values for PWM configuration.

`static void pwm_set_wrap (uint slice_num, uint16_t wrap)`

Set the current PWM counter wrap value.

`static void pwm_set_chan_level (uint slice_num, uint chan, uint16_t level)`

Set the current PWM counter compare value for one channel.

`static void pwm_set_both_levels (uint slice_num, uint16_t level_a, uint16_t level_b)`

Set PWM counter compare values.

`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.

`static uint16_t pwm_get_counter (uint slice_num)`

Get PWM counter.

`static void pwm_set_counter (uint slice_num, uint16_t c)`

Set PWM counter.

`static void pwm_advance_count (uint slice_num)`

Advance PWM count.

`static void pwm_retard_count (uint slice_num)`

Retard PWM count.

`static void pwm_set_clkdiv_int_frac4 (uint slice_num, uint8_t div_int, uint8_t div_frac4)`

Set PWM clock divider using an 8:4 fractional value.

`static void pwm_set_clkdiv (uint slice_num, float divider)`

Set PWM clock divider.

`static void pwm_set_output_polarity (uint slice_num, bool a, bool b)`

Set PWM output polarity.

`static void pwm_set_clkdiv_mode (uint slice_num, enum pwm_clkdiv_mode mode)`

Set PWM divider mode.

`static void pwm_set_phase_correct (uint slice_num, bool phase_correct)`

Set PWM phase correct on/off.

`static void pwm_set_enabled (uint slice_num, bool enabled)`

Enable/Disable PWM.

`static void pwm_set_mask_enabled (uint32_t mask)`

Enable/Disable multiple PWM slices simultaneously.

`static void pwm_set_irq_enabled (uint slice_num, bool enabled)`

Enable PWM instance interrupt via the default PWM IRQ (PWM_IRQ_WRAP_0 on RP2350) `static void pwm_set_irq0_enabled (uint slice_num, bool enabled)`

Enable PWM instance interrupt via PWM_IRQ_WRAP_0.

`static void pwm_irqn_set_slice_enabled (uint irq_index, uint slice_num, bool enabled)`

Enable PWM instance interrupt via either PWM_IRQ_WRAP_0 or PWM_IRQ_WRAP_1.

`static void pwm_set_irq_mask_enabled (uint32_t slice_mask, bool enabled)`

Enable multiple PWM instance interrupts via the default PWM IRQ (PWM_IRQ_WRAP_0 on RP2350)

`static void pwm_set_irq0_mask_enabled (uint32_t slice_mask, bool enabled)`

Enable multiple PWM instance interrupts via PWM_IRQ_WRAP_0.

`static void pwm_irqn_set_slice_mask_enabled (uint irq_index, uint slice_mask, bool enabled)`

Enable PWM instance interrupts via either PWM_IRQ_WRAP_0 or PWM_IRQ_WRAP_1.

`static void pwm_clear_irq (uint slice_num)`

Clear a single PWM channel interrupt.

`static uint32_t pwm_get_irq_status_mask (void)`

Get PWM interrupt status, raw for the default PWM IRQ (PWM_IRQ_WRAP_0 on RP2350)

`static uint32_t pwm_get_irq0_status_mask (void)`

Get PWM interrupt status, raw for the PWM_IRQ_WRAP_0.

`static uint32_t pwm_irqn_get_status_mask (uint irq_index)`

Get PWM interrupt status, raw for either PWM_IRQ_WRAP_0 or PWM_IRQ_WRAP_1.

`static void pwm_force_irq (uint slice_num)`

Force PWM interrupt for the default PWM IRQ (PWM_IRQ_WRAP_0 on RP2350)

`static void pwm_force_irq0 (uint slice_num)`

Force PWM interrupt via PWM_IRQ_WRAP_0.

`static void pwm_irqn_force (uint irq_index, uint slice_num)`

Force PWM interrupt via PWM_IRQ_WRAP_0 or PWM_IRQ_WRAP_1.

`static uint pwm_get_dreq (uint slice_num)`

Return the DREQ to use for pacing transfers to a particular PWM slice.

4.1.18.5. Macro Definition Documentation

4.1.18.5.1. PWM_DREQ_NUM

```
#define PWM_DREQ_NUM(slice_num)
```

Returns the dreq_num_t used for pacing DMA transfers for a given PWM slice.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.18.5.2. PWM_GPIO_SLICE_NUM

```
#define PWM_GPIO_SLICE_NUM(gpio)
```

Returns the PWM slice number for a given GPIO number.

4.1.18.5.3. PWM_DEFAULT_IRQ_NUM

`#define PWM_DEFAULT_IRQ_NUM()`

Returns the irq_num_t for the default PWM IRQ.

On RP2040, there is only one PWM irq: PWM_IRQ_WRAP

On RP2350 this returns to PWM_IRQ_WRAP0

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.18.6. Enumeration Type Documentation

4.1.18.6.1. pwm_clkdiv_mode

`enum pwm_clkdiv_mode`

PWM Divider mode settings.

Table 27. Enumerator

PWM_DIV_FREE_RUNNING	Free-running counting at rate dictated by fractional divider.
PWM_DIV_B_HIGH	Fractional divider is gated by the PWM B pin.
PWM_DIV_B_RISING	Fractional divider advances with each rising edge of the PWM B pin.
PWM_DIV_B_FALLING	Fractional divider advances with each falling edge of the PWM B pin.

4.1.18.7. Function Documentation

4.1.18.7.1. pwm_advance_count

`static void pwm_advance_count (uint slice_num) [inline], [static]`

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

4.1.18.7.2. pwm_clear_irq

`static void pwm_clear_irq (uint slice_num) [inline], [static]`

Clear a single PWM channel interrupt.

Parameters

`slice_num`	PWM slice number

4.1.18.7.3. pwm_config_set_clkdiv

`static void pwm_config_set_clkdiv (pwm_config * c, float div) [inline], [static]` Set PWM clock divider in a PWM configuration.

Parameters

`c`	PWM configuration struct to modify
`div`	Value to divide counting rate by. Must be greater than or equal to 1.

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.

4.1.18.7.4. pwm_config_set_clkdiv_int

`static void pwm_config_set_clkdiv_int (pwm_config * c, uint32_t div_int) [inline], [static]`

Set PWM clock divider in a PWM configuration.

Parameters

`c`	PWM configuration struct to modify
`div_int`	Integer value to reduce counting rate by. Must be greater than or equal to 1 and less than 256.

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.

4.1.18.7.5. pwm_config_set_clkdiv_int_frac4

`static void pwm_config_set_clkdiv_int_frac4 (pwm_config * c, uint32_t div_int, uint8_t div_frac4) [inline], [static]`

Set PWM clock divider in a PWM configuration using an 8:4 fractional value.

Parameters

`c`	PWM configuration struct to modify
`div_int`	8 bit integer part of the clock divider. Must be greater than or equal to 1.
`div_frac4`	4 bit fractional part of the clock divider

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.

4.1.18.7.6. pwm_config_set_clkdiv_mode

`static void pwm_config_set_clkdiv_mode (pwm_config * c, enum pwm_clkdiv_mode mode) [inline], [static]`

Set PWM counting mode in a PWM configuration.

Parameters

`c`	PWM configuration struct to modify
`mode`	PWM divide/count mode

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.

4.1.18.7.7. pwm_config_set_output_polarity

`static void pwm_config_set_output_polarity (pwm_config * c, bool a, bool b) [inline], [static]`

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

4.1.18.7.8. pwm_config_set_phase_correct

`static void pwm_config_set_phase_correct (pwm_config * c, bool phase_correct) [inline], [static]`

Set phase correction in a PWM configuration.

Parameters

`c`	PWM configuration struct to modify
`phase_correct`	true to set phase correct modulation, false to set trailing edge

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.

4.1.18.7.9. pwm_config_set_wrap

`static void pwm_config_set_wrap (pwm_config * c, uint16_t wrap) [inline], [static]`

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

4.1.18.7.10. pwm_force_irq

`static void pwm_force_irq (uint slice_num) [inline], [static]`

Force PWM interrupt for the default PWM IRQ (PWM_IRQ_WRAP_0 on RP2350)

Parameters

`slice_num`	PWM slice number

4.1.18.7.11. pwm_force_irq0

`static void pwm_force_irq0 (uint slice_num) [inline], [static]`

Force PWM interrupt via PWM_IRQ_WRAP_0.

Parameters

`slice_num`	PWM slice number

4.1.18.7.12. pwm_get_counter

`static uint16_t pwm_get_counter (uint slice_num) [inline], [static]`

Get PWM counter.

Get current value of PWM counter

Parameters

`slice_num`	PWM slice number

Returns

Current value of the PWM counter

4.1.18.7.13. pwm_get_default_config

`static pwm_config pwm_get_default_config (void) [inline], [static]`

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.

4.1.18.7.14. pwm_get_dreq

`static uint pwm_get_dreq (uint slice_num) [inline], [static]`

Return the DREQ to use for pacing transfers to a particular PWM slice.

Parameters

`slice_num`	PWM slice number

4.1.18.7.15. pwm_get_irq0_status_mask

`static uint32_t pwm_get_irq0_status_mask (void) [inline], [static]`

Get PWM interrupt status, raw for the PWM_IRQ_WRAP_0.

Returns

Bitmask of all PWM interrupts currently set

4.1.18.7.16. pwm_get_irq_status_mask

`static uint32_t pwm_get_irq_status_mask (void) [inline], [static]`

Get PWM interrupt status, raw for the default PWM IRQ (PWM_IRQ_WRAP_0 on RP2350)

Returns

Bitmask of all PWM interrupts currently set

4.1.18.7.17. pwm_gpio_to_channel

`static uint pwm_gpio_to_channel (uint gpio) [inline], [static]`

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.

4.1.18.7.18. pwm_gpio_to_slice_num

`static uint pwm_gpio_to_slice_num (uint gpio) [inline], [static]`

Determine the PWM slice that is attached to the specified GPIO.

Returns

The PWM slice number that controls the specified GPIO.

4.1.18.7.19. pwm_init

`static void pwm_init (uint slice_num, pwm_config * c, bool start) [inline], [static]`

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()

4.1.18.7.20. pwm_irqn_force

`static void pwm_irqn_force (uint irq_index, uint slice_num) [inline], [static]`

Force PWM interrupt via PWM_IRQ_WRAP_0 or PWM_IRQ_WRAP_1.

Parameters

`irq_index`	the IRQ index; either 0 or 1 for PWM_IRQ_WRAP_0 or PWM_IRQ_WRAP_1
`slice_num`	PWM slice number

4.1.18.7.21. pwm_irqn_get_status_mask

`static uint32_t pwm_irqn_get_status_mask (uint irq_index) [inline], [static]`

Get PWM interrupt status, raw for either PWM_IRQ_WRAP_0 or PWM_IRQ_WRAP_1.

Parameters

`irq_index`	the IRQ index; either 0 or 1 for PWM_IRQ_WRAP_0 or PWM_IRQ_WRAP_1

Returns

Bitmask of all PWM interrupts currently set

4.1.18.7.22. pwm_irqn_set_slice_enabled

`static void pwm_irqn_set_slice_enabled (uint irq_index, uint slice_num, bool enabled) [inline], [static]`

Enable PWM instance interrupt via either PWM_IRQ_WRAP_0 or PWM_IRQ_WRAP_1.

Used to enable a single PWM instance interrupt.

Note there is only one PWM_IRQ_WRAP on RP2040.

Parameters

`irq_index`	the IRQ index; either 0 or 1 for PWM_IRQ_WRAP_0 or PWM_IRQ_WRAP_1
`slice_num`	PWM block to enable/disable
`enabled`	true to enable, false to disable

4.1.18.7.23. pwm_irqn_set_slice_mask_enabled

`static void pwm_irqn_set_slice_mask_enabled (uint irq_index, uint slice_mask, bool enabled) [inline], [static]`

Enable PWM instance interrupts via either PWM_IRQ_WRAP_0 or PWM_IRQ_WRAP_1.

Used to enable a single PWM instance interrupt.

Note there is only one PWM_IRQ_WRAP on RP2040.

Parameters

`irq_index`	the IRQ index; either 0 or 1 for PWM_IRQ_WRAP_0 or PWM_IRQ_WRAP_1
`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

4.1.18.7.24. pwm_retard_count

`static void pwm_retard_count (uint slice_num) [inline], [static]`

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

4.1.18.7.25. pwm_set_both_levels

`static void pwm_set_both_levels (uint slice_num, uint16_t level_a, uint16_t level_b) [inline], [static]`

Set PWM counter compare values.

Set the value of the PWM counter compare values, A and B.

The counter compare register is double-buffered in hardware. This means that, when the PWM is running, a write to the counter compare values does not take effect until the next time the PWM slice wraps (or, in phase-correct mode, the next time the slice reaches 0). If the PWM is not running, the write is latched in immediately.

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

4.1.18.7.26. pwm_set_chan_level

`static void pwm_set_chan_level (uint slice_num, uint chan, uint16_t level) [inline], [static]`

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.

The counter compare register is double-buffered in hardware. This means that, when the PWM is running, a write to the counter compare values does not take effect until the next time the PWM slice wraps (or, in phase-correct mode, the next time the slice reaches 0). If the PWM is not running, the write is latched in immediately.

Parameters

`slice_num`	PWM slice number
`chan`	Which channel to update. 0 for A, 1 for B.
`level`	new level for the selected output

4.1.18.7.27. pwm_set_clkdiv

`static void pwm_set_clkdiv (uint slice_num, float divider) [inline], [static]`

Set PWM clock divider.

Set the clock divider. Counter increment will be on sysclock divided by this value, taking into account the gating.

Parameters

`slice_num`	PWM slice number
`divider`	Floating point clock divider, 1.f ⇐ value < 256.f

4.1.18.7.28. pwm_set_clkdiv_int_frac4

`static void pwm_set_clkdiv_int_frac4 (uint slice_num, uint8_t div_int, uint8_t div_frac4) [inline], [static]`

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 into account the gating.

Parameters

`slice_num`	PWM slice number
`div_int`	8 bit integer part of the clock divider
`div_frac4`	4 bit fractional part of the clock divider

4.1.18.7.29. pwm_set_clkdiv_mode

`static void pwm_set_clkdiv_mode (uint slice_num, enum pwm_clkdiv_mode mode) [inline], [static]`

Set PWM divider mode.

Parameters

`slice_num`	PWM slice number
`mode`	Required divider mode

4.1.18.7.30. pwm_set_counter

`static void pwm_set_counter (uint slice_num, uint16_t c) [inline], [static]`

Set PWM counter.

Set the value of the PWM counter

Parameters

`slice_num`	PWM slice number
`c`	Value to set the PWM counter to

4.1.18.7.31. pwm_set_enabled

`static void pwm_set_enabled (uint slice_num, bool enabled) [inline], [static]`

Enable/Disable PWM.

When a PWM is disabled, it halts its counter, and the output pins are left high or low depending on exactly when the counter is halted. When re-enabled the PWM resumes immediately from where it left off.

If the PWM's output pins need to be low when halted:

- The counter compare can be set to zero whilst the PWM is enabled, and then the PWM disabled once both pins are seen to be low
- The GPIO output overrides can be used to force the actual pins low
- The PWM can be run for one cycle (i.e. enabled then immediately disabled) with a TOP of 0, count of 0 and counter compare of 0, to force the pins low when the PWM has already been halted. The same method can be used with a counter compare value of 1 to force a pin high.

Note that, when disabled, the PWM can still be advanced one count at a time by pulsing the PH_ADV bit in its CSR. The output pins transition as though the PWM were enabled.

Parameters

`slice_num`	PWM slice number
`enabled`	true to enable the specified PWM, false to disable.

4.1.18.7.32. pwm_set_gpio_level

`static void pwm_set_gpio_level (uint gpio, uint16_t level) [inline], [static]`

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).

The counter compare register is double-buffered in hardware. This means that, when the PWM is running, a write to the counter compare values does not take effect until the next time the PWM slice wraps (or, in phase-correct mode, the next time the slice reaches 0). If the PWM is not running, the write is latched in immediately.

Parameters

`gpio`	GPIO to set level of
`level`	PWM level for this GPIO

4.1.18.7.33. pwm_set_irq0_enabled

```
static void pwm_set_irq0_enabled (uint slice_num, bool enabled) [inline], [static]
```

Enable PWM instance interrupt via PWM_IRQ_WRAP_0.

Used to enable a single PWM instance interrupt.

Parameters

`slice_num`	PWM block to enable/disable
`enabled`	true to enable, false to disable

4.1.18.7.34. pwm_set_irq0_mask_enabled

```
static void pwm_set_irq0_mask_enabled (uint32_t slice_mask, bool enabled) [inline], [static]
```

Enable multiple PWM instance interrupts via PWM_IRQ_WRAP_0.

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

4.1.18.7.35. pwm_set_irq_enabled

```
static void pwm_set_irq_enabled (uint slice_num, bool enabled) [inline], [static]
```

Enable PWM instance interrupt via the default PWM IRQ (PWM_IRQ_WRAP_0 on RP2350)

Used to enable a single PWM instance interrupt.

Note there is only one PWM_IRQ_WRAP on RP2040.

Parameters

`slice_num`	PWM block to enable/disable
`enabled`	true to enable, false to disable

4.1.18.7.36. pwm_set_irq_mask_enabled

```
static void pwm_set_irq_mask_enabled (uint32_t slice_mask, bool enabled) [inline], [static]
```

Enable multiple PWM instance interrupts via the default PWM IRQ (PWM_IRQ_WRAP_0 on RP2350)

Use this to enable multiple PWM interrupts at once.

Note there is only one PWM_IRQ_WRAP on RP2040.

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

4.1.18.7.37. pwm_set_mask_enabled

```
static void pwm_set_mask_enabled (uint32_t mask) [inline], [static]
```

Enable/Disable multiple PWM slices simultaneously.

Parameters

`mask`	Bitmap of PWMs to enable/disable. Bits 0 to 7 enable slices 0-7 respectively

4.1.18.7.38. pwm_set_output_polarity

`static void pwm_set_output_polarity (uint slice_num, bool a, bool b) [inline], [static]`

Set PWM output polarity.

Parameters

`slice_num`	PWM slice number
`a`	true to invert output A
`b`	true to invert output B

4.1.18.7.39. pwm_set_phase_correct

`static void pwm_set_phase_correct (uint slice_num, bool phase_correct) [inline], [static]`

Set PWM phase correct on/off.

Parameters

`slice_num`	PWM slice number
`phase_correct`	true to set phase correct modulation, false to set trailing edge

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.

4.1.18.7.40. pwm_set_wrap

`static void pwm_set_wrap (uint slice_num, uint16_t wrap) [inline], [static]`

Set the current PWM counter wrap value.

Set the highest value the counter will reach before returning to 0. Also known as TOP.

The counter wrap value is double-buffered in hardware. This means that, when the PWM is running, a write to the counter wrap value does not take effect until after the next time the PWM slice wraps (or, in phase-correct mode, the next time the slice reaches 0). If the PWM is not running, the write is latched in immediately.

Parameters

`slice_num`	PWM slice number
`wrap`	Value to set wrap to

4.1.19. hardware_resets

Hardware Reset API.

4.1.19.1. Detailed Description

The reset controller allows software control of the resets to all of the peripherals that are not critical to boot the processor in the RP-series microcontroller.

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"

int main() {
    stdio_init_all();

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

    // Put the PWM block into reset
    reset_block_num(RESET_PWM);

    // And bring it out
```

```
    unreset_block_num_wait_blocking(RESET_PWM);

    // Put the PWM and ADC block into reset
    reset_block_mask((1u << RESET_PWM) | (1u << RESET_ADC));

    // Wait for both to come out of reset
    unreset_block_mask_wait_blocking((1u << RESET_PWM) | (1u << RESET_ADC));

    return 0;
}
```

4.1.19.2. Typedefs

`typedef enum reset_num_rp2040 reset_num_t` RP2040

Resettable component numbers on RP2040 (used as typedef reset_num_t)

`typedef enum reset_num_rp2350 reset_num_t` RP2350

Resettable component numbers on RP2350 (used as typedef reset_num_t)

4.1.19.3. Enumerations

`enum reset_num_rp2040 { RESET_ADC = 0, RESET_BUSCTRL = 1, RESET_DMA = 2, RESET_I2C0 = 3, RESET_I2C1 = 4, RESET_IO_BANK0 = 5, RESET_IO_QSPI = 6, RESET_JTAG = 7, RESET_PADS_BANK0 = 8, RESET_PADS_QSPI = 9, RESET_PIO0 = 10, RESET_PIO1 = 11, RESET_PLL_SYS = 12, RESET_PLL_USB = 13, RESET_PWM = 14, RESET_RTC = 15, RESET_SPI0 = 16, RESET_SPI1 = 17, RESET_SYSCFG = 18, RESET_SYSINFO = 19, RESET_TBMAN = 20, RESET_TIMER = 21, RESET_UART0 = 22, RESET_UART1 = 23, RESET_USBCTRL = 24, RESET_COUNT }` RP2040

Resettable component numbers on RP2040 (used as typedef reset_num_t)

`enum reset_num_rp2350 { RESET_ADC = 0, RESET_BUSCTRL = 1, RESET_DMA = 2, RESET_HSTX = 3, RESET_I2C0 = 4, RESET_I2C1 = 5, RESET_IO_BANK0 = 6, RESET_IO_QSPI = 7, RESET_JTAG = 8, RESET_PADS_BANK0 = 9, RESET_PADS_QSPI = 10, RESET_PIO0 = 11, RESET_PIO1 = 12, RESET_PIO2 = 13, RESET_PLL_SYS = 14, RESET_PLL_USB = 15, RESET_PWM = 16, RESET_SHA256 = 17, RESET_SPI0 = 18, RESET_SPI1 = 19, RESET_SYSCFG = 20, RESET_SYSINFO = 21, RESET_TBMAN = 22, RESET_TIMER0 = 23, RESET_TIMER1 = 24, RESET_TRNG = 25, RESET_UART0 = 26, RESET_UART1 = 27, RESET_USBCTRL = 28, RESET_COUNT }` RP2350

Resettable component numbers on RP2350 (used as typedef reset_num_t)

4.1.19.4. Functions

`static __force_inline void reset_block_mask (uint32_t bits)`

Reset the specified HW blocks.

`static __force_inline void unreset_block_mask (uint32_t bits)`

bring specified HW blocks out of reset

`static __force_inline void unreset_block_mask_wait_blocking (uint32_t bits)`

Bring specified HW blocks out of reset and wait for completion.

`static void reset_block_num (uint32_t block_num)`

Reset the specified HW block.

`static void unreset_block_num (uint block_num)`

bring specified HW block out of reset

`static void unreset_block_num_wait_blocking (uint block_num)`

Bring specified HW block out of reset and wait for completion.

```
static void reset_unreset_block_num_wait_blocking (uint block_num)
```

Reset the specified HW block, and then bring at back out of reset and wait for completion.

4.1.19.5. Typedef Documentation

4.1.19.5.1. reset_num_t RP2040

```
typedef enum reset_num_rp2040 reset_num_t
```

Resettable component numbers on RP2040 (used as typedef reset_num_t)

4.1.19.5.2. reset_num_t RP2350

```
typedef enum reset_num_rp2350 reset_num_t
```

Resettable component numbers on RP2350 (used as typedef reset_num_t)

4.1.19.6. Enumeration Type Documentation

4.1.19.6.1. reset_num_rp2040 RP2040

```
enum reset_num_rp2040
```

Resettable component numbers on RP2040 (used as typedef reset_num_t)

Table 28. Enumerator

RESET_ADC	Select ADC to be reset.
RESET_BUSCTRL	Select BUSCTRL to be reset.
RESET_DMA	Select DMA to be reset.
RESET_I2C0	Select I2C0 to be reset.
RESET_I2C1	Select I2C1 to be reset.
RESET_IO_BANK0	Select IO_BANK0 to be reset.
RESET_IO_QSPI	Select IO_QSPI to be reset.
RESET_JTAG	Select JTAG to be reset.
RESET_PADS_BANK0	Select PADS_BANK0 to be reset.
RESET_PADS_QSPI	Select PADS_QSPI to be reset.
RESET_PIO0	Select PIO0 to be reset.
RESET_PIO1	Select PIO1 to be reset.
RESET_PLL_SYS	Select PLL_SYS to be reset.
RESET_PLL_USB	Select PLL_USB to be reset.
RESET_PWM	Select PWM to be reset.
RESET_RTC	Select RTC to be reset.
RESET_SPI0	Select SPI0 to be reset.
RESET_SPI1	Select SPI1 to be reset.

RESET_SYSCFG	Select SYSCFG to be reset.
RESET_SYSINFO	Select SYSINFO to be reset.
RESET_TBMAN	Select TBMAN to be reset.
RESET_TIMER	Select TIMER to be reset.
RESET_UART0	Select UART0 to be reset.
RESET_UART1	Select UART1 to be reset.
RESET_USBCTRL	Select USBCTRL to be reset.

4.1.19.6.2. reset_num_rp2350 RP2350

`enum reset_num_rp2350`

Resettable component numbers on RP2350 (used as typedef reset_num_t)

Table 29. Enumerator

RESET_ADC	Select ADC to be reset.
RESET_BUSCTRL	Select BUSCTRL to be reset.
RESET_DMA	Select DMA to be reset.
RESET_HSTX	Select HSTX to be reset.
RESET_I2C0	Select I2C0 to be reset.
RESET_I2C1	Select I2C1 to be reset.
RESET_IO_BANK0	Select IO_BANK0 to be reset.
RESET_IO_QSPI	Select IO_QSPI to be reset.
RESET_JTAG	Select JTAG to be reset.
RESET_PADS_BANK0	Select PADS_BANK0 to be reset.
RESET_PADS_QSPI	Select PADS_QSPI to be reset.
RESET_PIO0	Select PIO0 to be reset.
RESET_PIO1	Select PIO1 to be reset.
RESET_PIO2	Select PIO2 to be reset.
RESET_PLL_SYS	Select PLL_SYS to be reset.
RESET_PLL_USB	Select PLL_USB to be reset.
RESET_PWM	Select PWM to be reset.
RESET_SHA256	Select SHA256 to be reset.
RESET_SPI0	Select SPI0 to be reset.
RESET_SPI1	Select SPI1 to be reset.
RESET_SYSCFG	Select SYSCFG to be reset.
RESET_SYSINFO	Select SYSINFO to be reset.
RESET_TBMAN	Select TBMAN to be reset.
RESET_TIMER0	Select TIMER0 to be reset.
RESET_TIMER1	Select TIMER1 to be reset.

RESET_TRNG	Select TRNG to be reset.
RESET_UART0	Select UART0 to be reset.
RESET_UART1	Select UART1 to be reset.
RESET_USBCTRL	Select USBCTRL to be reset.

4.1.19.7. Function Documentation

4.1.19.7.1. reset_block_mask

`static __force_inline void reset_block_mask (uint32_t bits) [static]`

Reset the specified HW blocks.

Parameters

`bits` Bit pattern indicating blocks to reset. See reset_bitmask

4.1.19.7.2. reset_block_num

`static void reset_block_num (uint32_t block_num) [inline], [static]`

Reset the specified HW block.

Parameters

`block_num` the block number

4.1.19.7.3. reset_unreset_block_num_wait_blocking

`static void reset_unreset_block_num_wait_blocking (uint block_num) [inline], [static]`

Reset the specified HW block, and then bring at back out of reset and wait for completion.

Parameters

`block_num` the block number

4.1.19.7.4. unreset_block_mask

`static __force_inline void unreset_block_mask (uint32_t bits) [static]`

bring specified HW blocks out of reset

Parameters

`bits` Bit pattern indicating blocks to unreset. See reset_bitmask

4.1.19.7.5. unreset_block_mask_wait_blocking

`static __force_inline void unreset_block_mask_wait_blocking (uint32_t bits) [static]`

Bring specified HW blocks out of reset and wait for completion.

Parameters

`bits` Bit pattern indicating blocks to unreset. See reset_bitmask

4.1.19.7.6. unreset_block_num

`static void unreset_block_num (uint block_num) [inline], [static]`

bring specified HW block out of reset

Parameters

`block_num`	the block number

4.1.19.7.7. unreset_block_num_wait_blocking

`static void unreset_block_num_wait_blocking (uint block_num) [inline], [static]`

Bring specified HW block out of reset and wait for completion.

Parameters

`block_num`	the block number

4.1.20. hardware_riscv RP2350

Accessors for standard RISC-V hardware (mainly CSRs)

4.1.21. hardware_riscv_platform_timer RP2350

Accessors for standard RISC-V platform timer (mtime/mtimecmp), available on Raspberry Pi microcontrollers with RISC-V processors.

4.1.21.1. Detailed Description

Note this header can be used by Arm as well as RISC-V processors, as the timer is a memory-mapped peripheral external to the processors. The name refers to this timer being a standard RISC-V peripheral.

4.1.21.2. Functions

`static void riscv_timer_set_enabled (bool enabled)`

Enable or disable the RISC-V platform timer.

`static void riscv_timer_set_fullspeed (bool fullspeed)`

Configure the RISC-V platform timer to run at full system clock speed.

`static uint64_t riscv_timer_get_mtime (void)`

Read the RISC-V platform timer.

`static void riscv_timer_set_mtime (uint64_t mtime)`

Update the RISC-V platform timer.

`static uint64_t riscv_timer_get_mtimecmp (void)`

Get the current RISC-V platform timer mtimecmp value for this core.

`static void riscv_timer_set_mtimecmp (uint64_t mtimecmp)`

Set a new RISC-V platform timer interrupt comparison value (mtimecmp) for this core.

4.1.21.3. Function Documentation

4.1.21.3.1. riscv_timer_get_mtime

`static uint64_t riscv_timer_get_mtime (void) [inline], [static]`

Read the RISC-V platform timer.

Returns

Current 64-bit mtime value

4.1.21.3.2. riscv_timer_get_mtimecmp

`static uint64_t riscv_timer_get_mtimecmp (void) [inline], [static]`

Get the current RISC-V platform timer mtimecmp value for this core.

Get the current mtimecmp value for the calling core. This function is interrupt-safe as long as timer interrupts only increase the value of mtimecmp. Otherwise, it must be called with timer interrupts disabled.

Returns

Current value of mtimecmp

4.1.21.3.3. riscv_timer_set_enabled

`static void riscv_timer_set_enabled (bool enabled) [inline], [static]`

Enable or disable the RISC-V platform timer.

This enables and disables the counting of the RISC-V platform timer. It does not enable or disable the interrupts, which are asserted unconditionally when a given core's mtimecmp/mtimecmph registers are greater than the current 64-bit value of the mtime/mtimeh registers.

Parameters

`enabled`	Pass true to enable, false to disable

4.1.21.3.4. riscv_timer_set_fullspeed

`static void riscv_timer_set_fullspeed (bool fullspeed) [inline], [static]`

Configure the RISC-V platform timer to run at full system clock speed.

Parameters

`fullspeed`	Pass true to increment at system clock speed, false to increment at the frequency defined by the system tick generator (the `ticks` block)

4.1.21.3.5. riscv_timer_set_mtime

`static void riscv_timer_set_mtime (uint64_t mtime) [inline], [static]`

Update the RISC-V platform timer.

This function should only be called when the timer is disabled via riscv_timer_set_enabled(). Note also that unlike the mtimecmp comparison values, mtime is *not* core-local, so updates on one core will be visible to the other core.

Parameters

`mtime`	New value to set the RISC-V platform timer to

4.1.21.3.6. riscv_timer_set_mtimecmp

`static void riscv_timer_set_mtimecmp (uint64_t mtimecmp) [inline], [static]`

Set a new RISC-V platform timer interrupt comparison value (mtimecmp) for this core.

This function updates the mtimecmp value for the current core. The calling core's RISC-V platform timer interrupt is asserted whenever the 64-bit mtime value (stored in 32-bit mtime/mtimeh registers) is greater than or equal to this core's current mtime/mtimecmph value.

Parameters

`mtime`	New value to set the RISC-V platform timer to

4.1.22. hardware_rtc RP2040

Hardware Real Time Clock API.

4.1.22.1. Detailed Description

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);

    // clk_sys is >2000x faster than clk_rtc, so datetime is not updated immediately when
rtc_get_datetime() is called.
```

```
    // The delay is up to 3 RTC clock cycles (which is 64us with the default clock settings)
    sleep_us(64);

    // 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);
    }
}
```

4.1.22.2. Typedefs

`typedef void(* rtc_callback_t)(void)`

4.1.22.3. Functions

`void rtc_init (void)`

Initialise the RTC system.

`bool rtc_set_datetime (const datetime_t *t)`

Set the RTC to the specified time.

`bool rtc_get_datetime (datetime_t *t)`

Get the current time from the RTC.

`bool rtc_running (void)`

Is the RTC running?

`void rtc_set_alarm (const datetime_t *t, rtc_callback_t user_callback)`

Set a time in the future for the RTC to call a user provided callback.

`void rtc_enable_alarm (void)`

Enable the RTC alarm (if inactive)

`void rtc_disable_alarm (void)`

Disable the RTC alarm (if active)

4.1.22.4. Typedef Documentation

4.1.22.4.1. rtc_callback_t

`typedef void(* rtc_callback_t) (void)`

Callback function type for RTC alarms

See also

rtc_set_alarm()

4.1.22.5. Function Documentation

4.1.22.5.1. rtc_disable_alarm

`void rtc_disable_alarm (void)`

Disable the RTC alarm (if active)

4.1.22.5.2. rtc_enable_alarm

`void rtc_enable_alarm (void)`

Enable the RTC alarm (if inactive)

4.1.22.5.3. 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.

4.1.22.5.4. rtc_init

`void rtc_init (void)`

Initialise the RTC system.

4.1.22.5.5. rtc_running

`bool rtc_running (void)`

Is the RTC running?

4.1.22.5.6. rtc_set_alarm

`void rtc_set_alarm (const datetime_t * t, rtc_callback_t user_callback)`

Set a time in the future for the RTC to call a user provided callback.

Parameters

`t`	Pointer to a datetime_t structure containing a time in the future to fire the alarm. Any values set to -1 will not be matched on.
`user_callback`	pointer to a rtc_callback_t to call when the alarm fires

4.1.22.5.7. rtc_set_datetime

`bool rtc_set_datetime (const datetime_t * t)`

Set the RTC to the specified time.

NOTE

Note that after setting the RTC date and time, a subsequent read of the values (e.g. via rtc_get_datetime()) may not reflect the new setting until up to three cycles of the potentially-much-slower RTC clock domain have passed. This represents a period of 64 microseconds with the default RTC clock configuration.

Parameters

`t` Pointer to a datetime_t structure contains time to set

Returns

true if set, false if the passed in datetime was invalid.

4.1.23. hardware_rcp RP2350

Inline functions and assembly macros for the Redundancy Coprocessor.

4.1.24. hardware_spi

Hardware SPI API.

4.1.24.1. Detailed Description

RP-series microcontrollers have 2 identical instances of the 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.

4.1.24.2. Macros

- `#define spi0 ((spi_inst_t *)spi0_hw)`
- `#define spi1 ((spi_inst_t *)spi1_hw)`
- `#define SPI_NUM(spi)`
- `#define SPI_INSTANCE(num)`
- `#define SPI_DREQ_NUM(spi, is_tx)`

4.1.24.3. Enumerations

`enum spi_cpha_t { SPI_CPHA_0 = 0, SPI_CPHA_1 = 1 }`

Enumeration of SPI CPHA (clock phase) values.

`enum spi_cpol_t { SPI_CPOL_0 = 0, SPI_CPOL_1 = 1 }`

Enumeration of SPI CPOL (clock polarity) values.

`enum spi_order_t { SPI_LSB_FIRST = 0, SPI_MSB_FIRST = 1 }`

Enumeration of SPI bit-order values.

4.1.24.4. Functions

`uint spi_init (spi_inst_t *spi, uint baudrate)`

Initialise SPI instances.

`void spi_deinit (spi_inst_t *spi)`

Deinitialise SPI instances.

`uint spi_set_baudrate (spi_inst_t *spi, uint baudrate)`

Set SPI baudrate.

`uint spi_get_baudrate (const spi_inst_t *spi)`

Get SPI baudrate.

`static uint spi_get_index (const spi_inst_t *spi)`

Convert SPI instance to hardware instance number.

`static void spi_set_format (spi_inst_t *spi, uint data_bits, spi_cpol_t cpol, spi_cpha_t cpha, __unused spi_order_t order)`

Configure SPI.

`static void spi_set_slave (spi_inst_t *spi, bool slave)`

Set SPI master/slave.

`static bool spi_is_writable (const spi_inst_t *spi)`

Check whether a write can be done on SPI device.

`static bool spi_is_readable (const spi_inst_t *spi)`

Check whether a read can be done on SPI device.

`static bool spi_is_busy (const spi_inst_t *spi)`

Check whether SPI is busy.

`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.

`int spi_write_blocking (spi_inst_t *spi, const uint8_t *src, size_t len)`

Write to an SPI device, 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.

`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.

`int spi_write16_blocking (spi_inst_t *spi, const uint16_t *src, size_t len)`

Write to an SPI device.

`int spi_read16_blocking (spi_inst_t *spi, uint16_t repeated_tx_data, uint16_t *dst, size_t len)`

Read from an SPI device.

`static uint spi_get_dreq (spi_inst_t *spi, bool is_tx)`

Return the DREQ to use for pacing transfers to/from a particular SPI instance.

4.1.24.5. Macro Definition Documentation

4.1.24.5.1. spi0

```
#define spi0 ((spi_inst_t *)spi0_hw)
```

Identifier for the first (SPI 0) hardware SPI instance (for use in SPI functions).

e.g. spi_init(spi0, 48000)

4.1.24.5.2. spi1

```
#define spi1 ((spi_inst_t *)spi1_hw)
```

Identifier for the second (SPI 1) hardware SPI instance (for use in SPI functions).

e.g. spi_init(spi1, 48000)

4.1.24.5.3. SPI_NUM

```
#define SPI_NUM(spi)
```

Returns the SPI number for a SPI instance.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.24.5.4. SPI_INSTANCE

```
#define SPI_INSTANCE(num)
```

Returns the SPI instance with the given SPI number.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.24.5.5. SPI_DREQ_NUM

```
#define SPI_DREQ_NUM(spi, is_tx)
```

Returns the dreq_num_t used for pacing DMA transfers to or from this SPI instance. If is_tx is true, then it is for transfers to the SPI else for transfers from the SPI.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.24.6. Enumeration Type Documentation

4.1.24.6.1. spi_cpha_t

```
enum spi_cpha_t
```

Enumeration of SPI CPHA (clock phase) values.

4.1.24.6.2. spi_cpol_t

```
enum spi_cpol_t
```

Enumeration of SPI CPOL (clock polarity) values.

4.1.24.6.3. spi_order_t

`enum spi_order_t`

Enumeration of SPI bit-order values.

4.1.24.7. Function Documentation

4.1.24.7.1. 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 re-enable the device functions.

Parameters

`spi`	SPI instance specifier, either spi0 or spi1

4.1.24.7.2. spi_get_baudrate

`uint spi_get_baudrate (const spi_inst_t * spi)`

Get SPI baudrate.

Get SPI baudrate which was set by

See also

spi_set_baudrate

Parameters

`spi`	SPI instance specifier, either spi0 or spi1

Returns

The actual baudrate set

4.1.24.7.3. spi_get_dreq

`static uint spi_get_dreq (spi_inst_t * spi, bool is_tx) [inline], [static]`

Return the DREQ to use for pacing transfers to/from a particular SPI instance.

Parameters

`spi`	SPI instance specifier, either spi0 or spi1
`is_tx`	true for sending data to the SPI instance, false for receiving data from the SPI instance

4.1.24.7.4. spi_get_index

`static uint spi_get_index (const spi_inst_t * spi) [inline], [static]`

Convert SPI instance to hardware instance number.

Parameters

`spi`	SPI instance

Returns

Number of SPI, 0 or 1.

4.1.24.7.5. spi_init

`uint 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.

NOTE

There is no guarantee that the baudrate requested can be achieved exactly; the nearest will be chosen and returned

Parameters

`spi`	SPI instance specifier, either spi0 or spi1
`baudrate`	Baudrate requested in Hz

Returns

the actual baud rate set

4.1.24.7.6. spi_is_busy

`static bool spi_is_busy (const spi_inst_t * spi) [inline], [static]`

Check whether SPI is busy.

Parameters

`spi`	SPI instance specifier, either spi0 or spi1

Returns

true if SPI is busy

4.1.24.7.7. spi_is_readable

`static bool spi_is_readable (const spi_inst_t * spi) [inline], [static]`

Check whether a read can be done on SPI device.

Parameters

`spi`	SPI instance specifier, either spi0 or spi1

Returns

true if a read is possible i.e. data is present

4.1.24.7.8. spi_is_writable

`static bool spi_is_writable (const spi_inst_t * spi) [inline], [static]`

Check whether a write can be done on SPI device.

Parameters

`spi`	SPI instance specifier, either spi0 or spi1

Returns

false if no space is available to write. True if a write is possible

4.1.24.7.9. 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 TX as data is read in from RX. Generally this can be 0, but some devices require a specific value here, e.g. SD cards expect 0xff

> **NOTE**
>
> SPI should be initialised with 16 data_bits using spi_set_format first, otherwise this function will only read 8 data_bits.

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 halfwords written/read

4.1.24.7.10. 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 TX as data is read in from RX. 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

4.1.24.7.11. 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

4.1.24.7.12. spi_set_format

`static void spi_set_format (spi_inst_t * spi, uint data_bits, spi_cpol_t cpol, spi_cpha_t cpha, __unused spi_order_t order) [inline], [static]`

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

4.1.24.7.13. spi_set_slave

`static void spi_set_slave (spi_inst_t * spi, bool slave) [inline], [static]`

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.

4.1.24.7.14. 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.

> **NOTE**
>
> SPI should be initialised with 16 data_bits using spi_set_format first, otherwise this function will only write 8 data_bits.

Parameters

`spi`	SPI instance specifier, either spi0 or spi1
`src`	Buffer of data to write
`len`	Length of buffers

Returns

Number of halfwords written/read

4.1.24.7.15. 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.

> **NOTE**
>
> SPI should be initialised with 16 data_bits using spi_set_format first, otherwise this function will only read/write 8 data_bits.

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 halfwords written/read

4.1.24.7.16. 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

4.1.24.7.17. 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

4.1.25. hardware_sha256 RP2350

Hardware SHA-256 Accelerator API.

4.1.25.1. Detailed Description

RP2350 is equipped with an implementation of the SHA-256 hash algorithm. The hardware should first be configured by calling the sha256_set_dma_size and sha256_set_bswap functions. To generate a new hash the hardware should first be initialised by calling sha256_start. The hardware is ready to accept data when sha256_is_ready returns true, at which point the data to be hashed can be written to the address returned by sha256_get_write_addr. The hardware requires 64 bytes to be written in one go or else sha256_err_not_ready will indicate an error and the hashing process must be restarted. sha256_is_sum_valid will return true when there is a valid checksum result which can be retrieved by calling sha256_get_result.

4.1.25.2. Macros

- `#define SHA256_RESULT_BYTES 32`

4.1.25.3. Enumerations

`enum sha256_endianness { SHA256_LITTLE_ENDIAN, SHA256_BIG_ENDIAN }`

SHA-256 endianness definition used in the API.

4.1.25.4. Functions

`static void sha256_set_dma_size (uint size_in_bytes)`

Configure the correct DMA data size.

`static void sha256_set_bswap (bool swap)`

Enable or disable byte swapping of 32-bit values.

`static void sha256_start (void)`

Prepare the hardware for a new checksum.

`static bool sha256_is_sum_valid (void)`

Check if a valid checksum has been calculated.

`static bool sha256_is_ready (void)`

Check if a the hardware is ready to accept more data.

`static void sha256_wait_valid_blocking (void)`

Wait until the checksum is valid.

`static void sha256_wait_ready_blocking (void)`

Wait until the hardware is ready to accept more data.

`void sha256_get_result (sha256_result_t *out, enum sha256_endianness endianness)`

Get the checksum result.

`static bool sha256_err_not_ready (void)`

Check if data was written before the hardware was ready.

`static void sha256_err_not_ready_clear (void)`

Clear the "not ready" error condition.

`static volatile void * sha256_get_write_addr (void)`

Address to write the data to be hashed.

`static void sha256_put_word (uint32_t word)`

Write one 32bit word of data to the SHA-256 hardware.

`static void sha256_put_byte (uint8_t b)`

Write one byte of data to the SHA-256 hardware.

4.1.25.5. Macro Definition Documentation

4.1.25.5.1. SHA256_RESULT_BYTES

```
#define SHA256_RESULT_BYTES 32
```

Size of a sha256 result in bytes.

4.1.25.6. Enumeration Type Documentation

4.1.25.6.1. sha256_endianness

```
enum sha256_endianness
```

SHA-256 endianness definition used in the API.

Table 30. Enumerator

SHA256_LITTLE_ENDIAN	Little Endian.
SHA256_BIG_ENDIAN	Big Endian.

4.1.25.7. Function Documentation

4.1.25.7.1. sha256_err_not_ready

```
static bool sha256_err_not_ready (void) [inline], [static]
```

Check if data was written before the hardware was ready.

Indicates if an error has occurred due to data being written when the hardware is not ready.

Returns

True if data was written before the hardware was ready

4.1.25.7.2. sha256_err_not_ready_clear

`static void sha256_err_not_ready_clear (void) [inline], [static]`

Clear the "not ready" error condition.

Resets the hardware if a "not ready" error condition is indicated.

4.1.25.7.3. sha256_get_result

`void sha256_get_result (sha256_result_t * out, enum sha256_endianness endianness)`

Get the checksum result.

Read the 32 byte result calculated by the hardware. Only valid if sha256_is_sum_valid is True

Parameters

`out`	The checksum result

Copyright (c) 2024 Raspberry Pi (Trading) Ltd.

SPDX-License-Identifier: BSD-3-Clause

4.1.25.7.4. sha256_get_write_addr

`static volatile void * sha256_get_write_addr (void) [inline], [static]`

Address to write the data to be hashed.

Returns the hardware address where data to be hashed should be written

Returns

Address to write data to be hashed

4.1.25.7.5. sha256_is_ready

`static bool sha256_is_ready (void) [inline], [static]`

Check if a the hardware is ready to accept more data.

After writing 64 bytes of data to the hardware, it will be unable to accept more data for a time. Call this to check if the hardware is ready for more data to be written.

See also

sha256_err_not_ready

Returns

True if the hardware is ready to receive more data

4.1.25.7.6. sha256_is_sum_valid

`static bool sha256_is_sum_valid (void) [inline], [static]`

Check if a valid checksum has been calculated.

The checksum result will be invalid when data is first written to the hardware, and then once 64 bytes of data has been written it may take some time to complete the digest of the current block. This function can be used to determine when the checksum is valid.

Returns

True if sha256_get_result would return a valid result

4.1.25.7.7. sha256_put_byte

`static void sha256_put_byte (uint8_t b) [inline], [static]`

Write one byte of data to the SHA-256 hardware.

Parameters

`b`	data to write

4.1.25.7.8. sha256_put_word

`static void sha256_put_word (uint32_t word) [inline], [static]`

Write one 32bit word of data to the SHA-256 hardware.

Parameters

`word`	data to write

4.1.25.7.9. sha256_set_bswap

`static void sha256_set_bswap (bool swap) [inline], [static]`

Enable or disable byte swapping of 32-bit values.

The SHA256 algorithm expects bytes in big endian order, but the system bus deals with little endian data, so control is provided to convert little endian bus data to big endian internal data. This defaults to true

Parameters

`swap`	false to disable byte swapping

4.1.25.7.10. sha256_set_dma_size

`static void sha256_set_dma_size (uint size_in_bytes) [inline], [static]`

Configure the correct DMA data size.

This must be configured before the DMA channel is triggered and ensures the correct number of transfers is requested per block.

Parameters

`size_in_bytes`	Size of DMA transfers, either 1, 2 or 4 bytes only.

4.1.25.7.11. sha256_start

`static void sha256_start (void) [inline], [static]`

Prepare the hardware for a new checksum.

Called to initialise the hardware before starting the checksum calculation

4.1.25.7.12. sha256_wait_ready_blocking

`static void sha256_wait_ready_blocking (void) [inline], [static]` Wait until the hardware is ready to accept more data.

Before writing to the hardware, it's necessary to check it is ready to accept more data. This function waits until the hardware is ready to accept more data

4.1.25.7.13. sha256_wait_valid_blocking

```
static void sha256_wait_valid_blocking (void) [inline], [static]
```

Wait until the checksum is valid.

When a multiple of 64 bytes of data has been written to the hardware, the checksum will be valid once the digest of the current block is complete. This function waits until when the checksum result is valid.

4.1.26. hardware_sync

Low level hardware spin locks, barrier and processor event APIs.

4.1.26.1. Detailed Description

Spin Locks

The RP-series microcontrollers provide 32 hardware spin locks, which can be used to manage mutually-exclusive access to shared software and hardware resources.

Generally each spin lock itself is a shared resource, i.e. the same hardware spin lock can be used by multiple higher level primitives (as long as the spin locks are neither held for long periods, nor held concurrently with other spin locks by the same core - which could lead to deadlock). A hardware spin lock that is exclusively owned can be used individually without more flexibility and without regard to other software. Note that no hardware spin lock may be acquired re-entrantly (i.e. hardware spin locks are not on their own safe for use by both thread code and IRQs) however the default spinlock related methods here (e.g. spin_lock_blocking) always disable interrupts while the lock is held as use by IRQ handlers and user code is common/desirable, and spin locks are only expected to be held for brief periods.

RP2350 Warning. Due to erratum RP2350-E2, writes to new SIO registers above an offset of +0x180 alias the spinlocks, causing spurious lock releases. This SDK by default uses atomic memory accesses to implement the hardware_sync_spin_lock API, as a workaround on RP2350 A2.

The SDK uses the following default spin lock assignments, classifying which spin locks are reserved for exclusive/special purposes vs those suitable for more general shared use:

Number (ID)	Description
0-13	Currently reserved for exclusive use by the SDK and other libraries. If you use these spin locks, you risk breaking SDK or other library functionality. Each reserved spin lock used individually has its own PICO_SPINLOCK_ID so you can search for those.
14,15	(PICO_SPINLOCK_ID_OS1 and PICO_SPINLOCK_ID_OS2). Currently reserved for exclusive use by an operating system (or other system level software) co-existing with the SDK.

Number (ID)	Description
16-23	(PICO_SPINLOCK_ID_STRIPED_FIRST - PICO_SPINLOCK_ID_STRIPED_LAST). Spin locks from this range are assigned in a round-robin fashion via next_striped_spin_lock_num(). These spin locks are shared, but assigning numbers from a range reduces the probability that two higher level locking primitives using *striped* spin locks will actually be using the same spin lock.
24-31	(PICO_SPINLOCK_ID_CLAIM_FREE_FIRST - PICO_SPINLOCK_ID_CLAIM_FREE_LAST). These are reserved for exclusive use and are allocated on a first come first served basis at runtime via spin_lock_claim_unused()

4.1.26.2. Macros

- `#define SW_SPIN_LOCK_TYPE volatile uint8_t`

4.1.26.3. Functions

`static __force_inline void __nop (void)`

Insert a NOP instruction in to the code path.

`static __force_inline void __sev (void)`

Insert a SEV instruction in to the code path.

`static __force_inline void __wfe (void)`

Insert a WFE instruction in to the code path.

`static __force_inline void __wfi (void)`

Insert a WFI instruction in to the code path.

`static __force_inline void __dmb (void)`

Insert a DMB instruction in to the code path.

`static __force_inline void __dsb (void)`

Insert a DSB instruction in to the code path.

`static __force_inline void __isb (void)`

Insert a ISB instruction in to the code path.

`static __force_inline void __mem_fence_acquire (void)`

Acquire a memory fence.

`static __force_inline void __mem_fence_release (void)`

Release a memory fence.

`static __force_inline uint32_t disable_interrupts (void)`

Explicitly disable interrupts on the calling core.

`static __force_inline uint32_t enable_interrupts (void)`

Explicitly enable interrupts on the calling core.

`static __force_inline uint32_t save_and_disable_interrupts (void)`

Disable interrupts on the calling core, returning the previous interrupt state.

`static __force_inline void restore_interrupts (uint32_t status)`

Restore interrupts to a specified state on the calling core.

`static __force_inline void restore_interrupts_from_disabled (uint32_t status)`

Restore interrupts to a specified state on the calling core with restricted transitions.

`uint next_striped_spin_lock_num (void)`

Return a spin lock number from the *striped* range.

`void spin_lock_claim (uint lock_num)`

Mark a spin lock as used.

`void spin_lock_claim_mask (uint32_t lock_num_mask)`

Mark multiple spin locks as used.

`void spin_lock_unclaim (uint lock_num)`

Mark a spin lock as no longer used.

`int spin_lock_claim_unused (bool required)`

Claim a free spin lock.

`bool spin_lock_is_claimed (uint lock_num)`

Determine if a spin lock is claimed.

`static __force_inline spin_lock_t * spin_lock_instance (uint lock_num)`

Get HW Spinlock instance from number.

`static __force_inline uint spin_lock_get_num (spin_lock_t *lock)`

Get HW Spinlock number from instance.

`static __force_inline void spin_lock_unsafe_blocking (spin_lock_t *lock)`

Acquire a spin lock without disabling interrupts (hence unsafe)

`static __force_inline void spin_unlock_unsafe (spin_lock_t *lock)`

Release a spin lock without re-enabling interrupts.

`static __force_inline uint32_t spin_lock_blocking (spin_lock_t *lock)`

Acquire a spin lock safely.

`static bool is_spin_locked (spin_lock_t *lock)`

Check to see if a spinlock is currently acquired elsewhere.

`static __force_inline void spin_unlock (spin_lock_t *lock, uint32_t saved_irq)`

Release a spin lock safely.

`spin_lock_t * spin_lock_init (uint lock_num)`

Initialise a spin lock.

`void spin_locks_reset (void)`

Release all spin locks.

4.1.26.4. Macro Definition Documentation

4.1.26.4.1. SW_SPIN_LOCK_TYPE RP2350

```
#define SW_SPIN_LOCK_TYPE volatile uint8_t
```

A spin lock identifier.

4.1.26.5. Function Documentation

4.1.26.5.1. __dmb

`static __force_inline void __dmb (void) [static]`

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.

4.1.26.5.2. __dsb

`static __force_inline void __dsb (void) [static]`

Insert a DSB instruction in to the code path.

The DSB (data synchronization barrier) acts as a special kind of data memory barrier (DMB). The DSB operation completes when all explicit memory accesses before this instruction complete.

4.1.26.5.3. __isb

`static __force_inline void __isb (void) [static]`

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.

4.1.26.5.4. __mem_fence_acquire

`static __force_inline void __mem_fence_acquire (void) [static]`

Acquire a memory fence.

4.1.26.5.5. __mem_fence_release

`static __force_inline void __mem_fence_release (void) [static]`

Release a memory fence.

4.1.26.5.6. __nop

`static __force_inline void __nop (void) [static]`

Insert a NOP instruction in to the code path.

NOP does nothing for one cycle. On RP2350 Arm binaries this is forced to be a 32-bit instruction to avoid dual-issue of NOPs.

4.1.26.5.7. __sev

`static __force_inline void __sev (void) [static]`

Insert a SEV instruction in to the code path.

The SEV (send event) instruction sends an event to both cores.

4.1.26.5.8. __wfe

`static __force_inline void __wfe (void) [static]`

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 signalled by the SEV instruction on either core.

4.1.26.5.9. __wfi

`static __force_inline void __wfi (void) [static]`

Insert a WFI instruction in to the code path.

The WFI (wait for interrupt) instruction waits for a interrupt to wake up the core.

4.1.26.5.10. disable_interrupts

`static __force_inline uint32_t disable_interrupts (void) [static]`

Explicitly disable interrupts on the calling core.

4.1.26.5.11. enable_interrupts

`static __force_inline uint32_t enable_interrupts (void) [static]`

Explicitly enable interrupts on the calling core.

4.1.26.5.12. is_spin_locked

`static bool is_spin_locked (spin_lock_t * lock) [inline], [static]`

Check to see if a spinlock is currently acquired elsewhere.

Parameters

`lock`	Spinlock instance

4.1.26.5.13. next_striped_spin_lock_num

`uint next_striped_spin_lock_num (void)`

Return a spin lock number from the *striped* range.

Returns a spin lock number in the range PICO_SPINLOCK_ID_STRIPED_FIRST to PICO_SPINLOCK_ID_STRIPED_LAST in a round robin fashion. This does not grant the caller exclusive access to the spin lock, so the caller must:

1. Abide (with other callers) by the contract of only holding this spin lock briefly (and with IRQs disabled - the default via spin_lock_blocking()), and not whilst holding other spin locks.
2. Be OK with any contention caused by the - brief due to the above requirement - contention with other possible users of the spin lock.

Returns

lock_num a spin lock number the caller may use (non exclusively)

See also

PICO_SPINLOCK_ID_STRIPED_FIRST

PICO_SPINLOCK_ID_STRIPED_LAST

4.1.26.5.14. restore_interrupts

`static __force_inline void restore_interrupts (uint32_t status) [static]`

Restore interrupts to a specified state on the calling core.

Parameters

`status`	Previous interrupt status from save_and_disable_interrupts()

4.1.26.5.15. restore_interrupts_from_disabled

`static __force_inline void restore_interrupts_from_disabled (uint32_t status) [static]`

Restore interrupts to a specified state on the calling core with restricted transitions.

This method should only be used when the current interrupt state is known to be disabled, e.g. when paired with save_and_disable_interrupts()

Parameters

`status`	Previous interrupt status from save_and_disable_interrupts()

4.1.26.5.16. save_and_disable_interrupts

`static __force_inline uint32_t save_and_disable_interrupts (void) [static]`

Disable interrupts on the calling core, returning the previous interrupt state.

This method is commonly paired with restore_interrupts_from_disabled() to temporarily disable interrupts around a piece of code, without needing to care whether interrupts were previously enabled

Returns

The prior interrupt enable status for restoration later via restore_interrupts_from_disabled() or restore_interrupts()

4.1.26.5.17. spin_lock_blocking

`static __force_inline uint32_t spin_lock_blocking (spin_lock_t * lock) [static]`

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

4.1.26.5.18. 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

4.1.26.5.19. 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)

4.1.26.5.20. 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 available

Returns

the spin lock number or -1 if required was false, and none were free

4.1.26.5.21. spin_lock_get_num

`static __force_inline uint spin_lock_get_num (spin_lock_t * lock) [static]`

Get HW Spinlock number from instance.

Parameters

`lock`	The Spinlock instance

Returns

The Spinlock ID

4.1.26.5.22. 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

4.1.26.5.23. spin_lock_instance

`static __force_inline spin_lock_t * spin_lock_instance (uint lock_num) [static]`

Get HW Spinlock instance from number.

Parameters

`lock_num` Spinlock ID

Returns

The spinlock instance

4.1.26.5.24. spin_lock_is_claimed

`bool spin_lock_is_claimed (uint lock_num)`

Determine if a spin lock is claimed.

Parameters

`lock_num` the spin lock number

Returns

true if claimed, false otherwise

See also

spin_lock_claim

spin_lock_claim_mask

4.1.26.5.25. 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

4.1.26.5.26. spin_lock_unsafe_blocking

`static __force_inline void spin_lock_unsafe_blocking (spin_lock_t * lock) [static]`

Acquire a spin lock without disabling interrupts (hence unsafe)

Parameters

`lock` Spinlock instance

4.1.26.5.27. spin_locks_reset

`void spin_locks_reset (void)`

Release all spin locks.

4.1.26.5.28. spin_unlock

`static __force_inline void spin_unlock (spin_lock_t * lock, uint32_t saved_irq) [static]`

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.

See also

spin_lock_blocking()

4.1.26.5.29. spin_unlock_unsafe

`static __force_inline void spin_unlock_unsafe (spin_lock_t * lock) [static]`

Release a spin lock without re-enabling interrupts.

Parameters

`lock`	Spinlock instance

4.1.27. hardware_ticks

Hardware Tick API.

4.1.27.1. Detailed Description

RP2040 only has one tick generator, and it is part of the watchdog hardware.

The RP2350 has a dedicated Tick block that is used to supply ticks to TIMER0, TIMER1, RISC-V platform timer, Arm Cortex-M33 0 timer, Arm Cortex-M33 1 timer and the WATCHDOG block.

4.1.27.2. Typedefs

`typedef enum tick_gen_num_rp2350 tick_gen_num_t` RP2350

Tick generator numbers on RP2350 (used as typedef tick_gen_num_t)

`typedef enum tick_gen_num_rp2040 tick_gen_num_t` RP2040

Tick generator numbers on RP2040 (used as typedef tick_gen_num_t)

4.1.27.3. Enumerations

`enum tick_gen_num_rp2350 { TICK_PROC0 = 0, TICK_PROC1 = 1, TICK_TIMER0 = 2, TICK_TIMER1 = 3, TICK_WATCHDOG = 4, TICK_RISCV = 5, TICK_COUNT }` RP2350

Tick generator numbers on RP2350 (used as typedef tick_gen_num_t)

`enum tick_gen_num_rp2040 { TICK_WATCHDOG = 0, TICK_COUNT }` RP2040

Tick generator numbers on RP2040 (used as typedef tick_gen_num_t)

4.1.27.4. Functions

`void tick_start (tick_gen_num_t tick, uint cycles)`

Start a tick generator.

`void tick_stop (tick_gen_num_t tick)`

Stop a tick generator.

```
bool tick_is_running (tick_gen_num_t tick)
```

Check if a tick genererator is currently running.

4.1.27.5. Typedef Documentation

4.1.27.5.1. tick_gen_num_t RP2350

```
typedef enum tick_gen_num_rp2350 tick_gen_num_t
```

Tick generator numbers on RP2350 (used as typedef tick_gen_num_t)

4.1.27.5.2. tick_gen_num_t RP2040

```
typedef enum tick_gen_num_rp2040 tick_gen_num_t
```

Tick generator numbers on RP2040 (used as typedef tick_gen_num_t)

RP2040 only has one tick generator, and it is part of the watchdog hardware

4.1.27.6. Enumeration Type Documentation

4.1.27.6.1. tick_gen_num_rp2350 RP2350

```
enum tick_gen_num_rp2350
```

Tick generator numbers on RP2350 (used as typedef tick_gen_num_t)

4.1.27.6.2. tick_gen_num_rp2040 RP2040

```
enum tick_gen_num_rp2040
```

Tick generator numbers on RP2040 (used as typedef tick_gen_num_t)

RP2040 only has one tick generator, and it is part of the watchdog hardware

4.1.27.7. Function Documentation

4.1.27.7.1. tick_is_running

```
bool tick_is_running (tick_gen_num_t tick)
```

Check if a tick genererator is currently running.

Parameters

`tick`	The tick generator number

Returns

true if the specific ticker is running.

4.1.27.7.2. tick_start

`void tick_start (tick_gen_num_t tick, uint cycles)`

Start a tick generator.

Parameters

`tick`	The tick generator number
`cycles`	The number of clock cycles per tick

4.1.27.7.3. tick_stop

`void tick_stop (tick_gen_num_t tick)`

Stop a tick generator.

Parameters

`tick`	The tick generator number

4.1.28. hardware_timer

Low-level hardware timer API.

4.1.28.1. Detailed Description

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 RP-series microcontrollers supports the following features:

- RP2040 single 64-bit counter, incrementing once per microsecond
- RP2350 two 64-bit counters, ticks generated from the tick block
- 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.

On RP2040, by default the timer uses a one microsecond reference that is generated in the Watchdog (see RP2040 Datasheet Section 4.8.2) which is derived from the clk_ref.

On RP2350, by default the timer uses a one microsecond reference that is generated by the tick block (see RP2350 Datasheet Section 8.5)

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;

```

```c
int64_t alarm_callback(alarm_id_t id, __unused 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(__unused 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

4.1.28.2. Macros

- `#define TIMER_ALARM_IRQ_NUM(timer, alarm_num)`
- `#define TIMER_ALARM_NUM_FROM_IRQ(irq_num)`
- `#define TIMER_NUM_FROM_IRQ(irq_num)`
- `#define PICO_DEFAULT_TIMER 0`

- `#define PICO_DEFAULT_TIMER_INSTANCE()`

4.1.28.3. Typedefs

`typedef void(* hardware_alarm_callback_t)(uint alarm_num)`

4.1.28.4. Functions

`static uint32_t timer_time_us_32 (timer_hw_t *timer)`

Return a 32 bit timestamp value in microseconds for a given timer instance.

`static uint32_t time_us_32 (void)`

Return a 32 bit timestamp value in microseconds for the default timer instance.

`uint64_t timer_time_us_64 (timer_hw_t *timer)`

Return the current 64 bit timestamp value in microseconds for a given timer instance.

`uint64_t time_us_64 (void)`

Return the current 64 bit timestamp value in microseconds for the default timer instance.

`void timer_busy_wait_us_32 (timer_hw_t *timer, uint32_t delay_us)`

Busy wait wasting cycles for the given (32 bit) number of microseconds using the given timer instance.

`void busy_wait_us_32 (uint32_t delay_us)`

Busy wait wasting cycles for the given (32 bit) number of microseconds using the default timer instance.

`void timer_busy_wait_us (timer_hw_t *timer, uint64_t delay_us)`

Busy wait wasting cycles for the given (64 bit) number of microseconds using the given timer instance.

`void busy_wait_us (uint64_t delay_us)`

Busy wait wasting cycles for the given (64 bit) number of microseconds using the default timer instance.

`void timer_busy_wait_ms (timer_hw_t *timer, uint32_t delay_ms)`

Busy wait wasting cycles for the given number of milliseconds using the given timer instance.

`void busy_wait_ms (uint32_t delay_ms)`

Busy wait wasting cycles for the given number of milliseconds using the default timer instance.

`void timer_busy_wait_until (timer_hw_t *timer, absolute_time_t t)`

Busy wait wasting cycles until after the specified timestamp using the given timer instance.

`void busy_wait_until (absolute_time_t t)`

Busy wait wasting cycles until after the specified timestamp using the default timer instance.

`static bool timer_time_reached (timer_hw_t *timer, absolute_time_t t)`

Check if the specified timestamp has been reached on the given timer instance.

`static bool time_reached (absolute_time_t t)`

Check if the specified timestamp has been reached on the default timer instance.

`void timer_hardware_alarm_claim (timer_hw_t *timer, uint alarm_num)`

cooperatively claim the use of this hardware alarm_num on the given timer instance

`void hardware_alarm_claim (uint alarm_num)`

cooperatively claim the use of this hardware alarm_num on the default timer instance `int timer_hardware_alarm_claim_unused (timer_hw_t *timer, bool required)`

cooperatively claim the use of a hardware alarm_num on the given timer instance

`int hardware_alarm_claim_unused (bool required)`

cooperatively claim the use of a hardware alarm_num on the default timer instance

`void timer_hardware_alarm_unclaim (timer_hw_t *timer, uint alarm_num)`

cooperatively release the claim on use of this hardware alarm_num on the given timer instance

`void hardware_alarm_unclaim (uint alarm_num)`

cooperatively release the claim on use of this hardware alarm_num on the default timer instance

`bool timer_hardware_alarm_is_claimed (timer_hw_t *timer, uint alarm_num)`

Determine if a hardware alarm has been claimed on the given timer instance.

`bool hardware_alarm_is_claimed (uint alarm_num)`

Determine if a hardware alarm has been claimed on the default timer instance.

`void timer_hardware_alarm_set_callback (timer_hw_t *timer, uint alarm_num, hardware_alarm_callback_t callback)`

Enable/Disable a callback for a hardware alarm for a given timer instance on this core.

`void hardware_alarm_set_callback (uint alarm_num, hardware_alarm_callback_t callback)`

Enable/Disable a callback for a hardware alarm on the default timer instance on this core.

`bool timer_hardware_alarm_set_target (timer_hw_t *timer, uint alarm_num, absolute_time_t t)`

Set the current target for a specific hardware alarm on the given timer instance.

`bool hardware_alarm_set_target (uint alarm_num, absolute_time_t t)`

Set the current target for the specified hardware alarm on the default timer instance.

`void timer_hardware_alarm_cancel (timer_hw_t *timer, uint alarm_num)`

Cancel an existing target (if any) for a specific hardware_alarm on the given timer instance.

`void hardware_alarm_cancel (uint alarm_num)`

Cancel an existing target (if any) for the specified hardware_alarm on the default timer instance.

`void timer_hardware_alarm_force_irq (timer_hw_t *timer, uint alarm_num)`

Force and IRQ for a specific hardware alarm on the given timer instance.

`void hardware_alarm_force_irq (uint alarm_num)`

Force and IRQ for a specific hardware alarm on the default timer instance.

`static uint timer_hardware_alarm_get_irq_num (timer_hw_t *timer, uint alarm_num)`

Returns the irq_num_t for the alarm interrupt from the given alarm on the given timer instance.

`static uint hardware_alarm_get_irq_num (uint alarm_num)`

Returns the irq_num_t for the alarm interrupt from the given alarm on the default timer instance.

`static uint timer_get_index (timer_hw_t *timer)`

Returns the timer number for a timer instance.

`static timer_hw_t * timer_get_instance (uint timer_num)`

Returns the timer instance with the given timer number.

4.1.28.5. Macro Definition Documentation

4.1.28.5.1. TIMER_ALARM_IRQ_NUM

```
#define TIMER_ALARM_IRQ_NUM(timer, alarm_num)
```

Returns the irq_num_t for the alarm interrupt from the given alarm on the given timer instance.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.28.5.2. TIMER_ALARM_NUM_FROM_IRQ

```
#define TIMER_ALARM_NUM_FROM_IRQ(irq_num)
```

Returns the alarm number from an irq_num_t. See TIMER_INSTANCE_NUM_FROM_IRQ to get the timer instance number.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.28.5.3. TIMER_NUM_FROM_IRQ

```
#define TIMER_NUM_FROM_IRQ(irq_num)
```

Returns the alarm number from an irq_num_t. See TIMER_INSTANCE_NUM_FROM_IRQ to get the alarm number.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.28.5.4. PICO_DEFAULT_TIMER

```
#define PICO_DEFAULT_TIMER 0
```

The default timer instance number of the timer instance used for APIs that don't take an explicit timer instance On RP2040 this must be 0 as there is only one timer instance On RP2040 this may be set to 0 or 1 .

4.1.28.5.5. PICO_DEFAULT_TIMER_INSTANCE

```
#define PICO_DEFAULT_TIMER_INSTANCE()
```

Returns the default timer instance on the platform based on the setting of PICO_DEFAULT_TIMER.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.28.6. Typedef Documentation

4.1.28.6.1. hardware_alarm_callback_t

```
typedef void(* hardware_alarm_callback_t) (uint alarm_num)
```

Callback function type for hardware alarms

Parameters

`alarm_num`	the hardware alarm number

See also

hardware_alarm_set_callback()

4.1.28.7. Function Documentation

4.1.28.7.1. busy_wait_ms

`void busy_wait_ms (uint32_t delay_ms)`

Busy wait wasting cycles for the given number of milliseconds using the default timer instance.

Parameters

`delay_ms`	delay amount in milliseconds

See also

timer_busy_wait_ms

4.1.28.7.2. busy_wait_until

`void busy_wait_until (absolute_time_t t)`

Busy wait wasting cycles until after the specified timestamp using the default timer instance.

Parameters

`t`	Absolute time to wait until

See also

timer_busy_wait_until

4.1.28.7.3. busy_wait_us

`void busy_wait_us (uint64_t delay_us)`

Busy wait wasting cycles for the given (64 bit) number of microseconds using the default timer instance.

Parameters

`delay_us`	delay amount in microseconds

See also

timer_busy_wait_us

4.1.28.7.4. 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 using the default timer instance.

Parameters

`delay_us`	delay amount in microseconds

See also

timer_busy_wait_us_32

4.1.28.7.5. hardware_alarm_cancel

`void hardware_alarm_cancel (uint alarm_num)`

Cancel an existing target (if any) for the specified hardware_alarm on the default timer instance.

Parameters

`alarm_num`	the hardware alarm number

See also

timer_hardware_alarm_cancel

4.1.28.7.6. hardware_alarm_claim

`void hardware_alarm_claim (uint alarm_num)`

cooperatively claim the use of this hardware alarm_num on the default timer instance

This method hard asserts if the hardware alarm is currently claimed.

Parameters

`alarm_num`	the hardware alarm to claim

See also

timer_hardware_alarm_claim

hardware_claiming

4.1.28.7.7. hardware_alarm_claim_unused

`int hardware_alarm_claim_unused (bool required)`

cooperatively claim the use of a hardware alarm_num on the default timer instance

This method attempts to claim an unused hardware alarm

Parameters

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

Returns

alarm_num the hardware alarm claimed or -1 if required was false, and none are available

See also

timer_hardware_alarm_claim_unused

hardware_claiming

4.1.28.7.8. hardware_alarm_force_irq

`void hardware_alarm_force_irq (uint alarm_num)`

Force and IRQ for a specific hardware alarm on the default timer instance.

This method will forcibly make sure the current alarm callback (if present) for the hardware alarm is called from an IRQ context after this call. If an actual callback is due at the same time then the callback may only be called once.

Calling this method does not otherwise interfere with regular callback operations.

Parameters

`alarm_num`	the hardware alarm number

See also

timer_hardware_alarm_force_irq

4.1.28.7.9. hardware_alarm_get_irq_num

```
static uint hardware_alarm_get_irq_num (uint alarm_num) [inline], [static]
```

Returns the irq_num_t for the alarm interrupt from the given alarm on the default timer instance.

Parameters

`alarm_num`	the alarm number

4.1.28.7.10. hardware_alarm_is_claimed

```
bool hardware_alarm_is_claimed (uint alarm_num)
```

Determine if a hardware alarm has been claimed on the default timer instance.

Parameters

`alarm_num`	the hardware alarm number

Returns

true if claimed, false otherwise

See also

timer_hardware_alarm_is_claimed

hardware_alarm_claim

4.1.28.7.11. hardware_alarm_set_callback

```
void hardware_alarm_set_callback (uint alarm_num, hardware_alarm_callback_t callback)
```

Enable/Disable a callback for a hardware alarm on the default timer instance 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

> **NOTE**
>
> This will install the handler on the current core if the IRQ handler isn't already set. Therefore the user has the opportunity to call this up from the core of their choice

Parameters

`alarm_num`	the hardware alarm number
`callback`	the callback to install, or NULL to unset

See also

timer_hardware_alarm_set_callback

hardware_alarm_set_target()

4.1.28.7.12. hardware_alarm_set_target

```
bool hardware_alarm_set_target (uint alarm_num, absolute_time_t t)
```

Set the current target for the specified hardware alarm on the default timer instance.

This will replace any existing target

Parameters

`alarm_num`	the hardware alarm number
`t`	the target timestamp

Returns

true if the target was "missed"; i.e. it was in the past, or occurred before a future hardware timeout could be set

See also

timer_hardware_alarm_set_target

4.1.28.7.13. hardware_alarm_unclaim

`void hardware_alarm_unclaim (uint alarm_num)`

cooperatively release the claim on use of this hardware alarm_num on the default timer instance

Parameters

`alarm_num`	the hardware alarm to unclaim

See also

timer_hardware_alarm_unclaim

hardware_claiming

4.1.28.7.14. time_reached

`static bool time_reached (absolute_time_t t) [inline], [static]`

Check if the specified timestamp has been reached on the default timer instance.

Parameters

`t`	Absolute time to compare against current time

Returns

true if it is now after the specified timestamp

See also

timer_time_reached

4.1.28.7.15. time_us_32

`static uint32_t time_us_32 (void) [inline], [static]`

Return a 32 bit timestamp value in microseconds for the default timer instance.

Returns the low 32 bits of the hardware timer.

NOTE

> This value wraps roughly every 1 hour 11 minutes and 35 seconds.

Returns

the 32 bit timestamp

See also

timer_time_us_32

4.1.28.7.16. time_us_64

`uint64_t time_us_64 (void)`

Return the current 64 bit timestamp value in microseconds for the default timer instance.

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).

Returns

the 64 bit timestamp

See also

timer_time_us_64

4.1.28.7.17. timer_busy_wait_ms

`void timer_busy_wait_ms (timer_hw_t * timer, uint32_t delay_ms)`

Busy wait wasting cycles for the given number of milliseconds using the given timer instance.

Parameters

`timer`	the timer instance
`delay_ms`	delay amount in milliseconds

See also

busy_wait_ms

4.1.28.7.18. timer_busy_wait_until

`void timer_busy_wait_until (timer_hw_t * timer, absolute_time_t t)`

Busy wait wasting cycles until after the specified timestamp using the given timer instance.

Parameters

`timer`	the timer instance
`t`	Absolute time to wait until

See also

busy_wait_until

4.1.28.7.19. timer_busy_wait_us

`void timer_busy_wait_us (timer_hw_t * timer, uint64_t delay_us)`

Busy wait wasting cycles for the given (64 bit) number of microseconds using the given timer instance.

Parameters

`timer`	the timer instance
`delay_us`	delay amount in microseconds

See also

busy_wait_us

4.1.28.7.20. timer_busy_wait_us_32

```
void timer_busy_wait_us_32 (timer_hw_t * timer, uint32_t delay_us)
```

Busy wait wasting cycles for the given (32 bit) number of microseconds using the given timer instance.

Parameters

`timer`	the timer instance
`delay_us`	delay amount in microseconds

See also

busy_wait_us_32

Busy wait wasting cycles for the given (32 bit) number of microseconds using the given timer instance.

4.1.28.7.21. timer_get_index

```
static uint timer_get_index (timer_hw_t * timer) [inline], [static]
```

Returns the timer number for a timer instance.

Parameters

`timer`	the timer instance

Returns

the timer number

See also

TIMER_NUM

4.1.28.7.22. timer_get_instance

```
static timer_hw_t * timer_get_instance (uint timer_num) [inline], [static]
```

Returns the timer instance with the given timer number.

Parameters

`timer_num`	the timer number

Returns

the timer instance

4.1.28.7.23. timer_hardware_alarm_cancel

```
void timer_hardware_alarm_cancel (timer_hw_t * timer, uint alarm_num)
```

Cancel an existing target (if any) for a specific hardware_alarm on the given timer instance.

Parameters

`timer`	the timer instance
`alarm_num`	the hardware alarm number

See also

hardware_alarm_cancel

4.1.28.7.24. timer_hardware_alarm_claim

`void timer_hardware_alarm_claim (timer_hw_t * timer, uint alarm_num)`

cooperatively claim the use of this hardware alarm_num on the given timer instance

This method hard asserts if the hardware alarm is currently claimed.

Parameters

`timer`	the timer instance
`alarm_num`	the hardware alarm to claim

See also

hardware_alarm_claim

hardware_claiming

4.1.28.7.25. timer_hardware_alarm_claim_unused

`int timer_hardware_alarm_claim_unused (timer_hw_t * timer, bool required)`

cooperatively claim the use of a hardware alarm_num on the given timer instance

This method attempts to claim an unused hardware alarm

Parameters

`timer`	the timer instance
`required`	if true the function will panic if none are available

Returns

alarm_num the hardware alarm claimed or -1 if required was false, and none are available

See also

hardware_alarm_claim_unused

hardware_claiming

4.1.28.7.26. timer_hardware_alarm_force_irq

`void timer_hardware_alarm_force_irq (timer_hw_t * timer, uint alarm_num)`

Force and IRQ for a specific hardware alarm on the given timer instance.

This method will forcibly make sure the current alarm callback (if present) for the hardware alarm is called from an IRQ context after this call. If an actual callback is due at the same time then the callback may only be called once.

Calling this method does not otherwise interfere with regular callback operations.

Parameters

`timer`	the timer instance
`alarm_num`	the hardware alarm number

See also

hardware_alarm_force_irq

4.1.28.7.27. timer_hardware_alarm_get_irq_num

`static uint timer_hardware_alarm_get_irq_num (timer_hw_t * timer, uint alarm_num) [inline], [static]`

Returns the irq_num_t for the alarm interrupt from the given alarm on the given timer instance.

Parameters

`timer`	the timer instance
`alarm_num`	the alarm number

See also

TIMER_ALARM_IRQ_NUM

4.1.28.7.28. timer_hardware_alarm_is_claimed

`bool timer_hardware_alarm_is_claimed (timer_hw_t * timer, uint alarm_num)`

Determine if a hardware alarm has been claimed on the given timer instance.

Parameters

`timer`	the timer instance
`alarm_num`	the hardware alarm number

Returns

true if claimed, false otherwise

See also

hardware_alarm_is_claimed

hardware_alarm_claim

4.1.28.7.29. timer_hardware_alarm_set_callback

`void timer_hardware_alarm_set_callback (timer_hw_t * timer, uint alarm_num, hardware_alarm_callback_t callback)`

Enable/Disable a callback for a hardware alarm for a given timer instance 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

> **NOTE**
>
> This will install the handler on the current core if the IRQ handler isn't already set. Therefore the user has the opportunity to call this up from the core of their choice

Parameters

`timer`	the timer instance
`alarm_num`	the hardware alarm number
`callback`	the callback to install, or NULL to unset

See also

hardware_alarm_set_callback

timer_hardware_alarm_set_target()

4.1.28.7.30. timer_hardware_alarm_set_target

`bool timer_hardware_alarm_set_target (timer_hw_t * timer, uint alarm_num, absolute_time_t t)`

Set the current target for a specific hardware alarm on the given timer instance.

This will replace any existing target

Parameters

`timer`	the timer instance
`alarm_num`	the hardware alarm number
`t`	the target timestamp

Returns

true if the target was "missed"; i.e. it was in the past, or occurred before a future hardware timeout could be set

See also

hardware_alarm_set_target

4.1.28.7.31. timer_hardware_alarm_unclaim

`void timer_hardware_alarm_unclaim (timer_hw_t * timer, uint alarm_num)`

cooperatively release the claim on use of this hardware alarm_num on the given timer instance

Parameters

`timer`	the timer instance
`alarm_num`	the hardware alarm to unclaim

See also

hardware_alarm_unclaim

hardware_claiming

4.1.28.7.32. timer_time_reached

`static bool timer_time_reached (timer_hw_t * timer, absolute_time_t t) [inline], [static]`

Check if the specified timestamp has been reached on the given timer instance.

Parameters

`timer`	the timer instance
`t`	Absolute time to compare against current time

Returns

true if it is now after the specified timestamp

See also

time_reached

4.1.28.7.33. timer_time_us_32

`static uint32_t timer_time_us_32 (timer_hw_t * timer) [inline], [static]`

Return a 32 bit timestamp value in microseconds for a given timer instance.

Returns the low 32 bits of the hardware timer.

NOTE

This value wraps roughly every 1 hour 11 minutes and 35 seconds.

Parameters

`timer` the timer instance

Returns

the 32 bit timestamp

See also

time_us_32

4.1.28.7.34. timer_time_us_64

`uint64_t timer_time_us_64 (timer_hw_t * timer)`

Return the current 64 bit timestamp value in microseconds for a given timer instance.

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).

Parameters

`timer` the timer instance

Returns

the 64 bit timestamp

See also

time_us_64

Return the current 64 bit timestamp value in microseconds for a given timer instance.

4.1.29. hardware_uart

Hardware UART API.

4.1.29.1. Detailed Description

RP-series microcontrollers have 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() {

    // Set the GPIO pin mux to the UART - pin 0 is TX, 1 is RX; note use of UART_FUNCSEL_NUM for the general
    // case where the func sel used for UART depends on the pin number
```

```
    // Do this before calling uart_init to avoid losing data
    gpio_set_function(0, UART_FUNCSEL_NUM(uart0, 0));
    gpio_set_function(1, UART_FUNCSEL_NUM(uart0, 1));

    // Initialise UART 0
    uart_init(uart0, 115200);

    uart_puts(uart0, "Hello world!");
}
```

4.1.29.2. Macros

- `#define UART_NUM(uart)`
- `#define UART_INSTANCE(num)`
- `#define UART_DREQ_NUM(uart, is_tx)`
- `#define UART_CLOCK_NUM(uart)`
- `#define UART_FUNCSEL_NUM(uart, gpio)`
- `#define UART_IRQ_NUM(uart)`
- `#define UART_RESET_NUM(uart)`

4.1.29.3. Enumerations

`enum uart_parity_t { UART_PARITY_NONE, UART_PARITY_EVEN, UART_PARITY_ODD }`

UART Parity enumeration.

4.1.29.4. Functions

`static uint uart_get_index (uart_inst_t *uart)`

Convert UART instance to hardware instance number.

`static uart_inst_t * uart_get_instance (uint num)`

Get the UART instance from an instance number.

`static uart_hw_t * uart_get_hw (uart_inst_t *uart)`

Get the real hardware UART instance from a UART instance.

`uint uart_init (uart_inst_t *uart, uint baudrate)`

Initialise a UART.

`void uart_deinit (uart_inst_t *uart)`

DeInitialise a UART.

`uint uart_set_baudrate (uart_inst_t *uart, uint baudrate)`

Set UART baud rate.

`static void uart_set_hw_flow (uart_inst_t *uart, bool cts, bool rts)`

Set UART flow control CTS/RTS.

`void uart_set_format (uart_inst_t *uart, uint data_bits, uint stop_bits, uart_parity_t parity)`

Set UART data format.

`static void uart_set_irqs_enabled (uart_inst_t *uart, bool rx_has_data, bool tx_needs_data)`

Enable/Disable UART interrupt outputs.

`static bool uart_is_enabled (uart_inst_t *uart)`

Test if specific UART is enabled.

`void uart_set_fifo_enabled (uart_inst_t *uart, bool enabled)`

Enable/Disable the FIFOs on specified UART.

`static bool uart_is_writable (uart_inst_t *uart)`

Determine if space is available in the TX FIFO.

`static void uart_tx_wait_blocking (uart_inst_t *uart)`

Wait for the UART TX fifo to be drained.

`static bool uart_is_readable (uart_inst_t *uart)`

Determine whether data is waiting in the RX FIFO.

`static void uart_write_blocking (uart_inst_t *uart, const uint8_t *src, size_t len)`

Write to the UART for transmission.

`static void uart_read_blocking (uart_inst_t *uart, uint8_t *dst, size_t len)`

Read from the UART.

`static void uart_putc_raw (uart_inst_t *uart, char c)`

Write single character to UART for transmission.

`static void uart_putc (uart_inst_t *uart, char c)`

Write single character to UART for transmission, with optional CR/LF conversions.

`static void uart_puts (uart_inst_t *uart, const char *s)`

Write string to UART for transmission, doing any CR/LF conversions.

`static char uart_getc (uart_inst_t *uart)`

Read a single character from the UART.

`void uart_set_break (uart_inst_t *uart, bool en)`

Assert a break condition on the UART transmission.

`void uart_set_translate_crlf (uart_inst_t *uart, bool translate)`

Set CR/LF conversion on UART.

`static void uart_default_tx_wait_blocking (void)`

Wait for the default UART's TX FIFO to be drained.

`bool uart_is_readable_within_us (uart_inst_t *uart, uint32_t us)`

Wait for up to a certain number of microseconds for the RX FIFO to be non empty.

`static uint uart_get_dreq_num (uart_inst_t *uart, bool is_tx)`

Return the dreq_num_t to use for pacing transfers to/from a particular UART instance.

`static uint uart_get_reset_num (uart_inst_t *uart)`

Return the reset_num_t to use to reset a particular UART instance.

4.1.29.4.1. uart0

```
#define uart0 ((uart_inst_t *)uart0_hw)
```

Identifier for UART instance 0.

The UART identifiers for use in UART functions.

e.g. uart_init(uart1, 48000)

4.1.29.4.2. uart1

```
#define uart1 ((uart_inst_t *)uart1_hw)
```

Identifier for UART instance 1.

4.1.29.5. Macro Definition Documentation

4.1.29.5.1. UART_NUM

```
#define UART_NUM(uart)
```

Returns the UART number for a UART instance.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.29.5.2. UART_INSTANCE

```
#define UART_INSTANCE(num)
```

Returns the UART instance with the given UART number.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.29.5.3. UART_DREQ_NUM

```
#define UART_DREQ_NUM(uart, is_tx)
```

Returns the dreq_num_t used for pacing DMA transfers to or from this UART instance. If is_tx is true, then it is for transfers to the UART else for transfers from the UART.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.29.5.4. UART_CLOCK_NUM

```
#define UART_CLOCK_NUM(uart)
```

Returns clock_num_t of the clock for the given UART instance.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.29.5.5. UART_FUNCSEL_NUM

```
#define UART_FUNCSEL_NUM(uart, gpio)
```

Returns gpio_function_t needed to select the UART function for the given UART instance on the given GPIO number.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.29.5.6. UART_IRQ_NUM

```
#define UART_IRQ_NUM(uart)
```

Returns the irq_num_t for processor interrupts from the given UART instance.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.29.5.7. UART_RESET_NUM

```
#define UART_RESET_NUM(uart)
```

Returns the reset_num_t used to reset a given UART instance.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.1.29.6. Enumeration Type Documentation

4.1.29.6.1. uart_parity_t

```
enum uart_parity_t
```

UART Parity enumeration.

4.1.29.7. Function Documentation

4.1.29.7.1. uart_default_tx_wait_blocking

```
static void uart_default_tx_wait_blocking (void) [inline], [static]
```

Wait for the default UART's TX FIFO to be drained.

4.1.29.7.2. 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

4.1.29.7.3. uart_get_dreq_num

```
static uint uart_get_dreq_num (uart_inst_t * uart, bool is_tx) [inline], [static]
```

Return the dreq_num_t to use for pacing transfers to/from a particular UART instance.

Parameters

`uart`	UART instance. uart0 or uart1
`is_tx`	true for sending data to the UART instance, false for receiving data from the UART instance

4.1.29.7.4. uart_get_hw

```
static uart_hw_t * uart_get_hw (uart_inst_t * uart) [inline], [static]
```

Get the real hardware UART instance from a UART instance.

This extra level of abstraction was added to facilitate adding PIO UARTs in the future. It currently does nothing, and costs nothing.

Parameters

`uart` UART instance

Returns

The uart_hw_t pointer to the UART instance registers

4.1.29.7.5. uart_get_index

`static uint uart_get_index (uart_inst_t * uart) [inline], [static]`

Convert UART instance to hardware instance number.

Parameters

`uart` UART instance

Returns

Number of UART, 0 or 1

4.1.29.7.6. uart_get_instance

`static uart_inst_t * uart_get_instance (uint num) [inline], [static]`

Get the UART instance from an instance number.

Parameters

`num` Number of UART, 0 or 1

Returns

UART instance

4.1.29.7.7. uart_get_reset_num

`static uint uart_get_reset_num (uart_inst_t * uart) [inline], [static]`

Return the reset_num_t to use to reset a particular UART instance.

Parameters

`uart` UART instance. uart0 or uart1

4.1.29.7.8. uart_getc

`static char uart_getc (uart_inst_t * uart) [inline], [static]`

Read a single character from the UART.

This function will block until a character has been read

Parameters

`uart` UART instance. uart0 or uart1

Returns

The character read.

4.1.29.7.9. uart_init

```
uint 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.

This function always enables the FIFOs, and configures the UART for the following default line format:

- 8 data bits
- No parity bit
- One stop bit

NOTE

There is no guarantee that the baudrate requested will be possible, the nearest will be chosen, and this function will return the configured baud rate.

Parameters

`uart`	UART instance. uart0 or uart1
`baudrate`	Baudrate of UART in Hz

Returns

Actual set baudrate

4.1.29.7.10. uart_is_enabled

```
static bool uart_is_enabled (uart_inst_t * uart) [inline], [static]
```

Test if specific UART is enabled.

Parameters

`uart`	UART instance. uart0 or uart1

Returns

true if the UART is enabled

4.1.29.7.11. uart_is_readable

```
static bool uart_is_readable (uart_inst_t * uart) [inline], [static]
```

Determine whether data is waiting in the RX FIFO.

Parameters

`uart`	UART instance. uart0 or uart1

Returns

true if the RX FIFO is not empty, otherwise false.

4.1.29.7.12. uart_is_readable_within_us

```
bool uart_is_readable_within_us (uart_inst_t * uart, uint32_t us)
```

Wait for up to a certain number of microseconds for the RX FIFO to be non empty.

Parameters

`uart` UART instance. uart0 or uart1

`us` the number of microseconds to wait at most (may be 0 for an instantaneous check)

Returns

true if the RX FIFO became non empty before the timeout, false otherwise

4.1.29.7.13. uart_is_writable

`static bool uart_is_writable (uart_inst_t * uart) [inline], [static]`

Determine if space is available in the TX FIFO.

Parameters

`uart` UART instance. uart0 or uart1

Returns

false if no space available, true otherwise

4.1.29.7.14. uart_putc

`static void uart_putc (uart_inst_t * uart, char c) [inline], [static]`

Write single character to UART for transmission, with optional CR/LF conversions.

This function will block until the character has been sent to the UART transmit buffer

Parameters

`uart` UART instance. uart0 or uart1

`c` The character to send

4.1.29.7.15. uart_putc_raw

`static void uart_putc_raw (uart_inst_t * uart, char c) [inline], [static]`

Write single character to UART for transmission.

This function will block until the entire character has been sent to the UART transmit buffer

Parameters

`uart` UART instance. uart0 or uart1

`c` The character to send

4.1.29.7.16. uart_puts

`static void uart_puts (uart_inst_t * uart, const char * s) [inline], [static]`

Write string to UART for transmission, doing any CR/LF conversions.

This function will block until the entire string has been sent to the UART transmit buffer

Parameters

`uart` UART instance. uart0 or uart1

`s` The null terminated string to send

4.1.29.7.17. uart_read_blocking

`static void uart_read_blocking (uart_inst_t * uart, uint8_t * dst, size_t len) [inline], [static]`

Read from the UART.

This function blocks until len characters have been read from the UART

Parameters

`uart`	UART instance. uart0 or uart1
`dst`	Buffer to accept received bytes
`len`	The number of bytes to receive.

4.1.29.7.18. 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.

The UART is paused for around two character periods whilst the settings are changed. Data received during this time may be dropped by the UART.

Any characters still in the transmit buffer will be sent using the new updated baud rate. uart_tx_wait_blocking() can be called before this function to ensure all characters at the old baud rate have been sent before the rate is changed.

This function should not be called from an interrupt context, and the UART interrupt should be disabled before calling this function.

Parameters

`uart`	UART instance. uart0 or uart1
`baudrate`	Baudrate in Hz

Returns

Actual set baudrate

4.1.29.7.19. uart_set_break

`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.

4.1.29.7.20. uart_set_fifo_enabled

`void uart_set_fifo_enabled (uart_inst_t * uart, bool enabled)`

Enable/Disable the FIFOs on specified UART.

The UART is paused for around two character periods whilst the settings are changed. Data received during this time may be dropped by the UART.

Any characters still in the transmit FIFO will be lost if the FIFO is disabled. uart_tx_wait_blocking() can be called before this function to avoid this.

This function should not be called from an interrupt context, and the UART interrupt should be disabled when calling this function.

Parameters

`uart`	UART instance. uart0 or uart1
`enabled`	true to enable FIFO (default), false to disable

4.1.29.7.21. uart_set_format

`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.

The UART is paused for around two character periods whilst the settings are changed. Data received during this time may be dropped by the UART.

Any characters still in the transmit buffer will be sent using the new updated data format. uart_tx_wait_blocking() can be called before this function to ensure all characters needing the old format have been sent before the format is changed.

This function should not be called from an interrupt context, and the UART interrupt should be disabled before calling this function.

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.

4.1.29.7.22. uart_set_hw_flow

`static void uart_set_hw_flow (uart_inst_t * uart, bool cts, bool rts) [inline], [static]`

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

4.1.29.7.23. uart_set_irqs_enabled

`static void uart_set_irqs_enabled (uart_inst_t * uart, bool rx_has_data, bool tx_needs_data) [inline], [static]`

Enable/Disable UART interrupt outputs.

Enable/Disable the UART’s interrupt outputs. An interrupt handler should 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 contains data.
`tx_needs_data`	If true an interrupt will be fired when the TX FIFO needs data.

4.1.29.7.24. uart_set_translate_crlf

`void uart_set_translate_crlf (uart_inst_t * uart, bool translate)`

Set CR/LF conversion on UART.

Parameters

`uart`	UART instance. uart0 or uart1
`translate`	If true, convert line feeds to carriage return on transmissions

4.1.29.7.25. uart_tx_wait_blocking

`static void uart_tx_wait_blocking (uart_inst_t * uart) [inline], [static]`

Wait for the UART TX fifo to be drained.

Parameters

`uart`	UART instance. uart0 or uart1

4.1.29.7.26. uart_write_blocking

`static void uart_write_blocking (uart_inst_t * uart, const uint8_t * src, size_t len) [inline], [static]`

Write to the UART for transmission.

This function will block until all the data has been sent to the UART transmit buffer hardware. Note: Serial data transmission will continue until the Tx FIFO and the transmit shift register (not programmer-accessible) are empty. To ensure the UART FIFO has been emptied, you can use uart_tx_wait_blocking()

Parameters

`uart`	UART instance. uart0 or uart1
`src`	The bytes to send
`len`	The number of bytes to send

4.1.30. hardware_vreg

Voltage Regulation API.

4.1.30.1. Functions

`void vreg_set_voltage (enum vreg_voltage voltage)`

Set voltage.

`enum vreg_voltage vreg_get_voltage (void)`

Get voltage.

`void vreg_disable_voltage_limit (void)`

Enable use of voltages beyond the safe range of operation.

4.1.30.2. Function Documentation

4.1.30.2.1. vreg_disable_voltage_limit

`void vreg_disable_voltage_limit (void)`

Enable use of voltages beyond the safe range of operation.

This allows voltages beyond VREG_VOLTAGE_MAX to be used, on platforms where they are available (e.g. RP2350). Attempting to set a higher voltage without first calling this function will result in a voltage of VREG_VOLTAGE_MAX.

4.1.30.2.2. vreg_get_voltage

`enum vreg_voltage vreg_get_voltage (void)`

Get voltage.

Returns

The current voltage (from enumeration vreg_voltage) of the voltage regulator

4.1.30.2.3. 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

4.1.31. hardware_watchdog

Hardware Watchdog Timer API.

4.1.31.1. Detailed Description

Supporting functions for the Pico hardware watchdog timer.

The RP-series microcontrollers have 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

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

int main() {
    stdio_init_all();

    if (watchdog_enable_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
```

```c
    // 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);
}
```

4.1.31.2. Functions

`void watchdog_reboot (uint32_t pc, uint32_t sp, uint32_t delay_ms)`

Define actions to perform at watchdog timeout.

`void watchdog_start_tick (uint cycles)`

Start the watchdog tick.

`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.

`void watchdog_disable (void)`

Disable the watchdog.

`bool watchdog_caused_reboot (void)`

Did the watchdog cause the last reboot?

`bool watchdog_enable_caused_reboot (void)`

Did watchdog_enable cause the last reboot?

`uint32_t watchdog_get_time_remaining_ms (void)`

Returns the number of microseconds before the watchdog will reboot the chip.

4.1.31.3. Function Documentation

4.1.31.3.1. 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

Returns

false If there has been no watchdog reboot since the last power on reset. A power on reset is typically caused by a power cycle or the run pin (reset button) being toggled.

4.1.31.3.2. watchdog_disable

```
void watchdog_disable (void)
```

Disable the watchdog.

4.1.31.3.3. watchdog_enable

```
void watchdog_enable (uint32_t delay_ms, bool pause_on_debug)
```

Enable the watchdog.

NOTE

> If watchdog_start_tick value does not give a 1MHz clock to the watchdog system, then the `delay_ms` parameter will not be in milliseconds. See the datasheet for more details.

By default the SDK assumes a 12MHz XOSC and sets the watchdog_start_tick appropriately.

This method sets a marker in the watchdog scratch register 4 that is checked by watchdog_enable_caused_reboot. If the device is subsequently reset via a call to watchdog_reboot (including for example by dragging a UF2 onto the RPI-RP2), then this value will be cleared, and so watchdog_enable_caused_reboot will return false.

Parameters

`delay_ms`	Number of milliseconds before watchdog will reboot without watchdog_update being called. Maximum of 8388, which is approximately 8.3 seconds
`pause_on_debug`	If the watchdog should be paused when the debugger is stepping through code

4.1.31.3.4. watchdog_enable_caused_reboot

```
bool watchdog_enable_caused_reboot (void)
```

Did watchdog_enable cause the last reboot?

Perform additional checking along with watchdog_caused_reboot to determine if a watchdog timeout initiated by watchdog_enable caused the last reboot.

This method checks for a special value in watchdog scratch register 4 placed there by watchdog_enable. This would not be present if a watchdog reset is initiated by watchdog_reboot or by the RP-series microcontroller bootrom (e.g. dragging a UF2 onto the RPI-RP2 drive).

Returns

true If the watchdog timer or a watchdog force caused (see watchdog_caused_reboot) the last reboot and the watchdog reboot happened after watchdog_enable was called

Returns

false If there has been no watchdog reboot since the last power on reset, or the watchdog reboot was not caused by a watchdog timeout after watchdog_enable was called. A power on reset is typically caused by a power cycle or the run pin (reset button) being toggled.

4.1.31.3.5. watchdog_get_time_remaining_ms

```
uint32_t watchdog_get_time_remaining_ms (void)
```

Returns the number of microseconds before the watchdog will reboot the chip.

Returns

The number of microseconds before the watchdog will reboot the chip.

4.1.31.3.6. watchdog_reboot

`void watchdog_reboot (uint32_t pc, uint32_t sp, uint32_t delay_ms)`

Define actions to perform at watchdog timeout.

> **NOTE**
>
> If watchdog_start_tick value does not give a 1MHz clock to the watchdog system, then the `delay_ms` parameter will not be in milliseconds. See the datasheet for more details.

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 8388, approximately 8.3s.

4.1.31.3.7. 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.

4.1.31.3.8. watchdog_update

`void watchdog_update (void)`

Reload the watchdog counter with the amount of time set in watchdog_enable.

4.1.32. hardware_xip_cache

Low-level cache maintenance operations for the XIP cache.

4.1.32.1. Detailed Description

These functions apply some maintenance operation to either the entire cache contents, or a range of offsets within the downstream address space. Offsets start from 0 (indicating the first byte of flash), so pointers should have XIP_BASE subtracted before passing into one of these functions.

The only valid cache maintenance operation on RP2040 is "invalidate", which tells the cache to forget everything it knows about some address. This is necessary after a programming operation, because the cache does not automatically know about any serial programming operations performed on the external flash device, and could return stale data.

On RP2350, the three types of operation are:

- Invalidate: tell the cache to forget everything it knows about some address. The next access to that address will fetch from downstream memory.
- Clean: if the addressed cache line contains data not yet written to external memory, then write that data out now, and mark the line as "clean" (i.e. not containing uncommitted write data)

- Pin: mark an address as always being resident in the cache. This persists until the line is invalidated, and can be used to allocate part of the cache for cache-as-SRAM use.

When using both external flash and external RAM (e.g. PSRAM), a simple way to maintain coherence over flash programming operations is to:

1. Clean the entire cache (e.g. using xip_cache_clean_all())
2. Erase + program the flash using serial SPI commands
3. Invalidate ("flush") the entire cache (e.g. using xip_cache_invalidate_all())

The invalidate ensures the programming is visible to subsequent reads. The clean ensures that the invalidate does not discard any cached PSRAM write data.

4.1.32.2. Functions

`void xip_cache_invalidate_all (void)`

Invalidate the cache for the entire XIP address space.

`void xip_cache_invalidate_range (uintptr_t start_offset, uintptr_t size_bytes)`

Invalidate a range of offsets within the XIP address space.

`void xip_cache_clean_all (void)` RP2350

Clean the cache for the entire XIP address space.

`void xip_cache_clean_range (uintptr_t start_offset, uintptr_t size_bytes)` RP2350

Clean a range of offsets within the XIP address space.

`void xip_cache_pin_range (uintptr_t start_offset, uintptr_t size_bytes)` RP2350

Pin a range of offsets within the XIP address space.

4.1.32.3. Function Documentation

4.1.32.3.1. xip_cache_clean_all RP2350

`void xip_cache_clean_all (void)`

Clean the cache for the entire XIP address space.

This causes the cache to write out all pending write data to the downstream memory. For example, when suspending the system with state retained in external PSRAM, this ensures all data has made it out to external PSRAM before powering down.

This function is faster than calling xip_cache_clean_range() for the entire address space, because it iterates over cachelines instead of addresses.

On RP2040 this is a no-op, as the XIP cache is read-only. This is indicated by the XIP_CACHE_IS_READ_ONLY macro.

On RP2350, due to the workaround applied for RP2350-E11, this function also effectively invalidates all cache lines after cleaning them. The next access to each line will miss. Avoid this by calling xip_cache_clean_range() which does not suffer this issue.

4.1.32.3.2. xip_cache_clean_range RP2350

`void xip_cache_clean_range (uintptr_t start_offset, uintptr_t size_bytes)`

Clean a range of offsets within the XIP address space.

This causes the cache to write out pending write data at these offsets to the downstream memory.

On RP2040 this is a no-op, as the XIP cache is read-only. This is indicated by the XIP_CACHE_IS_READ_ONLY macro.

Parameters

`start_offset`	The first offset to be invalidated. Offset 0 means the first byte of XIP memory (e.g. flash). Pointers must have XIP_BASE subtracted before passing into this function. Must be aligned to the start of a cache line (XIP_CACHE_LINE_SIZE).
`size_bytes`	The number of bytes to clean. Must be a multiple of XIP_CACHE_LINE_SIZE.

4.1.32.3.3. xip_cache_invalidate_all

```
void xip_cache_invalidate_all (void)
```

Invalidate the cache for the entire XIP address space.

Invalidation ensures that subsequent reads will fetch data from the downstream memory, rather than using (potentially stale) cached data.

This function is faster than calling xip_cache_invalidate_range() for the entire address space, because it iterates over cachelines instead of addresses.

> **NOTE**
>
> Any pending write data held in the cache is lost: you can force the cache to commit these writes first, by calling xip_cache_clean_all()
>
> Unlike flash_flush_cache(), this function affects *only* the cache line state. flash_flush_cache() calls a ROM API which can have other effects on some platforms, like cleaning up the bootrom's QSPI GPIO setup on RP2040. Prefer this function for general cache maintenance use, and prefer flash_flush_cache in sequences of ROM flash API calls.

4.1.32.3.4. xip_cache_invalidate_range

```
void xip_cache_invalidate_range (uintptr_t start_offset, uintptr_t size_bytes)
```

Invalidate a range of offsets within the XIP address space.

Parameters

`start_offset`	The first offset to be invalidated. Offset 0 means the first byte of XIP memory (e.g. flash). Pointers must have XIP_BASE subtracted before passing into this function. Must be 4-byte-aligned on RP2040. Must be a aligned to the start of a cache line (XIP_CACHE_LINE_SIZE) on other platforms.
`size_bytes`	The number of bytes to invalidate. Must be a multiple of 4 bytes on RP2040. Must be a multiple of XIP_CACHE_LINE_SIZE on other platforms.

Invalidation ensures that subsequent reads will fetch data from the downstream memory, rather than using (potentially stale) cached data.

> **NOTE**
>
> Any pending write data held in the cache is lost: you can force the cache to commit these writes first, by calling xip_cache_clean_range() with the same parameters. Generally this is not necessary because invalidation is used with flash (write-behind via programming), and cleaning is used with PSRAM (writing through the cache).

4.1.32.3.5. xip_cache_pin_range RP2350

```
void xip_cache_pin_range (uintptr_t start_offset, uintptr_t size_bytes)
```

Pin a range of offsets within the XIP address space.

Pinning a line at an address allocates the line exclusively for use at that address. This means that all subsequent accesses to that address will hit the cache, and will not go to downstream memory. This persists until one of two things happens:

- The line is invalidated, e.g. via xip_cache_invalidate_all()
- The same line is pinned at a different address (note lines are selected by address modulo XIP_CACHE_SIZE)

Parameters

`start_offset`	The first offset to be pinnned. Offset 0 means the first byte of XIP memory (e.g. flash). Pointers must have XIP_BASE subtracted before passing into this function. Must be aligned to the start of a cache line (XIP_CACHE_LINE_SIZE).
`size_bytes`	The number of bytes to pin. Must be a multiple of XIP_CACHE_LINE_SIZE.

4.1.33. hardware_xosc

Crystal Oscillator (XOSC) API.

4.1.33.1. Functions

`void xosc_init (void)`

Initialise the crystal oscillator system.

`void xosc_disable (void)`

Disable the Crystal oscillator.

`void xosc_dormant (void)`

Set the crystal oscillator system to dormant.

4.1.33.2. Function Documentation

4.1.33.2.1. xosc_disable

`void xosc_disable (void)`

Disable the Crystal oscillator.

Turns off the crystal oscillator source, and waits for it to become unstable

4.1.33.2.2. 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. This 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.

4.1.33.2.3. xosc_init

`void xosc_init (void)`

Initialise the crystal oscillator system.

This function will block until the crystal oscillator has stabilised.

4.2. High Level APIs

This group of libraries provide higher level functionality that isn't hardware related or provides a richer set of functionality above the basic hardware interfaces

pico_aon_timer	High Level "Always on Timer" Abstraction.
pico_async_context	An async_context provides a logically single-threaded context for performing work, and responding to asynchronous events. Thus an async_context instance is suitable for servicing third-party libraries that are not re-entrant.
async_context_freertos	async_context_freertos provides an implementation of async_context that handles asynchronous work in a separate FreeRTOS task.
async_context_poll	async_context_poll provides an implementation of async_context that is intended for use with a simple polling loop on one core. It is not thread safe.
async_context_threadsafe_background	async_context_threadsafe_background provides an implementation of async_context that handles asynchronous work in a low priority IRQ, and there is no need for the user to poll for work
pico_bootsel_via_double_reset	Optional support to make fast double reset of the system enter BOOTSEL mode.
pico_flash	High level flash API.
pico_i2c_slave	Functions providing an interrupt driven I2C slave interface.
pico_multicore	Adds support for running code on, and interacting with the second processor core (core 1).
fifo	Functions for the inter-core FIFOs.
doorbell	Functions related to doorbells which a core can use to raise IRQs on itself or the other core.
lockout	Functions to enable one core to force the other core to pause execution in a known state.
pico_rand	Random Number Generator API.
pico_sha256	SHA-256 Hardware Accelerated implementation. RP2350
pico_stdlib	Aggregation of a core subset of Raspberry Pi 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.
lock_core	base synchronization/lock primitive support.
mutex	Mutex API for non IRQ mutual exclusion between cores.
sem	Semaphore API for restricting access to a resource.
pico_time	API for accurate timestamps, sleeping, and time based callbacks.
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.
pico_unique_id	Unique device ID access API.
pico_util	Useful data structures and utility functions.
datetime	Date/Time formatting.

pheap	Pairing Heap Implementation.
queue	Multi-core and IRQ safe queue implementation.

4.2.1. pico_aon_timer

High Level "Always on Timer" Abstraction.

4.2.1.1. Detailed Description

This library uses the RTC on RP2040. This library uses the Powman Timer on RP2350.

This library supports both `aon_timer_xxx_calendar()` methods which use a calendar date/time (as struct tm), and `aon_timer_xxx()` methods which use a linear time value relative an internal reference time (via struct timespec).

On RP2040 the non 'calendar date/time' methods must convert the linear time value to a calendar date/time internally; these methods are:

- aon_timer_start_with_timeofday
- aon_timer_start
- aon_timer_set_time
- aon_timer_get_time
- aon_timer_enable_alarm

This conversion is handled by the pico_localtime_r method. By default, this pulls in the C library `local_time_r` method which can lead to a big increase in binary size. The default implementation of `pico_localtime_r` is weak, so it can be overridden if a better/smaller alternative is available, otherwise you might consider the method variants ending in `_calendar()` instead on RP2040.

On RP2350 the 'calendar date/time' methods must convert the calendar date/time to a linear time value internally; these methods are:

- aon_timer_start_calendar
- aon_timer_set_time_calendar
- aon_timer_get_time_calendar
- aon_timer_enable_alarm_calendar

This conversion is handled by the pico_mktime method. By default, this pulls in the C library `mktime` method which can lead to a big increase in binary size. The default implementation of `pico_mktime` is weak, so it can be overridden if a better/smaller alternative is available, otherwise you might consider the method variants not ending in `_calendar()` instead on RP2350.

4.2.1.2. Functions

`void aon_timer_start_with_timeofday (void)`

Start the AON timer running using the result from the gettimeofday() function as the current time.

`bool aon_timer_start (const struct timespec *ts)`

Start the AON timer running using the specified timespec as the current time.

`bool aon_timer_start_calendar (const struct tm *tm)`

Start the AON timer running using the specified calendar date/time as the current time.

`void aon_timer_stop (void)`

Stop the AON timer.

`bool aon_timer_set_time (const struct timespec *ts)`

Set the current time of the AON timer.

`bool aon_timer_set_time_calendar (const struct tm *tm)`

Set the current time of the AON timer to the given calendar date/time.

`bool aon_timer_get_time (struct timespec *ts)`

Get the current time of the AON timer.

`bool aon_timer_get_time_calendar (struct tm *tm)`

Get the current time of the AON timer as a calendar date/time.

`void aon_timer_get_resolution (struct timespec *ts)`

Get the resolution of the AON timer.

`aon_timer_alarm_handler_t aon_timer_enable_alarm (const struct timespec *ts, aon_timer_alarm_handler_t handler, bool wakeup_from_low_power)`

Enable an AON timer alarm for a specified time.

`aon_timer_alarm_handler_t aon_timer_enable_alarm_calendar (const struct tm *tm, aon_timer_alarm_handler_t handler, bool wakeup_from_low_power)`

Enable an AON timer alarm for a specified calendar date/time.

`void aon_timer_disable_alarm (void)`

Disable the currently enabled AON timer alarm if any.

`bool aon_timer_is_running (void)`

Check if the AON timer is running.

4.2.1.3. Function Documentation

4.2.1.3.1. aon_timer_disable_alarm

`void aon_timer_disable_alarm (void)`

Disable the currently enabled AON timer alarm if any.

4.2.1.3.2. aon_timer_enable_alarm

`aon_timer_alarm_handler_t aon_timer_enable_alarm (const struct timespec * ts, aon_timer_alarm_handler_t handler, bool wakeup_from_low_power)`

Enable an AON timer alarm for a specified time.

On RP2350 the alarm will fire if it is in the past On RP2040 the alarm will not fire if it is in the past.

See caveats for using this method on RP2040

Parameters

`ts`	the alarm time
`handler`	a callback to call when the timer fires (can be NULL for wakeup_from_low_power = true)
`wakeup_from_low_power`	true if the AON timer is to be used to wake up from a DORMANT state

Returns

on success the old handler (or NULL if there was none) or PICO_ERROR_INVALID_ARG if internal time format conversion failed

See also

pico_localtime_r

4.2.1.3.3. aon_timer_enable_alarm_calendar

```
aon_timer_alarm_handler_t aon_timer_enable_alarm_calendar (const struct tm * tm, aon_timer_alarm_handler_t handler, bool wakeup_from_low_power)
```

Enable an AON timer alarm for a specified calendar date/time.

On RP2350 the alarm will fire if it is in the past

See caveats for using this method on RP2350

On RP2040 the alarm will not fire if it is in the past.

Parameters

`tm`	the alarm calendar date/time
`handler`	a callback to call when the timer fires (can be NULL for wakeup_from_low_power = true)
`wakeup_from_low_power`	true if the AON timer is to be used to wake up from a DORMANT state

Returns

on success the old handler (or NULL if there was none) or PICO_ERROR_INVALID_ARG if internal time format conversion failed

See also

pico_localtime_r

4.2.1.3.4. aon_timer_get_resolution

```
void aon_timer_get_resolution (struct timespec * ts)
```

Get the resolution of the AON timer.

Parameters

`ts`	out value for the resolution of the AON timer

4.2.1.3.5. aon_timer_get_time

```
bool aon_timer_get_time (struct timespec * ts)
```

Get the current time of the AON timer.

See caveats for using this method on RP2040

Parameters

`ts`	out value for the current time

Returns

true on success, false if internal time format conversion failed

See also

aon_timer_get_time_calendar

4.2.1.3.6. aon_timer_get_time_calendar

`bool aon_timer_get_time_calendar (struct tm * tm)`

Get the current time of the AON timer as a calendar date/time.

See caveats for using this method on RP2350

Parameters

`tm`	out value for the current calendar date/time

Returns

true on success, false if internal time format conversion failed

See also

aon_timer_get_time

4.2.1.3.7. aon_timer_is_running

`bool aon_timer_is_running (void)`

Check if the AON timer is running.

Returns

true if the AON timer is running

4.2.1.3.8. aon_timer_set_time

`bool aon_timer_set_time (const struct timespec * ts)`

Set the current time of the AON timer.

See caveats for using this method on RP2040

Parameters

`ts`	the new current time

Returns

true on success, false if internal time format conversion failed

See also

aon_timer_set_time_calendar

4.2.1.3.9. aon_timer_set_time_calendar

`bool aon_timer_set_time_calendar (const struct tm * tm)`

Set the current time of the AON timer to the given calendar date/time.

See caveats for using this method on RP2350

Parameters

`tm`	the new current time

Returns

true on success, false if internal time format conversion failed

See also

aon_timer_set_time

4.2.1.3.10. aon_timer_start

`bool aon_timer_start (const struct timespec * ts)`

Start the AON timer running using the specified timespec as the current time.

See caveats for using this method on RP2040

Parameters

`ts`	the time to set as 'now'

Returns

true on success, false if internal time format conversion failed

See also

aon_timer_start_calendar

4.2.1.3.11. aon_timer_start_calendar

`bool aon_timer_start_calendar (const struct tm * tm)`

Start the AON timer running using the specified calendar date/time as the current time.

See caveats for using this method on RP2350

Parameters

`tm`	the calendar date/time to set as 'now'

Returns

true on success, false if internal time format conversion failed

See also

aon_timer_start

4.2.1.3.12. aon_timer_start_with_timeofday

`void aon_timer_start_with_timeofday (void)`

Start the AON timer running using the result from the gettimeofday() function as the current time.

See caveats for using this method on RP2040

4.2.1.3.13. aon_timer_stop

`void aon_timer_stop (void)`

Stop the AON timer.

4.2.2. pico_async_context

An async_context provides a logically single-threaded context for performing work, and responding to asynchronous events. Thus an async_context instance is suitable for servicing third-party libraries that are not re-entrant.

4.2.2.1. Detailed Description

The "context" in async_context refers to the fact that when calling workers or timeouts within the async_context various pre-conditions hold:

1. That there is a single logical thread of execution; i.e. that the context does not call any worker functions concurrently.
2. That the context always calls workers from the same processor core, as most uses of async_context rely on interaction with IRQs which are themselves core-specific.

THe async_context provides two mechanisms for asynchronous work:

- *when_pending* workers, which are processed whenever they have work pending. See async_context_add_when_pending_worker, async_context_remove_when_pending_worker, and async_context_set_work_pending, the latter of which can be used from an interrupt handler to signal that servicing work is required to be performed by the worker from the regular async_context.
- *at_time* workers, that are executed after at a specific time.

Note: "when pending" workers with work pending are executed before "at time" workers.

The async_context provides locking mechanisms, see async_context_acquire_lock_blocking, async_context_release_lock and async_context_lock_check which can be used by external code to ensure execution of external code does not happen concurrently with worker code. Locked code runs on the calling core, however async_context_execute_sync is provided to synchronously run a function from the core of the async_context.

The SDK ships with the following default async_contexts:

async_context_poll - this context is not thread-safe, and the user is responsible for calling async_context_poll() periodically, and can use async_context_wait_for_work_until() to sleep between calls until work is needed if the user has nothing else to do.

async_context_threadsafe_background - in order to work in the background, a low priority IRQ is used to handle callbacks. Code is usually invoked from this IRQ context, but may be invoked after any other code that uses the async context in another (non-IRQ) context on the same core. Calling async_context_poll() is not required, and is a no-op. This context implements async_context locking and is thus safe to call from either core, according to the specific notes on each API.

async_context_freertos - Work is performed from a separate "async_context" task, however once again, code may also be invoked after a direct use of the async_context on the same core that the async_context belongs to. Calling async_context_poll() is not required, and is a no-op. This context implements async_context locking and is thus safe to call from any task, and from either core, according to the specific notes on each API.

Each async_context provides bespoke methods of instantiation which are provided in the corresponding headers (e.g. async_context_poll.h, async_context_threadsafe_background.h, asycn_context_freertos.h). async_contexts are de-initialized by the common async_context_deint() method.

Multiple async_context instances can be used by a single application, and they will operate independently.

4.2.2.2. Modules

async_context_freertos

async_context_freertos provides an implementation of async_context that handles asynchronous work in a separate FreeRTOS task.

async_context_poll

async_context_poll provides an implementation of async_context that is intended for use with a simple polling loop on one core. It is not thread safe.

async_context_threadsafe_background

async_context_threadsafe_background provides an implementation of async_context that handles asynchronous work in a low priority IRQ, and there is no need for the user to poll for work

4.2.2.3. Typedefs

`typedef struct async_work_on_timeout async_at_time_worker_t`

A "timeout" instance used by an async_context.

`typedef struct async_when_pending_worker async_when_pending_worker_t`

A "worker" instance used by an async_context.

`typedef struct async_context_type async_context_type_t`

Implementation of an async_context type, providing methods common to that type.

4.2.2.4. Functions

`static void async_context_acquire_lock_blocking (async_context_t *context)`

Acquire the async_context lock.

`static void async_context_release_lock (async_context_t *context)`

Release the async_context lock.

`static void async_context_lock_check (async_context_t *context)`

Assert if the caller does not own the lock for the async_context.

`static uint32_t async_context_execute_sync (async_context_t *context, uint32_t(*func)(void *param), void *param)`

Execute work synchronously on the core the async_context belongs to.

`static bool async_context_add_at_time_worker (async_context_t *context, async_at_time_worker_t *worker)`

Add an "at time" worker to a context.

`static bool async_context_add_at_time_worker_at (async_context_t *context, async_at_time_worker_t *worker, absolute_time_t at)`

Add an "at time" worker to a context.

`static bool async_context_add_at_time_worker_in_ms (async_context_t *context, async_at_time_worker_t *worker, uint32_t ms)`

Add an "at time" worker to a context.

`static bool async_context_remove_at_time_worker (async_context_t *context, async_at_time_worker_t *worker)`

Remove an "at time" worker from a context.

`static bool async_context_add_when_pending_worker (async_context_t *context, async_when_pending_worker_t *worker)`

Add a "when pending" worker to a context.

`static bool async_context_remove_when_pending_worker (async_context_t *context, async_when_pending_worker_t *worker)`

Remove a "when pending" worker from a context.

`static void async_context_set_work_pending (async_context_t *context, async_when_pending_worker_t *worker)`

Mark a "when pending" worker as having work pending.

`static void async_context_poll (async_context_t *context)`

Perform any pending work for polling style async_context.

`static void async_context_wait_until (async_context_t *context, absolute_time_t until)`

sleep until the specified time in an async_context callback safe way

`static void async_context_wait_for_work_until (async_context_t *context, absolute_time_t until)`

Block until work needs to be done or the specified time has been reached.

`static void async_context_wait_for_work_ms (async_context_t *context, uint32_t ms)`

Block until work needs to be done or the specified number of milliseconds have passed.

`static uint async_context_core_num (const async_context_t *context)`

Return the processor core this async_context belongs to.

`static void async_context_deinit (async_context_t *context)`

End async_context processing, and free any resources.

4.2.2.5. Typedef Documentation

4.2.2.5.1. async_at_time_worker_t

`typedef struct async_work_on_timeout async_at_time_worker_t`

A "timeout" instance used by an async_context.

A "timeout" represents some future action that must be taken at a specific time. Its methods are called from the async_context under lock at the given time

See also

async_context_add_worker_at

async_context_add_worker_in_ms

4.2.2.5.2. async_when_pending_worker_t

`typedef struct async_when_pending_worker async_when_pending_worker_t`

A "worker" instance used by an async_context.

A "worker" represents some external entity that must do work in response to some external stimulus (usually an IRQ). Its methods are called from the async_context under lock at the given time

See also

async_context_add_worker_at

async_context_add_worker_in_ms

4.2.2.5.3. async_context_type_t

`typedef struct async_context_type async_context_type_t`

Implementation of an async_context type, providing methods common to that type.

4.2.2.6. Function Documentation

4.2.2.6.1. async_context_acquire_lock_blocking

`static void async_context_acquire_lock_blocking (async_context_t * context) [inline], [static]`

Acquire the async_context lock.

The owner of the async_context lock is the logic owner of the async_context and other work related to this async_context will not happen concurrently.

This method may be called in a nested fashion by the the lock owner.

NOTE

the async_context lock is nestable by the same caller, so an internal count is maintained

for async_contexts that provide locking (not async_context_poll), this method is threadsafe. and may be called from within any worker method called by the async_context or from any other non-IRQ context.

Parameters

`context`	the async_context

See also

async_context_release_lock

4.2.2.6.2. async_context_add_at_time_worker

```
static bool async_context_add_at_time_worker (async_context_t * context, async_at_time_worker_t * worker) [inline],
[static]
```

Add an "at time" worker to a context.

An "at time" worker will run at or after a specific point in time, and is automatically when (just before) it runs.

The time to fire is specified in the next_time field of the worker.

NOTE

for async_contexts that provide locking (not async_context_poll), this method is threadsafe. and may be called from within any worker method called by the async_context or from any other non-IRQ context.

Parameters

`context`	the async_context
`worker`	the "at time" worker to add

Returns

true if the worker was added, false if the worker was already present.

4.2.2.6.3. async_context_add_at_time_worker_at

```
static bool async_context_add_at_time_worker_at (async_context_t * context, async_at_time_worker_t * worker,
absolute_time_t at) [inline], [static]
```

Add an "at time" worker to a context.

An "at time" worker will run at or after a specific point in time, and is automatically when (just before) it runs.

The time to fire is specified by the at parameter.

NOTE

for async_contexts that provide locking (not async_context_poll), this method is threadsafe. and may be called from within any worker method called by the async_context or from any other non-IRQ context.

Parameters

`context`	the async_context
`worker`	the "at time" worker to add

`at`	the time to fire at

Returns

true if the worker was added, false if the worker was already present.

4.2.2.6.4. async_context_add_at_time_worker_in_ms

`static bool async_context_add_at_time_worker_in_ms (async_context_t * context, async_at_time_worker_t * worker, uint32_t ms) [inline], [static]`

Add an "at time" worker to a context.

An "at time" worker will run at or after a specific point in time, and is automatically when (just before) it runs.

The time to fire is specified by a delay via the ms parameter

> **NOTE**
>
> for async_contexts that provide locking (not async_context_poll), this method is threadsafe. and may be called from within any worker method called by the async_context or from any other non-IRQ context.

Parameters

`context`	the async_context
`worker`	the "at time" worker to add
`ms`	the number of milliseconds from now to fire after

Returns

true if the worker was added, false if the worker was already present.

4.2.2.6.5. async_context_add_when_pending_worker

`static bool async_context_add_when_pending_worker (async_context_t * context, async_when_pending_worker_t * worker) [inline], [static]`

Add a "when pending" worker to a context.

An "when pending" worker will run when it is pending (can be set via async_context_set_work_pending), and is NOT automatically removed when it runs.

The time to fire is specified by a delay via the ms parameter

> **NOTE**
>
> for async_contexts that provide locking (not async_context_poll), this method is threadsafe. and may be called from within any worker method called by the async_context or from any other non-IRQ context.

Parameters

`context`	the async_context
`worker`	the "when pending" worker to add

Returns

true if the worker was added, false if the worker was already present.

4.2.2.6.6. async_context_core_num

`static uint async_context_core_num (const async_context_t * context) [inline], [static]`

Return the processor core this async_context belongs to.

Parameters

`context`	the async_context

Returns

the physical core number

4.2.2.6.7. async_context_deinit

`static void async_context_deinit (async_context_t * context) [inline], [static]`

End async_context processing, and free any resources.

Note the user should clean up any resources associated with workers in the async_context themselves.

Asynchronous (non-polled) async_contexts guarantee that no callback is being called once this method returns.

Parameters

`context`	the async_context

4.2.2.6.8. async_context_execute_sync

`static uint32_t async_context_execute_sync (async_context_t * context, uint32_t(*)(void *param) func, void * param) [inline], [static]`

Execute work synchronously on the core the async_context belongs to.

This method is intended for code external to the async_context (e.g. another thread/task) to execute a function with the same guarantees (single core, logical thread of execution) that async_context workers are called with.

NOTE

> you should NOT call this method while holding the async_context's lock

Parameters

`context`	the async_context
`func`	the function to call
`param`	the parameter to pass to the function

Returns

the return value from func

4.2.2.6.9. async_context_lock_check

`static void async_context_lock_check (async_context_t * context) [inline], [static]`

Assert if the caller does not own the lock for the async_context.

> **NOTE**
>
> this method is thread-safe

Parameters

`context`	the async_context

4.2.2.6.10. async_context_poll

`static void async_context_poll (async_context_t * context) [inline], [static]`

Perform any pending work for polling style async_context.

For a polled async_context (e.g. async_context_poll) the user is responsible for calling this method periodically to perform any required work.

This method may immediately perform outstanding work on other context types, but is not required to.

Parameters

`context`	the async_context

4.2.2.6.11. async_context_release_lock

`static void async_context_release_lock (async_context_t * context) [inline], [static]`

Release the async_context lock.

> **NOTE**
>
> the async_context lock may be called in a nested fashion, so an internal count is maintained. On the outermost release, When the outermost lock is released, a check is made for work which might have been skipped while the lock was held, and any such work may be performed during this call IF the call is made from the same core that the async_context belongs to.
>
> for async_contexts that provide locking (not async_context_poll), this method is threadsafe. and may be called from within any worker method called by the async_context or from any other non-IRQ context.

Parameters

`context`	the async_context

See also

async_context_acquire_lock_blocking

4.2.2.6.12. async_context_remove_at_time_worker

`static bool async_context_remove_at_time_worker (async_context_t * context, async_at_time_worker_t * worker) [inline], [static]`

Remove an "at time" worker from a context.

> **NOTE**
>
> for async_contexts that provide locking (not async_context_poll), this method is threadsafe. and may be called from within any worker method called by the async_context or from any other non-IRQ context.

Parameters

`context`	the async_context
`worker`	the "at time" worker to remove

Returns

true if the worker was removed, false if the instance not present.

4.2.2.6.13. async_context_remove_when_pending_worker

```
static bool async_context_remove_when_pending_worker (async_context_t * context, async_when_pending_worker_t * worker)
[inline], [static]
```

Remove a "when pending" worker from a context.

> **NOTE**
>
> for async_contexts that provide locking (not async_context_poll), this method is threadsafe. and may be called from within any worker method called by the async_context or from any other non-IRQ context.

Parameters

`context`	the async_context
`worker`	the "when pending" worker to remove

Returns

true if the worker was removed, false if the instance not present.

4.2.2.6.14. async_context_set_work_pending

```
static void async_context_set_work_pending (async_context_t * context, async_when_pending_worker_t * worker) [inline],
[static]
```

Mark a "when pending" worker as having work pending.

The worker will be run from the async_context at a later time.

> **NOTE**
>
> this method may be called from any context including IRQs

Parameters

`context`	the async_context
`worker`	the "when pending" worker to mark as pending.

4.2.2.6.15. async_context_wait_for_work_ms

```
static void async_context_wait_for_work_ms (async_context_t * context, uint32_t ms) [inline], [static]
```

Block until work needs to be done or the specified number of milliseconds have passed.

NOTE

this method should not be called from a worker callback

Parameters

`context` the async_context

`ms` the number of milliseconds to return after if no work is required

4.2.2.6.16. async_context_wait_for_work_until

`static void async_context_wait_for_work_until (async_context_t * context, absolute_time_t until) [inline], [static]`

Block until work needs to be done or the specified time has been reached.

NOTE

this method should not be called from a worker callback

Parameters

`context` the async_context

`until` the time to return at if no work is required

4.2.2.6.17. async_context_wait_until

`static void async_context_wait_until (async_context_t * context, absolute_time_t until) [inline], [static]`

sleep until the specified time in an async_context callback safe way

NOTE

for async_contexts that provide locking (not async_context_poll), this method is threadsafe. and may be called from within any worker method called by the async_context or from any other non-IRQ context.

Parameters

`context` the async_context

`until` the time to sleep until

4.2.2.7. async_context_freertos

async_context_freertos provides an implementation of async_context that handles asynchronous work in a separate FreeRTOS task.

4.2.2.7.1. Functions

`bool async_context_freertos_init (async_context_freertos_t *self, async_context_freertos_config_t *config)`

Initialize an async_context_freertos instance using the specified configuration.

`static async_context_freertos_config_t async_context_freertos_default_config (void)`

Return a copy of the default configuration object used by async_context_freertos_init_with_defaults()

`static bool async_context_freertos_init_with_defaults (async_context_freertos_t *self)`

Initialize an async_context_freertos instance with default values.

4.2.2.7.2. Function Documentation

async_context_freertos_default_config

`static async_context_freertos_config_t async_context_freertos_default_config (void) [inline], [static]`

Return a copy of the default configuration object used by async_context_freertos_init_with_defaults()

The caller can then modify just the settings it cares about, and call async_context_freertos_init()

Returns

the default configuration object

async_context_freertos_init

`bool async_context_freertos_init (async_context_freertos_t * self, async_context_freertos_config_t * config)`

Initialize an async_context_freertos instance using the specified configuration.

If this method succeeds (returns true), then the async_context is available for use and can be de-initialized by calling async_context_deinit().

Parameters

`self`	a pointer to async_context_freertos structure to initialize
`config`	the configuration object specifying characteristics for the async_context

Returns

true if initialization is successful, false otherwise

async_context_freertos_init_with_defaults

`static bool async_context_freertos_init_with_defaults (async_context_freertos_t * self) [inline], [static]`

Initialize an async_context_freertos instance with default values.

If this method succeeds (returns true), then the async_context is available for use and can be de-initialized by calling async_context_deinit().

Parameters

`self`	a pointer to async_context_freertos structure to initialize

Returns

true if initialization is successful, false otherwise

4.2.2.8. async_context_poll

async_context_poll provides an implementation of async_context that is intended for use with a simple polling loop on one core. It is not thread safe.

4.2.2.8.1. Detailed Description

The async_context_poll() method must be called periodically to handle asynchronous work that may now be pending. async_context_wait_for_work_until() may be used to block a polling loop until there is work to do, and prevent tight spinning.

4.2.2.8.2. Functions

`bool async_context_poll_init_with_defaults (async_context_poll_t *self)`

Initialize an async_context_poll instance with default values.

4.2.2.8.3. Function Documentation

async_context_poll_init_with_defaults

`bool async_context_poll_init_with_defaults (async_context_poll_t * self)`

Initialize an async_context_poll instance with default values.

If this method succeeds (returns true), then the async_context is available for use and can be de-initialized by calling async_context_deinit().

Parameters

`self`	a pointer to async_context_poll structure to initialize

Returns

true if initialization is successful, false otherwise

4.2.2.9. async_context_threadsafe_background

async_context_threadsafe_background provides an implementation of async_context that handles asynchronous work in a low priority IRQ, and there is no need for the user to poll for work

4.2.2.9.1. Detailed Description

> **NOTE**
>
> The workers used with this async_context MUST be safe to call from an IRQ.

4.2.2.9.2. Functions

`bool async_context_threadsafe_background_init (async_context_threadsafe_background_t *self, async_context_threadsafe_background_config_t *config)`

Initialize an async_context_threadsafe_background instance using the specified configuration.

`async_context_threadsafe_background_config_t async_context_threadsafe_background_default_config (void)`

Return a copy of the default configuration object used by async_context_threadsafe_background_init_with_defaults()

`static bool async_context_threadsafe_background_init_with_defaults (async_context_threadsafe_background_t *self)`

Initialize an async_context_threadsafe_background instance with default values.

4.2.2.9.3. Function Documentation

async_context_threadsafe_background_default_config

`async_context_threadsafe_background_config_t async_context_threadsafe_background_default_config (void)`

Return a copy of the default configuration object used by async_context_threadsafe_background_init_with_defaults()

The caller can then modify just the settings it cares about, and call async_context_threadsafe_background_init()

Returns

the default configuration object

async_context_threadsafe_background_init

`bool async_context_threadsafe_background_init (async_context_threadsafe_background_t * self, async_context_threadsafe_background_config_t * config)`

Initialize an async_context_threadsafe_background instance using the specified configuration.

If this method succeeds (returns true), then the async_context is available for use and can be de-initialized by calling async_context_deinit().

Parameters

- `self` a pointer to async_context_threadsafe_background structure to initialize
- `config` the configuration object specifying characteristics for the async_context

Returns

true if initialization is successful, false otherwise

async_context_threadsafe_background_init_with_defaults

`static bool async_context_threadsafe_background_init_with_defaults (async_context_threadsafe_background_t * self) [inline], [static]`

Initialize an async_context_threadsafe_background instance with default values.

If this method succeeds (returns true), then the async_context is available for use and can be de-initialized by calling async_context_deinit().

Parameters

- `self` a pointer to async_context_threadsafe_background structure to initialize

Returns

true if initialization is successful, false otherwise

4.2.3. pico_bootsel_via_double_reset

Optional support to make fast double reset of the system enter BOOTSEL mode.

4.2.3.1. Detailed Description

When the 'pico_bootsel_via_double_reset' library is linked, a function is injected before main() which will detect when the system has been reset twice in quick succession, and enter the USB ROM bootloader (BOOTSEL mode) when this happens. This allows a double tap of a reset button on a development board to be used to enter the ROM bootloader, provided this library is always linked.

4.2.4. pico_flash

High level flash API.

4.2.4.1. Detailed Description

Flash cannot be erased or written to when in XIP mode. However the system cannot directly access memory in the flash address space when not in XIP mode.

It is therefore critical that no code or data is being read from flash while flash is been written or erased.

If only one core is being used, then the problem is simple - just disable interrupts; however if code is running on the other core, then it has to be asked, nicely, to avoid flash for a bit. This is hard to do if you don't have complete control of the code running on that core at all times.

This library provides a flash_safe_execute method which calls a function back having successfully gotten into a state where interrupts are disabled, and the other core is not executing or reading from flash.

How it does this is dependent on the supported environment (Free RTOS SMP or pico_multicore). Additionally the user can provide their own mechanism by providing a strong definition of get_flash_safety_helper().

Using the default settings, flash_safe_execute will only call the callback function if the state is safe otherwise returning an error (or an assert depending on PICO_FLASH_ASSERT_ON_UNSAFE).

There are conditions where safety would not be guaranteed:

1. FreeRTOS smp with `configNUM_CORES=1` - FreeRTOS still uses pico_multicore in this case, so flash_safe_execute cannot know what the other core is doing, and there is no way to force code execution between a FreeRTOS core and a non FreeRTOS core.
2. FreeRTOS non SMP with pico_multicore - Again, there is no way to force code execution between a FreeRTOS core and a non FreeRTOS core.
3. pico_multicore without flash_safe_execute_core_init() having been called on the other core - The flash_safe_execute method does not know if code is executing on the other core, so it has to assume it is. Either way, it is not able to intervene if flash_safe_execute_core_init() has not been called on the other core.

Fortunately, all is not lost in this situation, you may:

- Set PICO_FLASH_ASSUME_CORE0_SAFE=1 to explicitly say that core 0 is never using flash.
- Set PICO_FLASH_ASSUME_CORE1_SAFE=1 to explicitly say that core 1 is never using flash.

4.2.4.2. Functions

`bool flash_safe_execute_core_init (void)`

Initialize a core such that the other core can lock it out during flash_safe_execute.

`bool flash_safe_execute_core_deinit (void)`

De-initialize work done by flash_safe_execute_core_init.

`int flash_safe_execute (void(*func)(void *), void *param, uint32_t enter_exit_timeout_ms)`

Execute a function with IRQs disabled and with the other core also not executing/reading flash.

`flash_safety_helper_t * get_flash_safety_helper (void)`

Internal method to return the flash safety helper implementation.

4.2.4.3. Function Documentation

4.2.4.3.1. flash_safe_execute

`int flash_safe_execute (void(*)(void *) func, void * param, uint32_t enter_exit_timeout_ms)`

Execute a function with IRQs disabled and with the other core also not executing/reading flash.

Parameters

`func`	the function to call
`param`	the parameter to pass to the function
`enter_exit_timeout_ms`	the timeout for each of the enter/exit phases when coordinating with the other core

Returns

PICO_OK on success (the function will have been called). PICO_ERROR_TIMEOUT on timeout (the function may have been called). PICO_ERROR_NOT_PERMITTED if safe execution is not possible (the function will not have been called). PICO_ERROR_INSUFFICIENT_RESOURCES if the method fails due to dynamic resource exhaustion (the function will not have been called)

NOTE

> if PICO_FLASH_ASSERT_ON_UNSAFE is 1, this function will assert in debug mode vs returning PICO_ERROR_NOT_PERMITTED

4.2.4.3.2. flash_safe_execute_core_deinit

`bool flash_safe_execute_core_deinit (void)`

De-initialize work done by flash_safe_execute_core_init.

Returns

true on success

4.2.4.3.3. flash_safe_execute_core_init

`bool flash_safe_execute_core_init (void)`

Initialize a core such that the other core can lock it out during flash_safe_execute.

NOTE

> This is not necessary for FreeRTOS SMP, but should be used when launching via multicore_launch_core1

Returns

true on success; there is no need to call flash_safe_execute_core_deinit() on failure.

4.2.4.3.4. get_flash_safety_helper

`flash_safety_helper_t * get_flash_safety_helper (void)`

Internal method to return the flash safety helper implementation.

Advanced users can provide their own implementation of this function to perform different inter-core coordination before disabling XIP mode.

Returns

the flash_safety_helper_t

4.2.5. pico_i2c_slave

Functions providing an interrupt driven I2C slave interface.

4.2.5.1. Detailed Description

This I2C slave helper library configures slave mode and hooks the relevant I2C IRQ so that a user supplied handler is called with enumerated I2C events.

An example application `slave_mem_i2c`, which makes use of this library, can be found in pico_examples.

4.2.5.2. Typedefs

`typedef enum i2c_slave_event_t i2c_slave_event_t`

I2C slave event types.

`typedef void(* i2c_slave_handler_t)(i2c_inst_t *i2c, i2c_slave_event_t event)`

I2C slave event handler.

4.2.5.3. Enumerations

`enum i2c_slave_event_t { I2C_SLAVE_RECEIVE, I2C_SLAVE_REQUEST, I2C_SLAVE_FINISH }`

I2C slave event types.

4.2.5.4. Functions

`void i2c_slave_init (i2c_inst_t *i2c, uint8_t address, i2c_slave_handler_t handler)`

Configure an I2C instance for slave mode.

`void i2c_slave_deinit (i2c_inst_t *i2c)`

Restore an I2C instance to master mode.

4.2.5.5. Typedef Documentation

4.2.5.5.1. i2c_slave_event_t

```
typedef enum i2c_slave_event_t i2c_slave_event_t
```

I2C slave event types.

4.2.5.5.2. i2c_slave_handler_t

```
typedef void(* i2c_slave_handler_t) (i2c_inst_t *i2c, i2c_slave_event_t event)
```

I2C slave event handler.

The event handler will run from the I2C ISR, so it should return quickly (under 25 us at 400 kb/s). Avoid blocking inside the handler and split large data transfers across multiple calls for best results. When sending data to master, up to i2c_get_write_available() bytes can be written without blocking. When receiving data from master, up to i2c_get_read_available() bytes can be read without blocking.

Parameters

`i2c`	Either i2c0 or i2c1
`event`	Event type.

4.2.5.6. Enumeration Type Documentation

4.2.5.6.1. i2c_slave_event_t

`enum i2c_slave_event_t`

I2C slave event types.

Table 31. Enumerator

I2C_SLAVE_RECEIVE	Data from master is available for reading. Slave must read from Rx FIFO.
I2C_SLAVE_REQUEST	Master is requesting data. Slave must write into Tx FIFO.
I2C_SLAVE_FINISH	Master has sent a Stop or Restart signal. Slave may prepare for the next transfer.

4.2.5.7. Function Documentation

4.2.5.7.1. i2c_slave_deinit

`void i2c_slave_deinit (i2c_inst_t * i2c)`

Restore an I2C instance to master mode.

Parameters

`i2c` Either i2c0 or i2c1

4.2.5.7.2. i2c_slave_init

`void i2c_slave_init (i2c_inst_t * i2c, uint8_t address, i2c_slave_handler_t handler)`

Configure an I2C instance for slave mode.

Parameters

`i2c` I2C instance.

`address` 7-bit slave address.

`handler` Callback for events from I2C master. It will run from the I2C ISR, on the CPU core where the slave was initialised.

4.2.6. pico_multicore

Adds support for running code on, and interacting with the second processor core (core 1).

4.2.6.1. Detailed Description

Example

```c
#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!");
    }

}
```

4.2.6.2. Modules

fifo

Functions for the inter-core FIFOs.

doorbell

Functions related to doorbells which a core can use to raise IRQs on itself or the other core.

lockout

Functions to enable one core to force the other core to pause execution in a known state.

4.2.6.3. Macros

- `#define SIO_FIFO_IRQ_NUM(core)`

4.2.6.4. Functions

`void multicore_reset_core1 (void)`

Reset core 1.

`void multicore_launch_core1 (void(*entry)(void))`

Run code on core 1.

`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.

`void multicore_launch_core1_raw (void(*entry)(void), uint32_t *sp, uint32_t vector_table)`

Launch code on core 1 with no stack protection.

4.2.6.5. Macro Definition Documentation

4.2.6.5.1. SIO_FIFO_IRQ_NUM

`#define SIO_FIFO_IRQ_NUM(core)`

Returns the irq_num_t for the FIFO IRQ on the given core.

On RP2040 each core has a different IRQ number: `SIO_IRQ_PROC0` and `SIO_IRQ_PROC1`. On RP2350 both cores share the same irq number (`SIO_IRQ_PROC`) just with a different SIO interrupt output routed to that IRQ input on each core.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.2.6.6. Function Documentation

4.2.6.6.1. multicore_launch_core1

`void multicore_launch_core1 (void(*)(void) entry)`

Run code on core 1.

Wake up (a previously reset) core 1 and enter the given function on core 1 using the default core 1 stack (below core 0 stack).

core 1 must previously have been reset either as a result of a system reset or by calling multicore_reset_core1

core 1 will use the same vector table as core 0

Parameters

`entry`	Function entry point

See also

multicore_reset_core1

4.2.6.6.2. multicore_launch_core1_raw

`void multicore_launch_core1_raw (void(*)(void) entry, uint32_t * sp, uint32_t vector_table)`

Launch code on core 1 with no stack protection.

Wake up (a previously reset) core 1 and start it executing with a specific entry point, stack pointer and vector table.

This is a low level function that does not provide a stack guard even if USE_STACK_GUARDS is defined

core 1 must previously have been reset either as a result of a system reset or by calling multicore_reset_core1

Parameters

`entry`	Function entry point
`sp`	Pointer to the top of the core 1 stack
`vector_table`	address of the vector table to use for core 1

See also

multicore_reset_core1

4.2.6.6.3. multicore_launch_core1_with_stack

`void multicore_launch_core1_with_stack (void(*)(void) entry, uint32_t * stack_bottom, size_t stack_size_bytes)`

Launch code on core 1 with stack.

Wake up (a previously reset) core 1 and enter the given function on core 1 using the passed stack for core 1

core 1 must previously have been reset either as a result of a system reset or by calling multicore_reset_core1

core 1 will use the same vector table as core 0

Parameters

`entry`	Function entry point
`stack_bottom`	The bottom (lowest address) of the stack
`stack_size_bytes`	The size of the stack in bytes (must be a multiple of 4)

See also

multicore_reset_core1

4.2.6.6.4. multicore_reset_core1

`void multicore_reset_core1 (void)`

Reset core 1.

This function can be used to reset core 1 into its initial state (ready for launching code against via multicore_launch_core1 and similar methods)

NOTE

> this function should only be called from core 0

4.2.6.7. fifo

Functions for the inter-core FIFOs.

4.2.6.7.1. Detailed Description

RP-series microcontrollers contains two FIFOs for passing data, messages or ordered events between the two cores. Each FIFO is 32 bits wide, and 8 entries deep on the RP2040, and 4 entries deep on the RP2350. 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.

> **NOTE**
>
> The inter-core FIFOs are a very precious resource and are frequently used for SDK functionality (e.g. during core 1 launch or by the lockout functions). Additionally they are often required for the exclusive use of an RTOS (e.g. FreeRTOS SMP). For these reasons it is suggested that you do not use the FIFO for your own purposes unless none of the above concerns apply; the majority of cases for transferring data between cores can be eqaully well handled by using a queue

4.2.6.7.2. Functions

`static bool multicore_fifo_rvalid (void)`

Check the read FIFO to see if there is data available (sent by the other core)

`static bool multicore_fifo_wready (void)`

Check the write FIFO to see if it has space for more data.

`void multicore_fifo_push_blocking (uint32_t data)`

Push data on to the write FIFO (data to the other core).

`static void multicore_fifo_push_blocking_inline (uint32_t data)`

Push data on to the write FIFO (data to the other core).

`bool multicore_fifo_push_timeout_us (uint32_t data, uint64_t timeout_us)`

Push data on to the write FIFO (data to the other core) with timeout.

`uint32_t multicore_fifo_pop_blocking (void)`

Pop data from the read FIFO (data from the other core).

`static uint32_t multicore_fifo_pop_blocking_inline (void)`

Pop data from the read FIFO (data from the other core).

`bool multicore_fifo_pop_timeout_us (uint64_t timeout_us, uint32_t *out)`

Pop data from the read FIFO (data from the other core) with timeout.

`static void multicore_fifo_drain (void)`

Discard any data in the read FIFO.

`static void multicore_fifo_clear_irq (void)`

Clear FIFO interrupt.

`static uint32_t multicore_fifo_get_status (void)`

Get FIFO statuses.

4.2.6.7.3. Function Documentation

multicore_fifo_clear_irq

`static void multicore_fifo_clear_irq (void) [inline], [static]`

Clear FIFO interrupt.

Note that this only clears an interrupt that was caused by the ROE or WOF flags. To clear the VLD flag you need to use one of the 'pop' or 'drain' functions.

See the note in the fifo section for considerations regarding use of the inter-core FIFOs

See also

multicore_fifo_get_status

multicore_fifo_drain

```
static void multicore_fifo_drain (void) [inline], [static]
```

Discard any data in the read FIFO.

See the note in the fifo section for considerations regarding use of the inter-core FIFOs

multicore_fifo_get_status

```
static uint32_t multicore_fifo_get_status (void) [inline], [static]
```

Get FIFO statuses.

Returns

The statuses as a bitfield

Bit	Description
3	Sticky flag indicating the RX FIFO was read when empty (ROE). This read was ignored by the FIFO.
2	Sticky flag indicating the TX FIFO was written when full (WOF). 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)

See the note in the fifo section for considerations regarding use of the inter-core FIFOs

multicore_fifo_pop_blocking

```
uint32_t multicore_fifo_pop_blocking (void)
```

Pop data from the read FIFO (data from the other core).

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.

See the note in the fifo section for considerations regarding use of the inter-core FIFOs

Returns

32 bit data from the read FIFO.

multicore_fifo_pop_blocking_inline

```
static uint32_t multicore_fifo_pop_blocking_inline (void) [inline], [static]
```

Pop data from the read FIFO (data from the other core).

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.

See the note in the fifo section for considerations regarding use of the inter-core FIFOs

Returns

32 bit data from the read FIFO.

multicore_fifo_pop_timeout_us

```
bool multicore_fifo_pop_timeout_us (uint64_t timeout_us, uint32_t * out)
```

Pop data from the read FIFO (data from the other core) with timeout.

This function will block until there is data ready to be read or the timeout is reached

See the note in the fifo section for considerations regarding use of the inter-core FIFOs

Parameters

`timeout_us`	the timeout in microseconds
`out`	the location to store the popped data if available

Returns

true if the data was popped and a value copied into `out`, false if the timeout occurred before data could be popped

multicore_fifo_push_blocking

```
void multicore_fifo_push_blocking (uint32_t data)
```

Push data on to the write FIFO (data to the other core).

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.

See the note in the fifo section for considerations regarding use of the inter-core FIFOs

Parameters

`data`	A 32 bit value to push on to the FIFO

multicore_fifo_push_blocking_inline

```
static void multicore_fifo_push_blocking_inline (uint32_t data) [inline], [static]
```

Push data on to the write FIFO (data to the other core).

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.

See the note in the fifo section for considerations regarding use of the inter-core FIFOs

Parameters

`data`	A 32 bit value to push on to the FIFO

multicore_fifo_push_timeout_us

```
bool multicore_fifo_push_timeout_us (uint32_t data, uint64_t timeout_us)
```

Push data on to the write FIFO (data to the other core) with timeout.

This function will block until there is space for the data to be sent or the timeout is reached

Parameters

`data`	A 32 bit value to push on to the FIFO
`timeout_us`	the timeout in microseconds

Returns

true if the data was pushed, false if the timeout occurred before data could be pushed

multicore_fifo_rvalid

```
static bool multicore_fifo_rvalid (void) [inline], [static]
```

Check the read FIFO to see if there is data available (sent by the other core)

See the note in the fifo section for considerations regarding use of the inter-core FIFOs

Returns

true if the FIFO has data in it, false otherwise

multicore_fifo_wready

`static bool multicore_fifo_wready (void) [inline], [static]`

Check the write FIFO to see if it has space for more data.

See the note in the fifo section for considerations regarding use of the inter-core FIFOs

Returns

true if the FIFO has room for more data, false otherwise

4.2.6.8. doorbell

Functions related to doorbells which a core can use to raise IRQs on itself or the other core.

4.2.6.8.1. Macros

- `#define DOORBELL_IRQ_NUM(doorbell_num)`

4.2.6.8.2. Functions

`void multicore_doorbell_claim (uint doorbell_num, uint core_mask)` RP2350

Cooperatively claim the use of this hardware alarm_num.

`int multicore_doorbell_claim_unused (uint core_mask, bool required)` RP2350

Cooperatively claim the use of this hardware alarm_num.

`void multicore_doorbell_unclaim (uint doorbell_num, uint core_mask)` RP2350

Cooperatively release the claim on use of this hardware alarm_num.

`static void multicore_doorbell_set_other_core (uint doorbell_num)` RP2350

Activate the given doorbell on the other core.

`static void multicore_doorbell_clear_other_core (uint doorbell_num)` RP2350

Deactivate the given doorbell on the other core.

`static void multicore_doorbell_set_current_core (uint doorbell_num)` RP2350

Activate the given doorbell on this core.

`static void multicore_doorbell_clear_current_core (uint doorbell_num)` RP2350

Deactivate the given doorbell on this core.

`static bool multicore_doorbell_is_set_current_core (uint doorbell_num)` RP2350

Determine if the given doorbell is active on the other core.

`static bool multicore_doorbell_is_set_other_core (uint doorbell_num)` RP2350

Determine if the given doorbell is active on the this core.

4.2.6.8.3. Macro Definition Documentation

DOORBELL_IRQ_NUM RP2350

`#define DOORBELL_IRQ_NUM(doorbell_num)`

Returns the irq_num_t for processor interrupts for the given doorbell number.

Note this macro is intended to resolve at compile time, and does no parameter checking

4.2.6.8.4. Function Documentation

multicore_doorbell_claim RP2350

`void multicore_doorbell_claim (uint doorbell_num, uint core_mask)`

Cooperatively claim the use of this hardware alarm_num.

This method hard asserts if the hardware alarm is currently claimed.

Parameters

`doorbell_num`	the doorbell number to claim
`core_mask`	0b01: core 0, 0b10: core 1, 0b11 both core 0 and core 1

See also

hardware_claiming

multicore_doorbell_claim_unused RP2350

`int multicore_doorbell_claim_unused (uint core_mask, bool required)`

Cooperatively claim the use of this hardware alarm_num.

This method attempts to claim an unused hardware alarm

Parameters

`core_mask`	0b01: core 0, 0b10: core 1, 0b11 both core 0 and core 1
`required`	if true the function will panic if none are available

Returns

the doorbell number claimed or -1 if required was false, and none are available

See also

hardware_claiming

multicore_doorbell_clear_current_core RP2350

`static void multicore_doorbell_clear_current_core (uint doorbell_num) [inline], [static]`

Deactivate the given doorbell on this core.

Parameters

`doorbell_num`	the doorbell number

multicore_doorbell_clear_other_core RP2350

`static void multicore_doorbell_clear_other_core (uint doorbell_num) [inline], [static]`

Deactivate the given doorbell on the other core.

Parameters

`doorbell_num`	the doorbell number

multicore_doorbell_is_set_current_core RP2350

`static bool multicore_doorbell_is_set_current_core (uint doorbell_num) [inline], [static]`

Determine if the given doorbell is active on the other core.

Parameters

`doorbell_num`	the doorbell number

multicore_doorbell_is_set_other_core RP2350

`static bool multicore_doorbell_is_set_other_core (uint doorbell_num) [inline], [static]` Determine if the given doorbell is active on the this core.

Parameters

`doorbell_num`	the doorbell number

multicore_doorbell_set_current_core RP2350

```
static void multicore_doorbell_set_current_core (uint doorbell_num) [inline], [static]
```

Activate the given doorbell on this core.

Parameters

`doorbell_num`	the doorbell number

multicore_doorbell_set_other_core RP2350

```
static void multicore_doorbell_set_other_core (uint doorbell_num) [inline], [static]
```

Activate the given doorbell on the other core.

Parameters

`doorbell_num`	the doorbell number

multicore_doorbell_unclaim RP2350

```
void multicore_doorbell_unclaim (uint doorbell_num, uint core_mask)
```

Cooperatively release the claim on use of this hardware alarm_num.

Parameters

`doorbell_num`	the doorbell number to unclaim
`core_mask`	0b01: core 0, 0b10: core 1, 0b11 both core 0 and core 1

See also

hardware_claiming

4.2.6.9. lockout

Functions to enable one core to force the other core to pause execution in a known state.

4.2.6.9.1. Detailed Description

Sometimes it is useful to enter a critical section on both cores at once. On a single core system a critical section can trivially be entered by disabling interrupts, however on a multi-core system that is not sufficient, and unless the other core is polling in some way, then it will need to be interrupted in order to cooperatively enter a blocked state.

These "lockout" functions use the inter core FIFOs to cause an interrupt on one core from the other, and manage waiting for the other core to enter the "locked out" state.

The usage is that the "victim" core … i.e the core that can be "locked out" by the other core calls multicore_lockout_victim_init to hook the FIFO interrupt. Note that either or both cores may do this.

NOTE

When "locked out" the victim core is paused (it is actually executing a tight loop with code in RAM) and has interrupts disabled. This makes the lockout functions suitable for use by code that wants to write to flash (at which point no code may be executing from flash)

The core which wishes to lockout the other core calls multicore_lockout_start_blocking or multicore_lockout_start_timeout_us to interrupt the other "victim" core and wait for it to be in a "locked out" state. Once the lockout is no longer needed it calls multicore_lockout_end_blocking or multicore_lockout_end_timeout_us to release the lockout and wait for confirmation.

NOTE

Because multicore lockout uses the intercore FIFOs, the FIFOs **cannot** be used for any other purpose

4.2.6.9.2. Functions

`void multicore_lockout_victim_init (void)`

Initialize the current core such that it can be a "victim" of lockout (i.e. forced to pause in a known state by the other core)

`void multicore_lockout_victim_deinit (void)`

Stop the current core being able to be a "victim" of lockout (i.e. forced to pause in a known state by the other core)

`bool multicore_lockout_victim_is_initialized (uint core_num)`

Determine if multicore_lockout_victim_init() has been called on the specified core.

`void multicore_lockout_start_blocking (void)`

Request the other core to pause in a known state and wait for it to do so.

`bool multicore_lockout_start_timeout_us (uint64_t timeout_us)`

Request the other core to pause in a known state and wait up to a time limit for it to do so.

`void multicore_lockout_end_blocking (void)`

Release the other core from a locked out state amd wait for it to acknowledge.

`bool multicore_lockout_end_timeout_us (uint64_t timeout_us)`

Release the other core from a locked out state amd wait up to a time limit for it to acknowledge.

4.2.6.9.3. Function Documentation

multicore_lockout_end_blocking

`void multicore_lockout_end_blocking (void)`

Release the other core from a locked out state amd wait for it to acknowledge.

NOTE

The other core must previously have been "locked out" by calling a `multicore_lockout_start_` function from this core

multicore_lockout_end_timeout_us

`bool multicore_lockout_end_timeout_us (uint64_t timeout_us)`

Release the other core from a locked out state amd wait up to a time limit for it to acknowledge.

The other core must previously have been "locked out" by calling a `multicore_lockout_start_` function from this core

> **NOTE**
>
> be very careful using small timeout values, as a timeout here will leave the "lockout" functionality in a bad state. It is probably preferable to use multicore_lockout_end_blocking anyway as if you have already waited for the victim core to enter the lockout state, then the victim core will be ready to exit the lockout state very quickly.

Parameters

`timeout_us`	the timeout in microseconds

Returns

true if the other core successfully exited locked out state within the timeout, false otherwise

multicore_lockout_start_blocking

```
void multicore_lockout_start_blocking (void)
```

Request the other core to pause in a known state and wait for it to do so.

The other (victim) core must have previously executed multicore_lockout_victim_init()

> **NOTE**
>
> multicore_lockout_start_ functions are not nestable, and must be paired with a call to a corresponding multicore_lockout_end_blocking

multicore_lockout_start_timeout_us

```
bool multicore_lockout_start_timeout_us (uint64_t timeout_us)
```

Request the other core to pause in a known state and wait up to a time limit for it to do so.

The other core must have previously executed multicore_lockout_victim_init()

> **NOTE**
>
> multicore_lockout_start_ functions are not nestable, and must be paired with a call to a corresponding multicore_lockout_end_blocking

Parameters

`timeout_us`	the timeout in microseconds

Returns

true if the other core entered the locked out state within the timeout, false otherwise

multicore_lockout_victim_deinit

```
void multicore_lockout_victim_deinit (void)
```

Stop the current core being able to be a "victim" of lockout (i.e. forced to pause in a known state by the other core)

This code unhooks the intercore FIFO IRQ, and the FIFO may be used for any other purpose after this.

multicore_lockout_victim_init

```
void multicore_lockout_victim_init (void)
```

Initialize the current core such that it can be a "victim" of lockout (i.e. forced to pause in a known state by the other core)

This code hooks the intercore FIFO IRQ, and the FIFO may not be used for any other purpose after this.

multicore_lockout_victim_is_initialized

```
bool multicore_lockout_victim_is_initialized (uint core_num)
```

Determine if multicore_lockout_victim_init() has been called on the specified core.

> **NOTE**
>
> this state persists even if the core is subsequently reset; therefore you are advised to always call multicore_lockout_victim_init() again after resetting a core, which had previously been initialized.

Parameters

`core_num` the core number (0 or 1)

Returns

true if multicore_lockout_victim_init() has been called on the specified core, false otherwise.

4.2.7. pico_rand

Random Number Generator API.

4.2.7.1. Detailed Description

This module generates random numbers at runtime utilizing a number of possible entropy sources and uses those sources to modify the state of a 128-bit 'Pseudo Random Number Generator' implemented in software.

The random numbers (32 to 128 bit) to be supplied are read from the PRNG which is used to help provide a large number space.

The following (multiple) sources of entropy are available (of varying quality), each enabled by a #define:

- The Ring Oscillator (ROSC) (PICO_RAND_ENTROPY_SRC_ROSC == 1): PICO_RAND_ROSC_BIT_SAMPLE_COUNT bits are gathered from the ring oscillator "random bit" and mixed in each time. This should not be used if the ROSC is off, or the processor is running from the ROSC.

  > **NOTE**
  >
  > the maximum throughput of ROSC bit sampling is controlled by PICO_RAND_MIN_ROSC_BIT_SAMPLE_TIME_US which defaults to 10us, i.e. 100,000 bits per second.

- Time (PICO_RAND_ENTROPY_SRC_TIME == 1): The 64-bit microsecond timer is mixed in each time.
- Bus Performance Counter (PICO_RAND_ENTROPY_SRC_BUS_PERF_COUNTER == 1): One of the bus fabric's performance counters is mixed in each time.

> **NOTE**
>
> All entropy sources are hashed before application to the PRNG state machine.

The *first* time a random number is requested, the 128-bit PRNG state must be seeded. Multiple entropy sources are also available for the seeding operation:

- The Ring Oscillator (ROSC) (PICO_RAND_SEED_ENTROPY_SRC_ROSC == 1): 64 bits are gathered from the ring oscillator "random bit" and mixed into the seed.
- Time (PICO_RAND_SEED_ENTROPY_SRC_TIME == 1): The 64-bit microsecond timer is mixed into the seed.
- Board Identifier (PICO_RAND_SEED_ENTROPY_SRC_BOARD_ID == 1): The board id via pico_get_unique_board_id is mixed into the seed.
- RAM hash (PICO_RAND_SEED_ENTROPY_SRC_RAM_HASH (PICO_RAND_SEED_ENTROPY_SRC_RAM_HASH): The hashed contents of a subset of RAM are mixed in. Initial RAM contents are undefined on power up, so provide a

reasonable source of entropy. By default the last 1K of RAM (which usually contains the core 0 stack) is hashed, which may also provide for differences after each warm reset.

With default settings, the seed generation takes approximately 1 millisecond while subsequent random numbers generally take between 10 and 20 microseconds to generate.

pico_rand methods may be safely called from either core or from an IRQ, but be careful in the latter case as the calls may block for a number of microseconds waiting on more entropy.

4.2.7.2. Functions

`void get_rand_128 (rng_128_t *rand128)`

Get 128-bit random number.

`uint64_t get_rand_64 (void)`

Get 64-bit random number.

`uint32_t get_rand_32 (void)`

Get 32-bit random number.

4.2.7.3. Function Documentation

4.2.7.3.1. get_rand_128

`void get_rand_128 (rng_128_t * rand128)`

Get 128-bit random number.

This method may be safely called from either core or from an IRQ, but be careful in the latter case as the call may block for a number of microseconds waiting on more entropy.

Parameters

`rand128`	Pointer to storage to accept a 128-bit random number

4.2.7.3.2. get_rand_32

`uint32_t get_rand_32 (void)`

Get 32-bit random number.

This method may be safely called from either core or from an IRQ, but be careful in the latter case as the call may block for a number of microseconds waiting on more entropy.

Returns

32-bit random number

4.2.7.3.3. get_rand_64

`uint64_t get_rand_64 (void)`

Get 64-bit random number.

This method may be safely called from either core or from an IRQ, but be careful in the latter case as the call may block for a number of microseconds waiting on more entropy.

Returns

64-bit random number

4.2.8. pico_sha256 RP2350

SHA-256 Hardware Accelerated implementation.

4.2.8.1. Detailed Description

RP2350 is equipped with a hardware accelerated implementation of the SHA-256 hash algorithm. This should be much quicker than performing a SHA-256 checksum in software.

```
pico_sha256_state_t state;
if (pico_sha256_try_start(&state, SHA256_BIG_ENDIAN, true) == PICO_OK) {
    sha256_result_t result;
    pico_sha256_update(&state, some_data, sizeof(some_data));
    pico_sha256_update(&state, some_more_data, sizeof(some_more_data));
    pico_sha256_finish(&state, &result);
    for (int i = 0; i < SHA256_RESULT_BYTES; i++) {
        printf("%02x", result.bytes[i]);
    }
}
```

Example

```
#include <stdio.h>
#include <string.h>
// Include sys/types.h before inttypes.h to work around issue with
// certain versions of GCC and newlib which causes omission of PRIu64
#include <sys/types.h>
#include <inttypes.h>
#include <stdlib.h>

#include "pico/stdlib.h"
#include "pico/sha256.h"

// This was generated by cmake from sample.txt.inc
#include "sample.txt.inc"

static void sha_example() {
    printf("Text: %d bytes\n", sizeof(sample_txt) - 1);
    for(int i = 0; i < sizeof(sample_txt) - 1; i++) {
        if (i > 0 && i % 128 == 0) printf("\n");
        putchar(sample_txt[i]);
    }
    printf("\n");

    // Allocate a state object and start the calculation
    pico_sha256_state_t state;
    int rc = pico_sha256_start_blocking(&state, SHA256_BIG_ENDIAN, true); // using some DMA system resources
    hard_assert(rc == PICO_OK);
    pico_sha256_update_blocking(&state, (const uint8_t*)sample_txt, sizeof(sample_txt) - 1);

    // Get the result of the sha256 calculation
    sha256_result_t result;
    pico_sha256_finish(&state, &result);

    // print resulting sha256 result
    printf("Result:\n");
    for(int i = 0; i < SHA256_RESULT_BYTES; i++) {
```

```c
        printf("%02x ", result.bytes[i]);
        if ((i+1) % 16 == 0) printf("\n");
    }

    // check it's what we expect from "sha256sum sample.txt"
    const uint8_t sha_expected[SHA256_RESULT_BYTES] = {
        0x2d, 0x8c, 0x2f, 0x6d, 0x97, 0x8c, 0xa2, 0x17, 0x12, 0xb5, 0xf6, 0xde, 0x36, 0xc9, 0xd3, 0x1f,
        0xa8, 0xe9, 0x6a, 0x4f, 0xa5, 0xd8, 0xff, 0x8b, 0x01, 0x88, 0xdf, 0xb9, 0xe7, 0xc1, 0x71, 0xbb
    };
    hard_assert(memcmp(sha_expected, &result, SHA256_RESULT_BYTES) == 0);
}


#define BUFFER_SIZE 10000

// A performance test with a large amount of data
static void nist_test(bool use_dma) {
    // nist 3
    uint8_t *buffer = malloc(BUFFER_SIZE);
    memset(buffer, 0x61, BUFFER_SIZE);
    const uint8_t nist_3_expected[] = { \
        0xcd, 0xc7, 0x6e, 0x5c, 0x99, 0x14, 0xfb, 0x92, 0x81, 0xa1, 0xc7, 0xe2, 0x84, 0xd7, 0x3e, 0x67,
        0xf1, 0x80, 0x9a, 0x48, 0xa4, 0x97, 0x20, 0x0e, 0x04, 0x6d, 0x39, 0xcc, 0xc7, 0x11, 0x2c, 0xd0 };

    uint64_t start = time_us_64();
    pico_sha256_state_t state;
    int rc = pico_sha256_start_blocking(&state, SHA256_BIG_ENDIAN, use_dma); // call start once
    hard_assert(rc == PICO_OK);
    for(int i = 0; i < 1000000; i += BUFFER_SIZE) {
        pico_sha256_update_blocking(&state, buffer, BUFFER_SIZE); // call update as many times as required
    }
    sha256_result_t result;
    pico_sha256_finish(&state, &result); // Call finish when done to get the result

    // Display the time taken
    uint64_t pico_time = time_us_64() - start;
    printf("Time for sha256 of 1M bytes %s DMA %"PRIu64"ms\n", use_dma ? "with" : "without", pico_time / 1000);
    hard_assert(memcmp(nist_3_expected, result.bytes, SHA256_RESULT_BYTES) == 0);
}

int main() {
    stdio_init_all();

    sha_example();

    // performance test with and without DMA
    nist_test(false);
    nist_test(true);

    printf("Success\n");
}
```

4.2.8.2. Typedefs

`typedef struct pico_sha256_state pico_sha256_state_t`

SHA-256 state used by the API.

4.2.8.3. Functions

`void pico_sha256_cleanup (pico_sha256_state_t *state)`

Release the internal lock on the SHA-256 hardware.

`int pico_sha256_try_start (pico_sha256_state_t *state, enum sha256_endianness endianness, bool use_dma)`

Start a SHA-256 calculation returning immediately with an error if the SHA-256 hardware is not available.

`int pico_sha256_start_blocking_until (pico_sha256_state_t *state, enum sha256_endianness endianness, bool use_dma, absolute_time_t until)`

Start a SHA-256 calculation waiting for a defined period for the SHA-256 hardware to be available.

`static int pico_sha256_start_blocking (pico_sha256_state_t *state, enum sha256_endianness endianness, bool use_dma)`

Start a SHA-256 calculation, blocking forever waiting until the SHA-256 hardware is available.

`void pico_sha256_update (pico_sha256_state_t *state, const uint8_t *data, size_t data_size_bytes)`

Add byte data to be SHA-256 calculation.

`void pico_sha256_update_blocking (pico_sha256_state_t *state, const uint8_t *data, size_t data_size_bytes)`

Add byte data to be SHA-256 calculation.

`void pico_sha256_finish (pico_sha256_state_t *state, sha256_result_t *out)`

Finish the SHA-256 calculation and return the result.

4.2.8.4. Typedef Documentation

4.2.8.4.1. pico_sha256_state_t

`typedef struct pico_sha256_state pico_sha256_state_t`

SHA-256 state used by the API.

4.2.8.5. Function Documentation

4.2.8.5.1. pico_sha256_cleanup

`void pico_sha256_cleanup (pico_sha256_state_t * state)`

Release the internal lock on the SHA-256 hardware.

Release the internal lock on the SHA-256 hardware. Does nothing if the internal lock was not claimed.

Parameters

`state`	A pointer to a pico_sha256_state_t instance

4.2.8.5.2. pico_sha256_finish

`void pico_sha256_finish (pico_sha256_state_t * state, sha256_result_t * out)`

Finish the SHA-256 calculation and return the result.

Ends the SHA-256 calculation freeing the hardware for use by another caller. You must have called pico_sha256_try_start already.

Parameters

`state`	A pointer to a pico_sha256_state_t instance
`out`	The SHA-256 checksum

4.2.8.5.3. pico_sha256_start_blocking

`static int pico_sha256_start_blocking (pico_sha256_state_t * state, enum sha256_endianness endianness, bool use_dma) [inline], [static]`

Start a SHA-256 calculation, blocking forever waiting until the SHA-256 hardware is available.

Initialises the hardware and state ready to start a new SHA-256 calculation. Only one instance can be started at any time.

Parameters

`state`	A pointer to a pico_sha256_state_t instance
`endianness`	SHA256_BIG_ENDIAN or SHA256_LITTLE_ENDIAN for data in and data out
`use_dma`	Set to true to use DMA internally to copy data to hardware. This is quicker at the expense of hardware DMA resources.

Returns

Returns PICO_OK if the hardware was available for use and the sha256 calculation could be started, otherwise an error is returned

4.2.8.5.4. pico_sha256_start_blocking_until

`int pico_sha256_start_blocking_until (pico_sha256_state_t * state, enum sha256_endianness endianness, bool use_dma, absolute_time_t until)`

Start a SHA-256 calculation waiting for a defined period for the SHA-256 hardware to be available.

Initialises the hardware and state ready to start a new SHA-256 calculation. Only one instance can be started at any time.

Parameters

`state`	A pointer to a pico_sha256_state_t instance
`endianness`	SHA256_BIG_ENDIAN or SHA256_LITTLE_ENDIAN for data in and data out
`use_dma`	Set to true to use DMA internally to copy data to hardware. This is quicker at the expense of hardware DMA resources.
`until`	How long to wait for the SHA hardware to be available

Returns

Returns PICO_OK if the hardware was available for use and the sha256 calculation could be started in time, otherwise an error is returned

4.2.8.5.5. pico_sha256_try_start

`int pico_sha256_try_start (pico_sha256_state_t * state, enum sha256_endianness endianness, bool use_dma)`

Start a SHA-256 calculation returning immediately with an error if the SHA-256 hardware is not available.

Initialises the hardware and state ready to start a new SHA-256 calculation. Only one instance can be started at any time.

Parameters

`state`	A pointer to a pico_sha256_state_t instance
`endianness`	SHA256_BIG_ENDIAN or SHA256_LITTLE_ENDIAN for data in and data out
`use_dma`	Set to true to use DMA internally to copy data to hardware. This is quicker at the expense of hardware DMA resources.

Returns

Returns PICO_OK if the hardware was available for use and the sha256 calculation could be started, otherwise an error is returned

4.2.8.5.6. pico_sha256_update

```
void pico_sha256_update (pico_sha256_state_t * state, const uint8_t * data, size_t data_size_bytes)
```

Add byte data to be SHA-256 calculation.

Add byte data to be SHA-256 calculation You may call this as many times as required to add all the data needed. You must have called pico_sha256_try_start (or equivalent) already.

Parameters

`state`	A pointer to a pico_sha256_state_t instance
`data`	Pointer to the data to be added to the calculation
`data_size_bytes`	Amount of data to add

NOTE

> This function may return before the copy has completed in which case the data passed to the function must remain valid and unchanged until a further call to pico_sha256_update or pico_sha256_finish. If this is not done, corrupt data may be used for the SHA-256 calculation giving an unexpected result.

4.2.8.5.7. pico_sha256_update_blocking

```
void pico_sha256_update_blocking (pico_sha256_state_t * state, const uint8_t * data, size_t data_size_bytes)
```

Add byte data to be SHA-256 calculation.

Add byte data to be SHA-256 calculation You may call this as many times as required to add all the data needed. You must have called pico_sha256_try_start already.

Parameters

`state`	A pointer to a pico_sha256_state_t instance
`data`	Pointer to the data to be added to the calculation
`data_size_bytes`	Amount of data to add

> **NOTE**
>
> This function will only return when the data passed in is no longer required, so it can be freed or changed on return.

4.2.9. pico_stdlib

Aggregation of a core subset of Raspberry Pi Pico SDK libraries used by most executables along with some additional utility methods.

4.2.9.1. Detailed Description

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_divider
- hardware_gpio
- hardware_uart
- pico_runtime
- pico_platform
- pico_stdio
- pico_time
- 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

4.2.9.2. Functions

`void setup_default_uart (void)`

Set up the default UART and assign it to the default GPIOs.

4.2.9.3. Function Documentation

4.2.9.3.1. setup_default_uart

`void setup_default_uart (void)`

Set up the default UART and assign it to the default GPIOs.

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 initializes 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

4.2.10. pico_sync

Synchronization primitives and mutual exclusion.

4.2.10.1. Modules

critical_section

Critical Section API for short-lived mutual exclusion safe for IRQ and multi-core.

lock_core

base synchronization/lock primitive support.

mutex

Mutex API for non IRQ mutual exclusion between cores.

sem

Semaphore API for restricting access to a resource.

4.2.10.2. critical_section

Critical Section API for short-lived mutual exclusion safe for IRQ and multi-core.

4.2.10.2.1. Detailed Description

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 when a critical_section is owned, uses of the critical_section should be as short as possible.

4.2.10.2.2. Functions

`void critical_section_init (critical_section_t *crit_sec)`

Initialise a critical_section structure allowing the system to assign a spin lock number.

`void critical_section_init_with_lock_num (critical_section_t *crit_sec, uint lock_num)`

Initialise a critical_section structure assigning a specific spin lock number.

`static void critical_section_enter_blocking (critical_section_t *crit_sec)`

Enter a critical_section.

`static void critical_section_exit (critical_section_t *crit_sec)`

Release a critical_section.

`void critical_section_deinit (critical_section_t *crit_sec)`

De-Initialise a critical_section created by the critical_section_init method.

4.2.10.2.3. Function Documentation

critical_section_deinit

`void critical_section_deinit (critical_section_t * crit_sec)`

De-Initialise a critical_section created by the critical_section_init method.

This method is only used to free the associated spin lock allocated via the critical_section_init method (it should not be used to de-initialize a spin lock created via critical_section_init_with_lock_num). After this call, the critical section is invalid

Parameters

`crit_sec` Pointer to critical_section structure

critical_section_enter_blocking

```
static void critical_section_enter_blocking (critical_section_t * crit_sec) [inline], [static]
```

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

`crit_sec` Pointer to critical_section structure

critical_section_exit

```
static void critical_section_exit (critical_section_t * crit_sec) [inline], [static]
```

Release a critical_section.

Parameters

`crit_sec` Pointer to critical_section structure

critical_section_init

```
void critical_section_init (critical_section_t * crit_sec)
```

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 critical sections, however if you do so you *must* use critical_section_init_with_lock_num to ensure that the spin locks used are different.

Parameters

`crit_sec` Pointer to critical_section structure

critical_section_init_with_lock_num

```
void critical_section_init_with_lock_num (critical_section_t * crit_sec, uint lock_num)
```

Initialise a critical_section structure assigning a specific spin lock number.

Parameters

`crit_sec` Pointer to critical_section structure

`lock_num` the specific spin lock number to use

4.2.10.3. lock_core

base synchronization/lock primitive support.

4.2.10.3.1. Detailed Description

Most of the pico_sync locking primitives contain a lock_core_t structure member. This currently just holds a spin lock which is used only to protect the contents of the rest of the structure as part of implementing the synchronization primitive. As such, the spin_lock member of lock core is never still held on return from any function for the primitive.

critical_section is an exceptional case in that it does not have a lock_core_t and simply wraps a spin lock, providing methods to lock and unlock said spin lock.

lock_core based structures work by locking the spin lock, checking state, and then deciding whether they additionally need to block or notify when the spin lock is released. In the blocking case, they will wake up again in the future, and try the process again.

By default the SDK just uses the processors' events via SEV and WEV for notification and blocking as these are sufficient for cross core, and notification from interrupt handlers. However macros are defined in this file that abstract the wait and notify mechanisms to allow the SDK locking functions to effectively be used within an RTOS or other environment.

When implementing an RTOS, it is desirable for the SDK synchronization primitives that wait, to block the calling task (and immediately yield), and those that notify, to wake a blocked task which isn't on processor. At least the wait macro implementation needs to be atomic with the protecting spin_lock unlock from the callers point of view; i.e. the task should unlock the spin lock when it starts its wait. Such implementation is up to the RTOS integration, however the macros are defined such that such operations are always combined into a single call (so they can be performed atomically) even though the default implementation does not need this, as a WFE which starts following the corresponding SEV is not missed.

4.2.10.3.2. Macros

- `#define lock_owner_id_t int8_t`
- `#define LOCK_INVALID_OWNER_ID ((lock_owner_id_t)-1)`
- `#define lock_get_caller_owner_id() ((lock_owner_id_t)get_core_num())`
- `#define lock_internal_spin_unlock_with_wait(lock, save) spin_unlock((lock)->spin_lock, save), __wfe()`
- `#define lock_internal_spin_unlock_with_notify(lock, save) spin_unlock((lock)->spin_lock, save), __sev()`
- `#define lock_internal_spin_unlock_with_best_effort_wait_or_timeout(lock, save, until)`
- `#define sync_internal_yield_until_before(until) ((void)0)`

4.2.10.3.3. Functions

`void lock_init (lock_core_t *core, uint lock_num)`

Initialise a lock structure.

4.2.10.3.4. Macro Definition Documentation

lock_owner_id_t

```
#define lock_owner_id_t int8_t
```

type to use to store the 'owner' of a lock.

By default this is int8_t as it only needs to store the core number or -1, however it may be overridden if a larger type is required (e.g. for an RTOS task id)

LOCK_INVALID_OWNER_ID

```
#define LOCK_INVALID_OWNER_ID ((lock_owner_id_t)-1)
```

marker value to use for a lock_owner_id_t which does not refer to any valid owner

lock_get_caller_owner_id

```
#define lock_get_caller_owner_id() ((lock_owner_id_t)get_core_num())
```

return the owner id for the caller

By default this returns the calling core number, but may be overridden (e.g. to return an RTOS task id)

lock_internal_spin_unlock_with_wait

`#define lock_internal_spin_unlock_with_wait(lock, save) spin_unlock((lock)->spin_lock, save), __wfe()`

Atomically unlock the lock's spin lock, and wait for a notification.

Atomic here refers to the fact that it should not be possible for a concurrent lock_internal_spin_unlock_with_notify to insert itself between the spin unlock and this wait in a way that the wait does not see the notification (i.e. causing a missed notification). In other words this method should always wake up in response to a lock_internal_spin_unlock_with_notify for the same lock, which completes after this call starts.

In an ideal implementation, this method would return exactly after the corresponding lock_internal_spin_unlock_with_notify has subsequently been called on the same lock instance, however this method is free to return at *any* point before that; this macro is *always* used in a loop which locks the spin lock, checks the internal locking primitive state and then waits again if the calling thread should not proceed.

By default this macro simply unlocks the spin lock, and then performs a WFE, but may be overridden (e.g. to actually block the RTOS task).

Parameters

`lock`	the lock_core for the primitive which needs to block
`save`	the uint32_t value that should be passed to spin_unlock when the spin lock is unlocked. (i.e. the `PRIMASK` state when the spin lock was acquire

lock_internal_spin_unlock_with_notify

`#define lock_internal_spin_unlock_with_notify(lock, save) spin_unlock((lock)->spin_lock, save), __sev()`

Atomically unlock the lock's spin lock, and send a notification.

Atomic here refers to the fact that it should not be possible for this notification to happen during a lock_internal_spin_unlock_with_wait in a way that that wait does not see the notification (i.e. causing a missed notification). In other words this method should always wake up any lock_internal_spin_unlock_with_wait which started before this call completes.

In an ideal implementation, this method would wake up only the corresponding lock_internal_spin_unlock_with_wait that has been called on the same lock instance, however it is free to wake up any of them, as they will check their condition and then re-wait if necessary/

By default this macro simply unlocks the spin lock, and then performs a SEV, but may be overridden (e.g. to actually un-block RTOS task(s)).

Parameters

`lock`	the lock_core for the primitive which needs to block
`save`	the uint32_t value that should be passed to spin_unlock when the spin lock is unlocked. (i.e. the PRIMASK state when the spin lock was acquire)

lock_internal_spin_unlock_with_best_effort_wait_or_timeout

```
#define lock_internal_spin_unlock_with_best_effort_wait_or_timeout(lock, save, until) ({ \
    spin_unlock((lock)->spin_lock, save);                                            \
    best_effort_wfe_or_timeout(until);                                               \
})
```

Atomically unlock the lock's spin lock, and wait for a notification or a timeout.

Atomic here refers to the fact that it should not be possible for a concurrent lock_internal_spin_unlock_with_notify to insert itself between the spin unlock and this wait in a way that the wait does not see the notification (i.e. causing a missed notification). In other words this method should always wake up in response to a lock_internal_spin_unlock_with_notify for the same lock, which completes after this call starts.

In an ideal implementation, this method would return exactly after the corresponding lock_internal_spin_unlock_with_notify has subsequently been called on the same lock instance or the timeout has been reached, however this method is free to return at *any* point before that; this macro is *always* used in a loop which locks the spin lock, checks the internal locking primitive state and then waits again if the calling thread should not proceed.

By default this simply unlocks the spin lock, and then calls best_effort_wfe_or_timeout but may be overridden (e.g. to actually block the RTOS task with a timeout).

Parameters

`lock`	the lock_core for the primitive which needs to block
`save`	the uint32_t value that should be passed to spin_unlock when the spin lock is unlocked. (i.e. the PRIMASK state when the spin lock was acquire)
`until`	the absolute_time_t value

Returns

true if the timeout has been reached

sync_internal_yield_until_before

```
#define sync_internal_yield_until_before(until) ((void)0)
```

yield to other processing until some time before the requested time

This method is provided for cases where the caller has no useful work to do until the specified time.

By default this method does nothing, however it can be overridden (for example by an RTOS which is able to block the current task until the scheduler tick before the given time)

Parameters

`until`	the absolute_time_t value

4.2.10.3.5. Function Documentation

lock_init

```
void lock_init (lock_core_t * core, uint lock_num)
```

Initialise a lock structure.

Inititalize a lock structure, providing the spin lock number to use for protecting internal state.

Parameters

`core`	Pointer to the lock_core to initialize
`lock_num`	Spin lock number to use for the lock. As the spin lock is only used internally to the locking primitive method implementations, this does not need to be globally unique, however could suffer contention

4.2.10.4. mutex

Mutex API for non IRQ mutual exclusion between cores.

4.2.10.4.1. Detailed Description

Mutexes are application level locks usually used protecting data structures that might be used by multiple threads of execution. Unlike critical sections, the mutex protected code is not necessarily required/expected to complete quickly, as no other system wide locks are held on account of an acquired mutex.

When acquired, the mutex has an owner (see lock_get_caller_owner_id) which with the plain SDK is just the acquiring core, but in an RTOS it could be a task, or an IRQ handler context.

Two variants of mutex are provided; mutex_t (and associated mutex_ functions) is a regular mutex that cannot be acquired recursively by the same owner (a deadlock will occur if you try). recursive_mutex_t (and associated recursive_mutex_ functions) is a recursive mutex that can be recursively obtained by the same caller, at the expense of some more overhead when acquiring and releasing.

It is generally a bad idea to call blocking mutex_ or recursive_mutex_ functions from within an IRQ handler. It is valid to call mutex_try_enter or recursive_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 owner).

NOTE

> For backwards compatibility with version 1.2.0 of the SDK, if the define PICO_MUTEX_ENABLE_SDK120_COMPATIBILITY is set to 1, then the the regular mutex_ functions may also be used for recursive mutexes. This flag will be removed in a future version of the SDK.

See critical_section.h for protecting access between multiple cores AND IRQ handlers

4.2.10.4.2. Macros

- `#define auto_init_mutex(name) static __attribute__((section(".mutex_array"))) mutex_t name`
- `#define auto_init_recursive_mutex(name) static __attribute__((section(".mutex_array"))) recursive_mutex_t name = { .core = { .spin_lock = (spin_lock_t *)1 /* marker for runtime_init */ }, .owner = 0, .enter_count = 0 }`

4.2.10.4.3. Typedefs

`typedef struct mutex mutex_t`

regular (non recursive) mutex instance

4.2.10.4.4. Functions

`static bool critical_section_is_initialized (critical_section_t *crit_sec)`

Test whether a critical_section has been initialized.

`void mutex_init (mutex_t *mtx)`

Initialise a mutex structure.

`void recursive_mutex_init (recursive_mutex_t *mtx)`

Initialise a recursive mutex structure.

`void mutex_enter_blocking (mutex_t *mtx)`

Take ownership of a mutex.

`void recursive_mutex_enter_blocking (recursive_mutex_t *mtx)`

Take ownership of a recursive mutex.

`bool mutex_try_enter (mutex_t *mtx, uint32_t *owner_out)`

Attempt to take ownership of a mutex.

`bool mutex_try_enter_block_until (mutex_t *mtx, absolute_time_t until)`

Attempt to take ownership of a mutex until the specified time.

`bool recursive_mutex_try_enter (recursive_mutex_t *mtx, uint32_t *owner_out)`

Attempt to take ownership of a recursive mutex.

`bool mutex_enter_timeout_ms (mutex_t *mtx, uint32_t timeout_ms)`

Wait for mutex with timeout.

`bool recursive_mutex_enter_timeout_ms (recursive_mutex_t *mtx, uint32_t timeout_ms)`

Wait for recursive mutex with timeout.

`bool mutex_enter_timeout_us (mutex_t *mtx, uint32_t timeout_us)`

Wait for mutex with timeout.

`bool recursive_mutex_enter_timeout_us (recursive_mutex_t *mtx, uint32_t timeout_us)`

Wait for recursive mutex with timeout.

`bool mutex_enter_block_until (mutex_t *mtx, absolute_time_t until)`

Wait for mutex until a specific time.

`bool recursive_mutex_enter_block_until (recursive_mutex_t *mtx, absolute_time_t until)`

Wait for mutex until a specific time.

`void mutex_exit (mutex_t *mtx)`

Release ownership of a mutex.

`void recursive_mutex_exit (recursive_mutex_t *mtx)`

Release ownership of a recursive mutex.

`static bool mutex_is_initialized (mutex_t *mtx)`

Test for mutex initialized state.

`static bool recursive_mutex_is_initialized (recursive_mutex_t *mtx)`

Test for recursive mutex initialized state.

4.2.10.4.5. Macro Definition Documentation

auto_init_mutex

`#define auto_init_mutex(name) static __attribute__((section(".mutex_array"))) mutex_t name`

Helper macro for static definition of mutexes.

A mutex defined as follows:

```
auto_init_mutex(my_mutex);
```

Is equivalent to doing

```
static mutex_t my_mutex;

void my_init_function() {
   mutex_init(&my_mutex);
}
```

But the initialization of the mutex is performed automatically during runtime initialization

auto_init_recursive_mutex

`#define auto_init_recursive_mutex(name) static __attribute__((section(".mutex_array"))) recursive_mutex_t name = { .core = { .spin_lock = (spin_lock_t *)1 /* marker for runtime_init */ }, .owner = 0, .enter_count = 0 }`

Helper macro for static definition of recursive mutexes.

A recursive mutex defined as follows:

```
auto_init_recursive_mutex(my_recursive_mutex);
```

Is equivalent to doing

```
static recursive_mutex_t my_recursive_mutex;

void my_init_function() {
    recursive_mutex_init(&my_recursive_mutex);
}
```

But the initialization of the mutex is performed automatically during runtime initialization

4.2.10.4.6. Typedef Documentation

mutex_t

`typedef struct mutex mutex_t`

regular (non recursive) mutex instance

4.2.10.4.7. Function Documentation

critical_section_is_initialized

`static bool critical_section_is_initialized (critical_section_t * crit_sec) [inline], [static]`

Test whether a critical_section has been initialized.

Parameters

`crit_sec`	Pointer to critical_section structure

Returns

true if the critical section is initialized, false otherwise

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 caller can be granted ownership of the mutex before the timeout expires, then true will be returned and the caller will own the mutex, otherwise false will be returned and the caller will NOT own the mutex.

Parameters

`mtx`	Pointer to mutex structure
`until`	The time after which to return if the caller cannot be granted ownership of the mutex

Returns

true if mutex now owned, false if timeout occurred before ownership could be granted

mutex_enter_blocking

`void mutex_enter_blocking (mutex_t * mtx)`

Take ownership of a mutex.

This function will block until the caller can be granted ownership of the mutex. On return the caller owns the mutex

Parameters

`mtx`	Pointer to mutex structure

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 caller can be granted ownership of the mutex before the timeout expires, then true will be returned and the caller will own the mutex, otherwise false will be returned and the caller will NOT 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 ownership could be granted

mutex_enter_timeout_us

`bool mutex_enter_timeout_us (mutex_t * mtx, uint32_t timeout_us)`

Wait for mutex with timeout.

Wait for up to the specific time to take ownership of the mutex. If the caller can be granted ownership of the mutex before the timeout expires, then true will be returned and the caller will own the mutex, otherwise false will be returned and the caller will NOT own the mutex.

Parameters

`mtx`	Pointer to mutex structure
`timeout_us`	The timeout in microseconds.

Returns

true if mutex now owned, false if timeout occurred before ownership could be granted

mutex_exit

`void mutex_exit (mutex_t * mtx)`

Release ownership of a mutex.

Parameters

`mtx`	Pointer to mutex structure

mutex_init

`void mutex_init (mutex_t * mtx)`

Initialise a mutex structure.

Parameters

`mtx`	Pointer to mutex structure

mutex_is_initialized

`static bool mutex_is_initialized (mutex_t * mtx) [inline], [static]`

Test for mutex initialized state.

Parameters

`mtx`	Pointer to mutex structure

Returns

true if the mutex is initialized, false otherwise

mutex_try_enter

`bool mutex_try_enter (mutex_t * mtx, uint32_t * owner_out)`

Attempt to take ownership of a mutex.

If the mutex wasn't owned, this will claim the mutex for the caller and return true. Otherwise (if the mutex was already owned) this will return false and the caller will NOT own the mutex.

Parameters

`mtx`	Pointer to mutex structure
`owner_out`	If mutex was already owned, and this pointer is non-zero, it will be filled in with the owner id of the current owner of the mutex

Returns

true if mutex now owned, false otherwise

mutex_try_enter_block_until

`bool mutex_try_enter_block_until (mutex_t * mtx, absolute_time_t until)`

Attempt to take ownership of a mutex until the specified time.

If the mutex wasn't owned, this method will immediately claim the mutex for the caller and return true. If the mutex is owned by the caller, this method will immediately return false, If the mutex is owned by someone else, this method will try to claim it until the specified time, returning true if it succeeds, or false on timeout

Parameters

`mtx`	Pointer to mutex structure
`until`	The time after which to return if the caller cannot be granted ownership of the mutex

Returns

true if mutex now owned, false otherwise

recursive_mutex_enter_block_until

`bool recursive_mutex_enter_block_until (recursive_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 caller already has ownership of the mutex or can be granted ownership of the mutex before the timeout expires, then true will be returned and the caller will own the mutex, otherwise false will be returned and the caller will NOT own the mutex.

Parameters

`mtx`	Pointer to recursive mutex structure
`until`	The time after which to return if the caller cannot be granted ownership of the mutex

Returns

true if the recursive mutex (now) owned, false if timeout occurred before ownership could be granted

recursive_mutex_enter_blocking

`void recursive_mutex_enter_blocking (recursive_mutex_t * mtx)`

Take ownership of a recursive mutex.

This function will block until the caller can be granted ownership of the mutex. On return the caller owns the mutex

Parameters

`mtx`	Pointer to recursive mutex structure

recursive_mutex_enter_timeout_ms

`bool recursive_mutex_enter_timeout_ms (recursive_mutex_t * mtx, uint32_t timeout_ms)`

Wait for recursive mutex with timeout.

Wait for up to the specific time to take ownership of the recursive mutex. If the caller already has ownership of the mutex or can be granted ownership of the mutex before the timeout expires, then true will be returned and the caller will own the mutex, otherwise false will be returned and the caller will NOT own the mutex.

Parameters

`mtx`	Pointer to recursive mutex structure
`timeout_ms`	The timeout in milliseconds.

Returns

true if the recursive mutex (now) owned, false if timeout occurred before ownership could be granted

recursive_mutex_enter_timeout_us

`bool recursive_mutex_enter_timeout_us (recursive_mutex_t * mtx, uint32_t timeout_us)`

Wait for recursive mutex with timeout.

Wait for up to the specific time to take ownership of the recursive mutex. If the caller already has ownership of the mutex or can be granted ownership of the mutex before the timeout expires, then true will be returned and the caller will own the mutex, otherwise false will be returned and the caller will NOT own the mutex.

Parameters

`mtx`	Pointer to mutex structure
`timeout_us`	The timeout in microseconds.

Returns

true if the recursive mutex (now) owned, false if timeout occurred before ownership could be granted

recursive_mutex_exit

`void recursive_mutex_exit (recursive_mutex_t * mtx)`

Release ownership of a recursive mutex.

Parameters

`mtx`	Pointer to recursive mutex structure

recursive_mutex_init

`void recursive_mutex_init (recursive_mutex_t * mtx)`

Initialise a recursive mutex structure.

A recursive mutex may be entered in a nested fashion by the same owner

Parameters

`mtx`	Pointer to recursive mutex structure

recursive_mutex_is_initialized

`static bool recursive_mutex_is_initialized (recursive_mutex_t * mtx) [inline], [static]`

Test for recursive mutex initialized state.

Parameters

`mtx` Pointer to recursive mutex structure

Returns

true if the recursive mutex is initialized, false otherwise

recursive_mutex_try_enter

`bool recursive_mutex_try_enter (recursive_mutex_t * mtx, uint32_t * owner_out)`

Attempt to take ownership of a recursive mutex.

If the mutex wasn't owned or was owned by the caller, this will claim the mutex and return true. Otherwise (if the mutex was already owned by another owner) this will return false and the caller will NOT own the mutex.

Parameters

`mtx` Pointer to recursive mutex structure

`owner_out` If mutex was already owned by another owner, and this pointer is non-zero, it will be filled in with the owner id of the current owner of the mutex

Returns

true if the recursive mutex (now) owned, false otherwise

4.2.10.5. sem

Semaphore API for restricting access to a resource.

4.2.10.5.1. Detailed Description

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)

4.2.10.5.2. Functions

`void sem_init (semaphore_t *sem, int16_t initial_permits, int16_t max_permits)`

Initialise a semaphore structure.

`int sem_available (semaphore_t *sem)`

Return number of available permits on the semaphore.

`bool sem_release (semaphore_t *sem)`

Release a permit on a semaphore.

`void sem_reset (semaphore_t *sem, int16_t permits)`

Reset semaphore to a specific number of available permits.

`void sem_acquire_blocking (semaphore_t *sem)`

Acquire a permit from the semaphore.

`bool sem_acquire_timeout_ms (semaphore_t *sem, uint32_t timeout_ms)`

Acquire a permit from a semaphore, with timeout.

`bool sem_acquire_timeout_us (semaphore_t *sem, uint32_t timeout_us)`

Acquire a permit from a semaphore, with timeout.

`bool sem_acquire_block_until (semaphore_t *sem, absolute_time_t until)`

Wait to acquire a permit from a semaphore until a specific time.

`bool sem_try_acquire (semaphore_t *sem)`

Attempt to acquire a permit from a semaphore without blocking.

4.2.10.5.3. Function Documentation

sem_acquire_block_until

`bool sem_acquire_block_until (semaphore_t * sem, absolute_time_t until)`

Wait to acquire a permit from a semaphore until a specific time.

This function will block and wait if no permits are available, until the specified timeout time. If the timeout is reached the function will return false, otherwise it will return true.

Parameters

`sem`	Pointer to semaphore structure
`until`	The time after which to return if the sem is not available.

Returns

true if permit was acquired, false if the until time was reached before acquiring.

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

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 acquire the semaphore, in milliseconds.

Returns

false if timeout reached, true if permit was acquired.

sem_acquire_timeout_us

`bool sem_acquire_timeout_us (semaphore_t * sem, uint32_t timeout_us)`

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_us`	Time to wait to acquire the semaphore, in microseconds.

Returns

false if timeout reached, true if permit was acquired.

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.

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

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 available was increased.

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

sem_try_acquire

```
bool sem_try_acquire (semaphore_t * sem)
```

Attempt to acquire a permit from a semaphore without blocking.

This function will return false without blocking if no permits are available, otherwise it will acquire a permit and return true.

Parameters

`sem`	Pointer to semaphore structure

Returns

true if permit was acquired.

4.2.11. pico_time

API for accurate timestamps, sleeping, and time based callbacks.

4.2.11.1. Detailed Description

> **NOTE**
>
> 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 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 modelled by adding or subtracting a constant value from the unmodified hardware timer.

See also

hardware_timer

4.2.11.2. 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.

4.2.11.3. timestamp

Timestamp functions relating to points in time (including the current time).

4.2.11.3.1. Detailed Description

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.

4.2.11.3.2. Typedefs

`typedef uint64_t absolute_time_t`

An opaque 64 bit timestamp in microseconds.

4.2.11.3.3. Functions

`static uint64_t to_us_since_boot (absolute_time_t t)`

convert an absolute_time_t into a number of microseconds since boot.

`static void update_us_since_boot (absolute_time_t *t, uint64_t us_since_boot)`

update an absolute_time_t value to represent a given number of microseconds since boot

`static absolute_time_t from_us_since_boot (uint64_t us_since_boot)`

convert a number of microseconds since boot to an absolute_time_t

`static absolute_time_t get_absolute_time (void)`

Return a representation of the current time.

`static uint32_t to_ms_since_boot (absolute_time_t t)`

Convert a timestamp into a number of milliseconds since boot.

`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.

`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.

`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.

`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.

`static int64_t absolute_time_diff_us (absolute_time_t from, absolute_time_t to)`

Return the difference in microseconds between two timestamps.

`static absolute_time_t absolute_time_min (absolute_time_t a, absolute_time_t b)`

Return the earlier of two timestamps.

`static bool is_at_the_end_of_time (absolute_time_t t)`

Determine if the given timestamp is "at_the_end_of_time".

`static bool is_nil_time (absolute_time_t t)`

Determine if the given timestamp is nil.

4.2.11.3.4. Variables

`const absolute_time_t at_the_end_of_time`

The timestamp representing the end of time; this is actually not the maximum possible timestamp, but is set to 0x7fffffff_ffffffff microseconds to avoid sign overflows with time arithmetic. This is almost 300,000 years, so should be sufficient.

`const absolute_time_t nil_time`

The timestamp representing a null timestamp.

4.2.11.3.5. Typedef Documentation

absolute_time_t

`typedef uint64_t absolute_time_t`

An opaque 64 bit timestamp in microseconds.

The type is used instead of a raw uint64_t to prevent accidentally passing relative times or times in the wrong time units where an absolute time is required.

note: As of SDK 2.0.0 this type defaults to being a uin64_t (i.e. no protection); it is enabled by setting PICO_OPAQUE_ABSOLUTE_TIME_T to 1

See also

to_us_since_boot()

update_us_since_boot()

4.2.11.3.6. Function Documentation

absolute_time_diff_us

`static int64_t absolute_time_diff_us (absolute_time_t from, absolute_time_t to) [inline], [static]`

Return the difference in microseconds between two timestamps.

> **NOTE**
>
> be careful when diffing against large timestamps (e.g. at_the_end_of_time) as the signed integer may overflow.

Parameters

`from`	the first timestamp
`to`	the second timestamp

Returns

the number of microseconds between the two timestamps (positive if `to` is after `from` except in case of overflow)

absolute_time_min

`static absolute_time_t absolute_time_min (absolute_time_t a, absolute_time_t b) [inline], [static]`

Return the earlier of two timestamps.

Parameters

`a`	the first timestamp
`b`	the second timestamp

Returns

the earlier of the two timestamps

delayed_by_ms

`static absolute_time_t delayed_by_ms (const absolute_time_t t, uint32_t ms) [inline], [static]` 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

delayed_by_us

`static absolute_time_t delayed_by_us (const absolute_time_t t, uint64_t us) [inline], [static]`

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

from_us_since_boot

`static absolute_time_t from_us_since_boot (uint64_t us_since_boot) [inline], [static]`

convert a number of microseconds since boot to an absolute_time_t

fn from_us_since_boot

Parameters

`us_since_boot`	number of microseconds since boot

Returns

an absolute time equivalent to us_since_boot

get_absolute_time

`static absolute_time_t get_absolute_time (void) [inline], [static]`

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()

is_at_the_end_of_time

`static bool is_at_the_end_of_time (absolute_time_t t) [inline], [static]`

Determine if the given timestamp is "at_the_end_of_time".

Parameters

`t`	the timestamp

Returns

true if the timestamp is at_the_end_of_time

See also

at_the_end_of_time

is_nil_time

`static bool is_nil_time (absolute_time_t t) [inline], [static]`

Determine if the given timestamp is nil.

Parameters

`t`	the timestamp

Returns

true if the timestamp is nil

See also

nil_time

make_timeout_time_ms

`static absolute_time_t make_timeout_time_ms (uint32_t ms) [inline], [static]`

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

make_timeout_time_us

`static absolute_time_t make_timeout_time_us (uint64_t us) [inline], [static]`

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

to_ms_since_boot

`static uint32_t to_ms_since_boot (absolute_time_t t) [inline], [static]`

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 milliseconds since boot represented by t

See also

to_us_since_boot()

to_us_since_boot

`static uint64_t to_us_since_boot (absolute_time_t t) [inline], [static]` convert an absolute_time_t into a number of microseconds since boot.

fn to_us_since_boot

Parameters

`t`	the absolute time to convert

Returns

a number of microseconds since boot, equivalent to t

update_us_since_boot

`static void update_us_since_boot (absolute_time_t * t, uint64_t us_since_boot) [inline], [static]`

update an absolute_time_t value to represent a given number of microseconds since boot

fn update_us_since_boot

Parameters

`t`	the absolute time value to update
`us_since_boot`	the number of microseconds since boot to represent. Note this should be representable as a signed 64 bit integer

4.2.11.3.7. Variable Documentation

at_the_end_of_time

`const absolute_time_t at_the_end_of_time`

The timestamp representing the end of time; this is actually not the maximum possible timestamp, but is set to 0x7fffffff_ffffffff microseconds to avoid sign overflows with time arithmetic. This is almost 300,000 years, so should be sufficient.

nil_time

`const absolute_time_t nil_time`

The timestamp representing a null timestamp.

4.2.11.4. sleep

Sleep functions for delaying execution in a lower power state.

4.2.11.4.1. Detailed Description

These functions allow the calling core to sleep. This is a lower powered sleep; waking and re-checking time on every processor event (WFE)

NOTE

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.

See also

busy_wait_until()

busy_wait_us()

busy_wait_us_32()

4.2.11.4.2. Functions

`void sleep_until (absolute_time_t target)`

Wait until after the given timestamp to return.

`void sleep_us (uint64_t us)`

Wait for the given number of microseconds before returning.

`void sleep_ms (uint32_t ms)`

Wait for the given number of milliseconds before returning.

`bool best_effort_wfe_or_timeout (absolute_time_t timeout_timestamp)`

Helper method for blocking on a timeout.

4.2.11.4.3. Function Documentation

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 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 *best_effort* 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:

```
bool my_function_with_timeout_us(uint64_t timeout_us) {
    absolute_time_t timeout_time = make_timeout_time_us(timeout_us);
    do {
        // each time round the loop, we check to see if the condition
        // we are waiting on has happened
        if (my_check_done()) {
            // do something
            return true;
        }
```

```
        // will try to sleep until timeout or the next processor event
    } while (!best_effort_wfe_or_timeout(timeout_time));
    return false; // timed out
}
```

NOTE

This method should always be used in a loop associated with checking another "event" variable, since processor events are a shared resource and can happen for a large number of reasons.

Parameters

`timeout_timestamp`	the timeout time

Returns

true if the target time is reached, false otherwise

sleep_ms

`void sleep_ms (uint32_t ms)`

Wait for the given number of milliseconds before returning.

NOTE

This method attempts to perform a lower power sleep (using WFE) as much as possible.

Parameters

`ms`	the number of milliseconds to sleep

sleep_until

`void sleep_until (absolute_time_t target)`

Wait until after the given timestamp to return.

NOTE

This method attempts to perform a lower power (WFE) sleep

Parameters

`target`	the time after which to return

See also

sleep_us()

busy_wait_until()

sleep_us

`void sleep_us (uint64_t us)`

Wait for the given number of microseconds before returning.

NOTE

This method attempts to perform a lower power (WFE) sleep

Parameters

`us` the number of microseconds to sleep

See also

busy_wait_us()

4.2.11.5. alarm

Alarm functions for scheduling future execution.

4.2.11.5.1. Detailed Description

Alarms are added to alarm pools, which may hold a certain fixed number of active alarms. Each alarm pool utilizes one of four underlying timer_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 timer_alarm IRQ handler, so care must be taken in their implementation.

A default pool is created 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

4.2.11.5.2. Macros

- `#define PICO_TIME_DEFAULT_ALARM_POOL_DISABLED 0`
- `#define PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM 3`
- `#define PICO_TIME_DEFAULT_ALARM_POOL_MAX_TIMERS 16`

4.2.11.5.3. Typedefs

`typedef int32_t alarm_id_t`

The identifier for an alarm.

`typedef int64_t(* alarm_callback_t)(alarm_id_t id, void *user_data)`

User alarm callback.

4.2.11.5.4. Functions

`void alarm_pool_init_default (void)`

Create the default alarm pool (if not already created or disabled)

`alarm_pool_t * alarm_pool_get_default (void)`

The default alarm pool used when alarms are added without specifying an alarm pool, and also used by the SDK to support lower power sleeps and timeouts.

`static alarm_pool_t * alarm_pool_create (uint timer_alarm_num, uint max_timers)`

Create an alarm pool.

`static alarm_pool_t * alarm_pool_create_with_unused_hardware_alarm (uint max_timers)`

Create an alarm pool, claiming an used timer_alarm to back it.

`uint alarm_pool_timer_alarm_num (alarm_pool_t *pool)`

Return the timer alarm used by an alarm pool.

`uint alarm_pool_core_num (alarm_pool_t *pool)`

Return the core number the alarm pool was initialized on (and hence callbacks are called on)

`void alarm_pool_destroy (alarm_pool_t *pool)`

Destroy the alarm pool, cancelling all alarms and freeing up the underlying timer_alarm.

`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.

`alarm_id_t alarm_pool_add_alarm_at_force_in_context (alarm_pool_t *pool, absolute_time_t time, alarm_callback_t callback, void *user_data)`

Add an alarm callback to be called at or after a specific time.

`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.

`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.

`int64_t alarm_pool_remaining_alarm_time_us (alarm_pool_t *pool, alarm_id_t alarm_id)`

Return the time remaining before the next trigger of an alarm.

`int32_t alarm_pool_remaining_alarm_time_ms (alarm_pool_t *pool, alarm_id_t alarm_id)`

Return the time remaining before the next trigger of an alarm.

`bool alarm_pool_cancel_alarm (alarm_pool_t *pool, alarm_id_t alarm_id)`

Cancel an alarm.

`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.

`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.

`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.

`static bool cancel_alarm (alarm_id_t alarm_id)`

Cancel an alarm from the default alarm pool.

`int64_t remaining_alarm_time_us (alarm_id_t alarm_id)`

Return the time remaining before the next trigger of an alarm.

`int32_t remaining_alarm_time_ms (alarm_id_t alarm_id)`

Return the time remaining before the next trigger of an alarm.

4.2.11.5.5. Macro Definition Documentation

PICO_TIME_DEFAULT_ALARM_POOL_DISABLED

```
#define PICO_TIME_DEFAULT_ALARM_POOL_DISABLED 0
```

If 1 then the default alarm pool is disabled (so no timer_alarm is claimed for the pool)

> **NOTE**
>
> Setting to 1 may cause some code not to compile as default timer pool related methods are removed
>
> When the default alarm pool is disabled, _sleep *methods and timeouts are no longer lower powered (they become _busy_wait*)

See also

PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM

alarm_pool_get_default()

PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM

```
#define PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM 3
```

Selects which timer_alarm is used for the default alarm pool.

See also

alarm_pool_get_default()

PICO_TIME_DEFAULT_ALARM_POOL_MAX_TIMERS

```
#define PICO_TIME_DEFAULT_ALARM_POOL_MAX_TIMERS 16
```

Selects the maximum number of concurrent timers in the default alarm pool.

> **NOTE**
>
> For implementation reasons this is limited to PICO_PHEAP_MAX_ENTRIES which defaults to 255

See also

PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM

alarm_pool_get_default()

4.2.11.5.6. Typedef Documentation

alarm_id_t

```
typedef int32_t alarm_id_t
```

The identifier for an alarm.

> **NOTE**
>
> this identifier is signed because <0 is used as an error condition when creating alarms
>
> alarm ids may be reused, however for convenience the implementation makes an attempt to defer reusing as long as possible. You should certainly expect it to be hundreds of ids before one is reused, although in most cases it is more. Nonetheless care must still be taken when cancelling alarms or other functionality based on alarms when the alarm may have expired, as eventually the alarm id may be reused for another alarm.

See also

pico_error_codes

alarm_callback_t

```
typedef int64_t(* alarm_callback_t) (alarm_id_t id, void *user_data)
```

User alarm callback.

Parameters

`id`	the alarm_id as returned when the alarm was added
`user_data`	the user data passed when the alarm was added

Returns

<0 to reschedule the same alarm this many us from the time the alarm was previously scheduled to fire

Returns

>0 to reschedule the same alarm this many us from the time this method returns

Returns

0 to not reschedule the alarm

4.2.11.5.7. Function Documentation

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)
[inline], [static]
```

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.

> **NOTE**
>
> It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

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, and the alarm time falls before or during this call before the alarm can be set, then the callback should be called during (by) this function instead

Returns

>0 the alarm id

Returns

0 if the alarm time passed before or during the call and fire_if_past was false

Returns

<0 if there were no alarm slots available, or other error occurred

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) [inline], [static]`

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.

> **NOTE**
>
> It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

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, and the alarm time falls during this call before the alarm can be set, then the callback should be called during (by) this function instead

Returns

>0 the alarm id

Returns

0 if the alarm time passed before or during the call and fire_if_past was false

Returns

<0 if there were no alarm slots available, or other error occurred

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) [inline], [static]`

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.

> **NOTE**
>
> It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

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, and the alarm time falls during this call before the alarm can be set, then the callback should be called during (by) this function instead

Returns

>0 the alarm id

Returns

0 if the alarm time passed before or during the call and fire_if_past was false

Returns

<0 if there were no alarm slots available, or other error occurred

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.

> **NOTE**
>
> It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

Parameters

`pool`	the alarm pool to use for scheduling the callback (this determines which timer_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, and the alarm time falls before or during this call before the alarm can be set, then the callback should be called during (by) this function instead

Returns

>0 the alarm id for an active (at the time of return) alarm

Returns

0 if the alarm time passed before or during the call and fire_if_past was false

Returns

<0 if there were no alarm slots available, or other error occurred

alarm_pool_add_alarm_at_force_in_context

`alarm_id_t alarm_pool_add_alarm_at_force_in_context (alarm_pool_t * pool, absolute_time_t time, alarm_callback_t callback, void * user_data)`

Add an alarm callback to be called at or after a specific time.

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. Unlike alarm_pool_add_alarm_at, this method guarantees to call the callback from that core even if the time is during this method call or in the past.

NOTE

It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

Parameters

`pool`	the alarm pool to use for scheduling the callback (this determines which timer_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

Returns

>0 the alarm id for an active (at the time of return) alarm

Returns

<0 if there were no alarm slots available, or other error occurred

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) [inline], [static]`

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.

NOTE

It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

Parameters

`pool`	the alarm pool to use for scheduling the callback (this determines which timer_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, and the alarm time falls before or during this call before the alarm can be set, then the callback should be called during (by) this function instead

Returns

>0 the alarm id

Returns

0 if the alarm time passed before or during the call and fire_if_past was false

Returns

<0 if there were no alarm slots available, or other error occurred

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) [inline], [static]`

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.

> **NOTE**
>
> It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

Parameters

`pool`	the alarm pool to use for scheduling the callback (this determines which timer_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, and the alarm time falls during this call before the alarm can be set, then the callback should be called during (by) this function instead

Returns

>0 the alarm id

Returns

0 if the alarm time passed before or during the call and fire_if_past was false

Returns

<0 if there were no alarm slots available, or other error occurred

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

alarm_pool_core_num

`uint alarm_pool_core_num (alarm_pool_t * pool)`

Return the core number the alarm pool was initialized on (and hence callbacks are called on)

Parameters

`pool` the pool

Returns

the core used by the pool

alarm_pool_create

`static alarm_pool_t * alarm_pool_create (uint timer_alarm_num, uint max_timers) [inline], [static]`

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)

NOTE

This method will hard assert if the timer_alarm is already claimed.

Parameters

`timer_alarm_num`	the timer_alarm to use to back this pool
`max_timers`	the maximum number of timers

NOTE

For implementation reasons this is limited to PICO_PHEAP_MAX_ENTRIES which defaults to 255

See also

alarm_pool_get_default()

hardware_claiming

alarm_pool_create_with_unused_hardware_alarm

`static alarm_pool_t * alarm_pool_create_with_unused_hardware_alarm (uint max_timers) [inline], [static]`

Create an alarm pool, claiming an used timer_alarm to back it.

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)

NOTE

This method will hard assert if the there is no free hardware to claim.

Parameters

`max_timers`	the maximum number of timers

> **NOTE**
>
> For implementation reasons this is limited to PICO_PHEAP_MAX_ENTRIES which defaults to 255

See also

alarm_pool_get_default()

hardware_claiming

alarm_pool_destroy

`void alarm_pool_destroy (alarm_pool_t * pool)`

Destroy the alarm pool, cancelling all alarms and freeing up the underlying timer_alarm.

Parameters

`pool`	the pool

alarm_pool_get_default

`alarm_pool_t * alarm_pool_get_default (void)`

The default alarm pool used when alarms are added without specifying an alarm pool, and also used by the SDK to support lower power sleeps and timeouts.

See also

PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM

alarm_pool_init_default

`void alarm_pool_init_default (void)`

Create the default alarm pool (if not already created or disabled)

alarm_pool_remaining_alarm_time_ms

`int32_t alarm_pool_remaining_alarm_time_ms (alarm_pool_t * pool, alarm_id_t alarm_id)`

Return the time remaining before the next trigger of an alarm.

Parameters

`pool`	the alarm_pool containing the alarm
`alarm_id`	the alarm

Returns

>=0 the number of milliseconds before the next trigger (INT32_MAX if the number of ms is higher than can be represented0

Returns

<0 if either the given alarm is not in progress or it has passed

alarm_pool_remaining_alarm_time_us

`int64_t alarm_pool_remaining_alarm_time_us (alarm_pool_t * pool, alarm_id_t alarm_id)`

Return the time remaining before the next trigger of an alarm.

Parameters

`pool`	the alarm_pool containing the alarm
`alarm_id`	the alarm

Returns

>=0 the number of microseconds before the next trigger

Returns

<0 if either the given alarm is not in progress or it has passed

alarm_pool_timer_alarm_num

`uint alarm_pool_timer_alarm_num (alarm_pool_t * pool)`

Return the timer alarm used by an alarm pool.

Parameters

`pool`	the pool

Returns

the timer_alarm used by the pool

cancel_alarm

`static bool cancel_alarm (alarm_id_t alarm_id) [inline], [static]`

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

remaining_alarm_time_ms

`int32_t remaining_alarm_time_ms (alarm_id_t alarm_id)`

Return the time remaining before the next trigger of an alarm.

Parameters

`alarm_id`	the alarm

Returns

>=0 the number of milliseconds before the next trigger (INT32_MAX if the number of ms is higher than can be represented0

Returns

<0 if either the given alarm is not in progress or it has passed

remaining_alarm_time_us

`int64_t remaining_alarm_time_us (alarm_id_t alarm_id)`

Return the time remaining before the next trigger of an alarm.

Parameters

`alarm_id`	the alarm

Returns

>=0 the number of microseconds before the next trigger

Returns

<0 if either the given alarm is not in progress or it has passed

4.2.11.6. repeating_timer

Repeating Timer functions for simple scheduling of repeated execution.

4.2.11.6.1. Detailed Description

> **NOTE**
>
> 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).

4.2.11.6.2. Typedefs

`typedef bool(* repeating_timer_callback_t)(repeating_timer_t *rt)`

Callback for a repeating timer.

4.2.11.6.3. 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.

`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.

`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.

`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.

`bool cancel_repeating_timer (repeating_timer_t *timer)`

Cancel a repeating timer.

4.2.11.6.4. Typedef Documentation

repeating_timer_callback_t

```
typedef bool(* repeating_timer_callback_t) (repeating_timer_t *rt)
```

Callback for a repeating timer.

Parameters

`rt`	repeating time structure containing information about the repeating time. user_data is of primary important to the user

Returns

true to continue repeating, false to stop.

4.2.11.6.5. Function Documentation

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) [inline], [static]`

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.

> **NOTE**
>
> It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

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.

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) [inline], [static]`

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.

> **NOTE**
>
> It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

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.

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) [inline], [static]`

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.

NOTE

> It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

Parameters

`pool`	the alarm pool to use for scheduling the repeating timer (this determines which timer_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.

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.

NOTE

> It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

Parameters

`pool`	the alarm pool to use for scheduling the repeating timer (this determines which timer_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.

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

4.2.12. pico_unique_id

Unique device ID access API.

4.2.12.1. Detailed Description

RP2040 does not have an on-board unique identifier (all instances of RP2040 silicon are identical and have no persistent state). However, RP2040 boots from serial NOR flash devices which have at least a 64-bit unique ID as a standard feature, and there is a 1:1 association between RP2040 and flash, so this is suitable for use as a unique identifier for an RP2040-based board.

This library injects a call to the flash_get_unique_id function from the hardware_flash library, to run before main, and stores the result in a static location which can safely be accessed at any time via pico_get_unique_id().

This avoids some pitfalls of the hardware_flash API, which requires any flash-resident interrupt routines to be disabled when called into.

On boards using RP2350, the unique identifier is read from OTP memory on boot.

4.2.12.2. Functions

`void pico_get_unique_board_id (pico_unique_board_id_t *id_out)`

Get unique ID.

`void pico_get_unique_board_id_string (char *id_out, uint len)`

Get unique ID in string format.

4.2.12.3. Function Documentation

4.2.12.3.1. pico_get_unique_board_id

`void pico_get_unique_board_id (pico_unique_board_id_t * id_out)`

Get unique ID.

Get the unique 64-bit device identifier.

On an RP2040-based board, the unique identifier is retrieved from the external NOR flash device at boot, or for

PICO_NO_FLASH builds the unique identifier is set to all 0xEE.

On an RP2350-based board, the unique identifier is retrieved from OTP memory at boot.

Parameters

`id_out`	a pointer to a pico_unique_board_id_t struct, to which the identifier will be written

4.2.12.3.2. pico_get_unique_board_id_string

`void pico_get_unique_board_id_string (char * id_out, uint len)`

Get unique ID in string format.

Get the unique 64-bit device identifier formatted as a 0-terminated ASCII hex string.

On an RP2040-based board, the unique identifier is retrieved from the external NOR flash device at boot, or for PICO_NO_FLASH builds the unique identifier is set to all 0xEE.

On an RP2350-based board, the unique identifier is retrieved from OTP memory at boot.

Parameters

`id_out`	a pointer to a char buffer of size len, to which the identifier will be written
`len`	the size of id_out. For full serial, len >= 2 * PICO_UNIQUE_BOARD_ID_SIZE_BYTES + 1

4.2.13. pico_util

Useful data structures and utility functions.

4.2.13.1. Modules

datetime
    Date/Time formatting.

pheap
    Pairing Heap Implementation.

queue
    Multi-core and IRQ safe queue implementation.

4.2.13.2. datetime

Date/Time formatting.

4.2.13.2.1. Functions

`struct tm * pico_localtime_r (const time_t *time, struct tm *tm)`
    localtime_r implementation for use by the pico_util datetime functions

`time_t pico_mktime (struct tm *tm)`
    mktime implementation for use by the pico_util datetime functions

4.2.13.2.2. Function Documentation

pico_localtime_r

`struct tm * pico_localtime_r (const time_t * time, struct tm * tm)`

localtime_r implementation for use by the pico_util datetime functions

This method calls localtime_r from the C library by default, but is declared as a weak implementation to allow user code to override it

pico_mktime

`time_t pico_mktime (struct tm * tm)`

mktime implementation for use by the pico_util datetime functions

This method calls mktime from the C library by default, but is declared as a weak implementation to allow user code to override it

4.2.13.3. pheap

Pairing Heap Implementation.

4.2.13.3.1. Detailed Description

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_and_free_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

4.2.13.3.2. Macros

- `#define PHEAP_DEFINE_STATIC(name, _max_nodes)`

4.2.13.3.3. Typedefs

`typedef bool(* pheap_comparator)(void *user_data, pheap_node_id_t a, pheap_node_id_t b)`

A user comparator function for nodes in a pairing heap.

4.2.13.3.4. Functions

`pheap_t * ph_create (uint max_nodes, pheap_comparator comparator, void *user_data)`

Create a pairing heap, which effectively maintains an efficient sorted ordering of nodes. The heap itself stores no user per-node state, it is expected that the user maintains a companion array. A comparator function must be provided so that the heap implementation can determine the relative ordering of nodes.

`void ph_clear (pheap_t *heap)`

Removes all nodes from the pairing heap.

`void ph_destroy (pheap_t *heap)`

De-allocates a pairing heap.

`static pheap_node_id_t ph_new_node (pheap_t *heap)`

Allocate a new node from the unused space in the heap.

`static pheap_node_id_t ph_insert_node (pheap_t *heap, pheap_node_id_t id)`

Inserts a node into the heap.

`static pheap_node_id_t ph_peek_head (pheap_t *heap)`

Returns the head node in the heap, i.e. the node which compares first, but without removing it from the heap.

`pheap_node_id_t ph_remove_head (pheap_t *heap, bool free)`

Remove the head node from the pairing heap. This head node is the node which compares first in the logical ordering provided by the comparator.

`static pheap_node_id_t ph_remove_and_free_head (pheap_t *heap)`

Remove the head node from the pairing heap. This head node is the node which compares first in the logical ordering provided by the comparator.

`bool ph_remove_and_free_node (pheap_t *heap, pheap_node_id_t id)`

Remove and free an arbitrary node from the pairing heap. This is a more costly operation than removing the head via ph_remove_and_free_head()

`static bool ph_contains_node (pheap_t *heap, pheap_node_id_t id)`

Determine if the heap contains a given node. Note containment refers to whether the node is inserted (ph_insert_node()) vs allocated (ph_new_node())

`static void ph_free_node (pheap_t *heap, pheap_node_id_t id)`

Free a node that is not currently in the heap, but has been allocated.

`void ph_dump (pheap_t *heap, void(*dump_key)(pheap_node_id_t id, void *user_data), void *user_data)`

Print a representation of the heap for debugging.

`void ph_post_alloc_init (pheap_t *heap, uint max_nodes, pheap_comparator comparator, void *user_data)`

Initialize a statically allocated heap (ph_create() using the C heap). The heap member `nodes` must be allocated of size max_nodes.

4.2.13.3.5. Macro Definition Documentation

PHEAP_DEFINE_STATIC

```
#define PHEAP_DEFINE_STATIC(name, _max_nodes) static_assert(_max_nodes && _max_nodes < (1u << (8 *
sizeof(pheap_node_id_t))), ""); \
    static pheap_node_t name ## _nodes[_max_nodes]; \
    static pheap_t name = { \
            .nodes = name ## _nodes, \
            .max_nodes = _max_nodes \
    };
```

Define a statically allocated pairing heap. This must be initialized by ph_post_alloc_init.

4.2.13.3.6. Typedef Documentation

pheap_comparator

`typedef bool(* pheap_comparator) (void *user_data, pheap_node_id_t a, pheap_node_id_t b)`

A user comparator function for nodes in a pairing heap.

Returns

true if a < b in natural order. Note this relative ordering must be stable from call to call.

4.2.13.3.7. Function Documentation

ph_clear

`void ph_clear (pheap_t * heap)`

Removes all nodes from the pairing heap.

Parameters

`heap`	the heap

ph_contains_node

`static bool ph_contains_node (pheap_t * heap, pheap_node_id_t id) [inline], [static]`

Determine if the heap contains a given node. Note containment refers to whether the node is inserted (ph_insert_node()) vs allocated (ph_new_node())

Parameters

`heap`	the heap
`id`	the id of the node

Returns

true if the heap contains a node with the given id, false otherwise.

ph_create

`pheap_t * ph_create (uint max_nodes, pheap_comparator comparator, void * user_data)`

Create a pairing heap, which effectively maintains an efficient sorted ordering of nodes. The heap itself stores no user per-node state, it is expected that the user maintains a companion array. A comparator function must be provided so that the heap implementation can determine the relative ordering of nodes.

Parameters

`max_nodes`	the maximum number of nodes that may be in the heap (this is bounded by PICO_PHEAP_MAX_ENTRIES which defaults to 255 to be able to store indexes in a single byte).
`comparator`	the node comparison function
`user_data`	a user data pointer associated with the heap that is provided in callbacks

Returns

a newly allocated and initialized heap

ph_destroy

`void ph_destroy (pheap_t * heap)`

De-allocates a pairing heap.

Note this method must *ONLY* be called on heaps created by ph_create()

Parameters

`heap`	the heap

ph_dump

`void ph_dump (pheap_t * heap, void(*)(pheap_node_id_t id, void *user_data) dump_key, void * user_data)`

Print a representation of the heap for debugging.

Parameters

`heap`	the heap
`dump_key`	a method to print a node value
`user_data`	the user data to pass to the dump_key method

ph_free_node

`static void ph_free_node (pheap_t * heap, pheap_node_id_t id) [inline], [static]`

Free a node that is not currently in the heap, but has been allocated.

Parameters

`heap`	the heap
`id`	the id of the node

ph_insert_node

`static pheap_node_id_t ph_insert_node (pheap_t * heap, pheap_node_id_t id) [inline], [static]`

Inserts a node into the heap.

This method inserts a node (previously allocated by ph_new_node()) into the heap, determining the correct order by calling the heap's comparator

Parameters

`heap`	the heap
`id`	the id of the node to insert

Returns

the id of the new head of the pairing heap (i.e. node that compares first)

ph_new_node

`static pheap_node_id_t ph_new_node (pheap_t * heap) [inline], [static]`

Allocate a new node from the unused space in the heap.

Parameters

`heap`	the heap

Returns

an identifier for the node, or 0 if the heap is full

ph_peek_head

`static pheap_node_id_t ph_peek_head (pheap_t * heap) [inline], [static]`

Returns the head node in the heap, i.e. the node which compares first, but without removing it from the heap.

Parameters

`heap`	the heap

Returns

the current head node id

ph_post_alloc_init

`void ph_post_alloc_init (pheap_t * heap, uint max_nodes, pheap_comparator comparator, void * user_data)`

Initialize a statically allocated heap (ph_create() using the C heap). The heap member `nodes` must be allocated of size max_nodes.

Parameters

`heap`	the heap
`max_nodes`	the max number of nodes in the heap (matching the size of the heap's nodes array)
`comparator`	the comparator for the heap
`user_data`	the user data for the heap.

ph_remove_and_free_head

`static pheap_node_id_t ph_remove_and_free_head (pheap_t * heap) [inline], [static]`

Remove the head node from the pairing heap. This head node is the node which compares first in the logical ordering provided by the comparator.

Note that the returned id will be freed, and thus may be re-used by future node allocations, so the caller should retrieve any per node state from the companion array before modifying the heap further.

Parameters

`heap`	the heap

Returns

the old head node id.

ph_remove_and_free_node

`bool ph_remove_and_free_node (pheap_t * heap, pheap_node_id_t id)`

Remove and free an arbitrary node from the pairing heap. This is a more costly operation than removing the head via ph_remove_and_free_head()

Parameters

`heap`	the heap
`id`	the id of the node to free

Returns

true if the the node was in the heap, false otherwise

ph_remove_head

`pheap_node_id_t ph_remove_head (pheap_t * heap, bool free)`

Remove the head node from the pairing heap. This head node is the node which compares first in the logical ordering provided by the comparator.

Note that in the case of free == true, the returned id is no longer allocated and may be re-used by future node allocations, so the caller should retrieve any per node state from the companion array before modifying the heap further.

Parameters

`heap`	the heap
`free`	true if the id is also to be freed; false if not - useful if the caller may wish to re-insert an item with the same id)

Returns

the old head node id.

4.2.13.4. queue

Multi-core and IRQ safe queue implementation.

4.2.13.4.1. Detailed Description

Note that this queue stores values of a specified size, and pushed values are copied into the queue

4.2.13.4.2. 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.

`static void queue_init (queue_t *q, uint element_size, uint element_count)`

Initialise a queue, allocating a (possibly shared) spinlock.

`void queue_free (queue_t *q)`

Destroy the specified queue.

`static uint queue_get_level_unsafe (queue_t *q)`

Unsafe check of level of the specified queue.

`static uint queue_get_level (queue_t *q)`

Check of level of the specified queue.

`static bool queue_is_empty (queue_t *q)`

Check if queue is empty.

`static bool queue_is_full (queue_t *q)`

Check if queue is full.

`bool queue_try_add (queue_t *q, const void *data)`

Non-blocking add value queue if not full.

`bool queue_try_remove (queue_t *q, void *data)`

Non-blocking removal of entry from the queue if non empty.

`bool queue_try_peek (queue_t *q, void *data)`

Non-blocking peek at the next item to be removed from the queue.

`void queue_add_blocking (queue_t *q, const void *data)`

Blocking add of value to queue.

`void queue_remove_blocking (queue_t *q, void *data)`

Blocking remove entry from queue.

`void queue_peek_blocking (queue_t *q, void *data)`

Blocking peek at next value to be removed from queue.

4.2.13.4.3. Function Documentation

queue_add_blocking

`void queue_add_blocking (queue_t * q, const void * data)`

Blocking add of value to queue.

Parameters

`q`	Pointer to a queue_t structure, used as a handle
`data`	Pointer to value to be copied into the queue

If the queue is full this function will block, until a removal happens on the queue

queue_free

`void queue_free (queue_t * q)`

Destroy the specified queue.

Parameters

`q`	Pointer to a queue_t structure, used as a handle

Does not deallocate the queue_t structure itself.

queue_get_level

`static uint queue_get_level (queue_t * q) [inline], [static]`

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

queue_get_level_unsafe

`static uint queue_get_level_unsafe (queue_t * q) [inline], [static]`

Unsafe 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

This does not use the spinlock, so may return incorrect results if the spin lock is not externally locked

queue_init

`static void queue_init (queue_t * q, uint element_size, uint element_count) [inline], [static]`

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

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

queue_is_empty

`static bool queue_is_empty (queue_t * q) [inline], [static]`

Check if queue is empty.

Parameters

`q`	Pointer to a queue_t structure, used as a handle

Returns

true if queue is empty, false otherwise

This function is interrupt and multicore safe.

queue_is_full

`static bool queue_is_full (queue_t * q) [inline], [static]`

Check if queue is full.

Parameters

`q`	Pointer to a queue_t structure, used as a handle

Returns

true if queue is full, false otherwise

This function is interrupt and multicore safe.

queue_peek_blocking

`void queue_peek_blocking (queue_t * q, void * data)`

Blocking peek at next value to be removed from queue.

Parameters

`q`	Pointer to a queue_t structure, used as a handle
`data`	Pointer to the location to receive the peeked value, or NULL if the data isn't required

If the queue is empty function will block until a value is added

queue_remove_blocking

`void queue_remove_blocking (queue_t * q, void * data)`

Blocking remove entry from queue.

Parameters

`q`	Pointer to a queue_t structure, used as a handle
`data`	Pointer to the location to receive the removed value, or NULL if the data isn't required

If the queue is empty this function will block until a value is added.

queue_try_add

`bool queue_try_add (queue_t * q, const void * data)`

Non-blocking add value queue if not full.

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

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.

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.

Parameters

`q`	Pointer to a queue_t structure, used as a handle
`data`	Pointer to the location to receive the peeked value, or NULL if the data isn't required

Returns

true if there was a value to peek

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.

queue_try_remove

```
bool queue_try_remove (queue_t * q, void * data)
```

Non-blocking removal of entry from the queue if non empty.

Parameters

`q`	Pointer to a queue_t structure, used as a handle
`data`	Pointer to the location to receive the removed value, or NULL if the data isn't required

Returns

true if a value was removed

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.

4.3. Third-party Libraries

Third party libraries for implementing high level functionality.

tinyusb_device	TinyUSB Device-mode support for the RP2040. The TinyUSB documentation site can be found here.
tinyusb_host	TinyUSB Host-mode support for the RP2040.

4.3.1. tinyusb_device

TinyUSB Device-mode support for the RP2040. The TinyUSB documentation site can be found here.

4.3.2. tinyusb_host

TinyUSB Host-mode support for the RP2040.

4.4. Networking Libraries

Functions for implementing networking

pico_btstack	Integration/wrapper libraries for BTstack the documentation for which is here.
pico_lwip	Integration/wrapper libraries for lwIP the documentation for which is here.
pico_lwip_arch	lwIP compiler adapters. This is not included by default in `pico_lwip in` case you wish to implement your own.
pico_lwip_freertos	Glue library for integration lwIP in `NO_SYS=0` mode with the SDK.
pico_lwip_nosys	Glue library for integration lwIP in `NO_SYS=1` mode with the SDK.
pico_cyw43_driver	A wrapper around the lower level cyw43_driver, that integrates it with pico_async_context for handling background work.
pico_btstack_cyw43	Low-level Bluetooth HCI support.
pico_cyw43_arch	Architecture for integrating the CYW43 driver (for the wireless on Pico W) and lwIP (for TCP/IP stack) into the SDK. It is also necessary for accessing the on-board LED on Pico W.
cyw43_driver	Driver used for Pico W wireless.
cyw43_ll	Low Level CYW43 driver interface.

4.4.1. pico_btstack

Integration/wrapper libraries for BTstack the documentation for which is here.

4.4.1.1. Detailed Description

A supplemental license for BTstack (in addition to the stock BTstack licensing terms) is provided here.

The `pico_btstack_ble library` adds the support needed for Bluetooth Low Energy (BLE). The `pico_btstack_classic library` adds the support needed for Bluetooth Classic. You can link to either library individually, or to both libraries thus enabling dual-mode support provided by BTstack.

To use BTstack you need to provide a `btstack_config.h` file in your source tree and add its location to your include path. The BTstack configuration macros `ENABLE_CLASSIC` and `ENABLE_BLE` are defined for you when you link the `pico_btstack_classic` and `pico_btstack_ble` libraries respectively, so you should not define them yourself.

For more details, see How to configure BTstack and the relevant pico-examples.

The follow libraries are provided for you to link.

- `pico_btstack_ble` - Adds Bluetooth Low Energy (LE) support.
- `pico_btstack_classic` - Adds Bluetooth Classic support.
- `pico_btstack_sbc_encoder` - Adds Bluetooth Sub Band Coding (SBC) encoder support.
- `pico_btstack_sbc_decoder` - Adds Bluetooth Sub Band Coding (SBC) decoder support.
- `pico_btstack_bnep_lwip` - Adds Bluetooth Network Encapsulation Protocol (BNEP) support using LwIP.
- `pico_btstack_bnep_lwip_sys_freertos` - Adds Bluetooth Network Encapsulation Protocol (BNEP) support using LwIP with FreeRTOS for NO_SYS=0.
- `pico_btstack_mesh` - Adds Bluetooth mesh support from BTstack.

NOTE

The CMake function pico_btstack_make_gatt_header can be used to run the BTstack compile_gatt tool to make a GATT header file from a BTstack GATT file.

See also

pico_btstack_cyw43 in pico_cyw43_driver, which adds the cyw43 driver support needed for BTstack including BTstack run loop support.

4.4.1.2. Functions

`const hal_flash_bank_t * pico_flash_bank_instance (void)`

Return the singleton BTstack HAL flash instance, used for non-volatile storage.

`const btstack_run_loop_t * btstack_run_loop_async_context_get_instance (async_context_t *context)`

Initialize and return the singleton BTstack run loop instance that integrates with the async_context API.

`const btstack_chipset_t * btstack_chipset_cyw43_instance (void)`

Return the singleton BTstack chipset CY43 API instance.

4.4.1.3. Function Documentation

4.4.1.3.1. btstack_chipset_cyw43_instance

`const btstack_chipset_t * btstack_chipset_cyw43_instance (void)`

Return the singleton BTstack chipset CY43 API instance.

4.4.1.3.2. btstack_run_loop_async_context_get_instance

`const btstack_run_loop_t * btstack_run_loop_async_context_get_instance (async_context_t * context)`

Initialize and return the singleton BTstack run loop instance that integrates with the async_context API.

Parameters

`context` the async_context instance that provides the abstraction for handling asynchronous work.

Returns

the BTstack run loop instance

4.4.1.3.3. pico_flash_bank_instance

`const hal_flash_bank_t * pico_flash_bank_instance (void)`

Return the singleton BTstack HAL flash instance, used for non-volatile storage.

> **NOTE**
>
> By default two sectors at the end of flash are used (see `PICO_FLASH_BANK_STORAGE_OFFSET` and `PICO_FLASH_BANK_TOTAL_SIZE`)

4.4.2. pico_lwip

Integration/wrapper libraries for lwIP the documentation for which is here.

4.4.2.1. Detailed Description

The main `pico_lwip library` itself aggregates the lwIP RAW API: `pico_lwip_core, pico_lwip_core4, pico_lwip_core6, pico_lwip_api, pico_lwip_netif, pico_lwip_sixlowpan and pico_lwip_ppp.`

If you wish to run in NO_SYS=1 mode, then you can link `pico_lwip along` with pico_lwip_nosys.

If you wish to run in NO_SYS=0 mode, then you can link `pico_lwip with` (for instance) pico_lwip_freertos, and also link in pico_lwip_api for the additional blocking/thread-safe APIs.

Additionally you must link in pico_lwip_arch unless you provide your own compiler bindings for lwIP.

Additional individual pieces of lwIP functionality are available à la cart, by linking any of the libraries below.

The following libraries are provided that contain exactly the equivalent lwIP functionality groups:

- `pico_lwip_core -`
- `pico_lwip_core4 -`
- `pico_lwip_core6 -`
- `pico_lwip_netif -`
- `pico_lwip_sixlowpan -`
- `pico_lwip_ppp -`
- `pico_lwip_api -`

The following libraries are provided that contain exactly the equivalent lwIP application support:

- `pico_lwip_snmp -`
- `pico_lwip_http -`
- `pico_lwip_makefsdata -`
- `pico_lwip_iperf -`
- `pico_lwip_smtp -`
- `pico_lwip_sntp -`
- `pico_lwip_mdns -`
- `pico_lwip_netbios -`
- `pico_lwip_tftp -`
- `pico_lwip_mbedtls -`
- `pico_lwip_mqtt -`

4.4.2.2. Modules

pico_lwip_arch

lwIP compiler adapters. This is not included by default in `pico_lwip in` case you wish to implement your own.

pico_lwip_freertos

Glue library for integration lwIP in `NO_SYS=0` mode with the SDK.

pico_lwip_nosys

Glue library for integration lwIP in `NO_SYS=1` mode with the SDK.

4.4.2.3. pico_lwip_arch

lwIP compiler adapters. This is not included by default in `pico_lwip in` case you wish to implement your own.

4.4.2.4. pico_lwip_freertos

Glue library for integration lwIP in `NO_SYS=0` mode with the SDK.

4.4.2.4.1. Detailed Description

Simple `init` and `deinit` are all that is required to hook up lwIP (with full blocking API support) via an async_context instance

4.4.2.4.2. Functions

`bool lwip_freertos_init (async_context_t *context)`

Initializes lwIP (NO_SYS=0 mode) support support for FreeRTOS using the provided async_context.

`void lwip_freertos_deinit (async_context_t *context)`

De-initialize lwIP (NO_SYS=0 mode) support for FreeRTOS.

4.4.2.4.3. Function Documentation

lwip_freertos_deinit

`void lwip_freertos_deinit (async_context_t * context)`

De-initialize lwIP (NO_SYS=0 mode) support for FreeRTOS.

Note that since lwIP may only be initialized once, and doesn't itself provide a shutdown mechanism, lwIP itself may still consume resources.

It is however safe to call lwip_freertos_init again later.

Parameters

`context`	the async_context the lwip_freertos support was added to via lwip_freertos_init

lwip_freertos_init

`bool lwip_freertos_init (async_context_t * context)`

Initializes lwIP (NO_SYS=0 mode) support support for FreeRTOS using the provided async_context.

If the initialization succeeds, lwip_freertos_deinit() can be called to shutdown lwIP support

Parameters

`context`	the async_context instance that provides the abstraction for handling asynchronous work. Note in general this would be an async_context_freertos instance, though it doesn't have to be.

Returns

true if the initialization succeeded

4.4.2.5. pico_lwip_nosys

Glue library for integration lwIP in `NO_SYS=1` mode with the SDK.

4.4.2.5.1. Detailed Description

Simple `init` and `deinit` are all that is required to hook up lwIP via an async_context instance.

4.4.2.5.2. Functions

`bool lwip_nosys_init (async_context_t *context)`

Initializes lwIP (NO_SYS=1 mode) support support using the provided async_context.

`void lwip_nosys_deinit (async_context_t *context)`

De-initialize lwIP (NO_SYS=1 mode) support.

4.4.2.5.3. Function Documentation

lwip_nosys_deinit

`void lwip_nosys_deinit (async_context_t * context)`

De-initialize lwIP (NO_SYS=1 mode) support.

Note that since lwIP may only be initialized once, and doesn't itself provide a shutdown mechanism, lwIP itself may still consume resources

It is however safe to call lwip_nosys_init again later.

Parameters

`context`	the async_context the lwip_nosys support was added to via lwip_nosys_init

lwip_nosys_init

`bool lwip_nosys_init (async_context_t * context)`

Initializes lwIP (NO_SYS=1 mode) support support using the provided async_context.

If the initialization succeeds, lwip_nosys_deinit() can be called to shutdown lwIP support

Parameters

`context`	the async_context instance that provides the abstraction for handling asynchronous work.

Returns

true if the initialization succeeded

4.4.3. pico_cyw43_driver

A wrapper around the lower level cyw43_driver, that integrates it with pico_async_context for handling background work.

4.4.3.1. Modules

pico_btstack_cyw43

Low-level Bluetooth HCI support.

4.4.3.2. Functions

`const hci_transport_t * hci_transport_cyw43_instance (void)`

Get the Bluetooth HCI transport instance for cyw43.

`bool cyw43_driver_init (struct async_context *context)`

Initializes the lower level cyw43_driver and integrates it with the provided async_context.

`void cyw43_driver_deinit (struct async_context *context)`

De-initialize the lowever level cyw43_driver and unhooks it from the async_context.

4.4.3.3. Function Documentation

4.4.3.3.1. cyw43_driver_deinit

`void cyw43_driver_deinit (struct async_context * context)`

De-initialize the lowever level cyw43_driver and unhooks it from the async_context.

Parameters

`context`	the async_context the cyw43_driver support was added to via cyw43_driver_init

4.4.3.3.2. cyw43_driver_init

`bool cyw43_driver_init (struct async_context * context)`

Initializes the lower level cyw43_driver and integrates it with the provided async_context.

If the initialization succeeds, lwip_nosys_deinit() can be called to shutdown lwIP support

Parameters

`context`	the async_context instance that provides the abstraction for handling asynchronous work.

Returns

true if the initialization succeeded

4.4.3.3.3. hci_transport_cyw43_instance

`const hci_transport_t * hci_transport_cyw43_instance (void)`

Get the Bluetooth HCI transport instance for cyw43.

Returns

An instantiation of the hci_transport_t interface for the cyw43 chipset

4.4.3.4. pico_btstack_cyw43

Low-level Bluetooth HCI support.

4.4.3.4.1. Detailed Description

This library provides utility functions to initialise and de-initialise BTstack for CYW43,

4.4.4. pico_cyw43_arch

Architecture for integrating the CYW43 driver (for the wireless on Pico W) and lwIP (for TCP/IP stack) into the SDK. It is also necessary for accessing the on-board LED on Pico W.

4.4.4.1. Detailed Description

Both the low level `cyw43_driver` and the lwIP stack require periodic servicing, and have limitations on whether they can be called from multiple cores/threads.

`pico_cyw43_arch` attempts to abstract these complications into several behavioral groups:

- *'poll'* - This not multi-core/IRQ safe, and requires the user to call cyw43_arch_poll periodically from their main loop
- *'thread_safe_background'* - This is multi-core/thread/task safe, and maintenance of the driver and TCP/IP stack is handled automatically in the background
- *'freertos'* - This is multi-core/thread/task safe, and uses a separate FreeRTOS task to handle lwIP and and driver work.

As of right now, lwIP is the only supported TCP/IP stack, however the use of `pico_cyw43_arch` is intended to be independent of the particular TCP/IP stack used (and possibly Bluetooth stack used) in the future. For this reason, the integration of lwIP is handled in the base (`pico_cyw43_arch`) library based on the #define CYW43_LWIP used by the `cyw43_driver`.

NOTE

> As of version 1.5.0 of the Raspberry Pi Pico SDK, the `pico_cyw43_arch` library no longer directly implements the distinct behavioral abstractions. This is now handled by the more general pico_async_context library. The user facing behavior of pico_cyw43_arch has not changed as a result of this implementation detail, however pico_cyw43_arch is now just a thin wrapper which creates an appropriate async_context and makes a simple call to add lwIP or cyw43_driver support as appropriate. You are free to perform this context creation and adding of lwIP, cyw43_driver or indeed any other additional future protocol/driver support to your async_context, however for now pico_cyw43_arch does still provide a few cyw43_ specific (i.e. Pico W) APIs for connection management, locking and GPIO interaction.
>
> The connection management APIs at least may be moved to a more generic library in a future release. The locking methods are now backed by their pico_async_context equivalents, and those methods may be used interchangeably (see cyw43_arch_lwip_begin, cyw43_arch_lwip_end and cyw43_arch_lwip_check for more details).
>
> For examples of creating of your own async_context and addition of `cyw43_driver` and `lwIP` support, please refer to the specific source files `cyw43_arch_poll.c`, `cyw43_arch_threadsafe_background.c` and `cyw43_arch_freertos.c`.

Whilst you can use the `pico_cyw43_arch` library directly and specify CYW43_LWIP (and other defines) yourself, several other libraries are made available to the build which aggregate the defines and other dependencies for you:

- **pico_cyw43_arch_lwip_poll** - For using the RAW lwIP API (in `NO_SYS=1` mode) without any background processing or multi-core/thread safety.

  The user must call cyw43_arch_poll periodically from their main loop.

This wrapper library:

- Sets `CYW43_LWIP=1` to enable lwIP support in `pico_cyw43_arch` and `cyw43_driver`.
- Sets `PICO_CYW43_ARCH_POLL=1` to select the polling behavior.
- Adds the `pico_lwip` as a dependency to pull in lwIP.

• **pico_cyw43_arch_lwip_threadsafe_background** - For using the RAW lwIP API (in `NO_SYS=1` mode) with multi-core/thread safety, and automatic servicing of the `cyw43_driver` and lwIP in background.

Calls into the `cyw43_driver` high level API (cyw43.h) may be made from either core or from lwIP callbacks, however calls into lwIP (which is not thread-safe) other than those made from lwIP callbacks, must be bracketed with cyw43_arch_lwip_begin and cyw43_arch_lwip_end. It is fine to bracket calls made from within lwIP callbacks too; you just don't have to.

NOTE

lwIP callbacks happen in a (low priority) IRQ context (similar to an alarm callback), so care should be taken when interacting with other code.

This wrapper library:

- Sets `CYW43_LWIP=1` to enable lwIP support in `pico_cyw43_arch` and `cyw43_driver`
- Sets `PICO_CYW43_ARCH_THREADSAFE_BACKGROUND=1` to select the thread-safe/non-polling behavior.
- Adds the pico_lwip as a dependency to pull in lwIP.

This library *can* also be used under the RP2040 port of FreeRTOS with lwIP in `NO_SYS=1` mode (allowing you to call `cyw43_driver` APIs from any task, and to call lwIP from lwIP callbacks, or from any task if you bracket the calls with cyw43_arch_lwip_begin and cyw43_arch_lwip_end. Again, you should be careful about what you do in lwIP callbacks, as you cannot call most FreeRTOS APIs from within an IRQ context. Unless you have good reason, you should probably use the full FreeRTOS integration (with `NO_SYS=0`) provided by `pico_cyw43_arch_lwip_sys_freertos`.

• **pico_cyw43_arch_lwip_sys_freertos** - For using the full lwIP API including blocking sockets in OS (`NO_SYS=0`) mode, along with with multi-core/task/thread safety, and automatic servicing of the `cyw43_driver` and the lwIP stack.

This wrapper library:

- Sets `CYW43_LWIP=1` to enable lwIP support in `pico_cyw43_arch` and `cyw43_driver`.
- Sets `PICO_CYW43_ARCH_FREERTOS=1` to select the NO_SYS=0 lwip/FreeRTOS integration
- Sets `LWIP_PROVIDE_ERRNO=1` to provide error numbers needed for compilation without an OS
- Adds the `pico_lwip` as a dependency to pull in lwIP.
- Adds the lwIP/FreeRTOS code from lwip-contrib (in the contrib directory of lwIP)

Calls into the `cyw43_driver` high level API (cyw43.h) may be made from any task or from lwIP callbacks, but not from IRQs. Calls into the lwIP RAW API (which is not thread safe) must be bracketed with cyw43_arch_lwip_begin and cyw43_arch_lwip_end. It is fine to bracket calls made from within lwIP callbacks too; you just don't have to.

NOTE

this wrapper library requires you to link FreeRTOS functionality with your application yourself.

• **pico_cyw43_arch_none** - If you do not need the TCP/IP stack but wish to use the on-board LED.

This wrapper library:

- Sets `CYW43_LWIP=0` to disable lwIP support in `pico_cyw43_arch` and `cyw43_driver`

4.4.4.2. Modules

cyw43_driver

Driver used for Pico W wireless.

4.4.4.3. Macros

- `#define cyw43_arch_lwip_check(void) cyw43_thread_lock_check()`

4.4.4.4. Functions

`int cyw43_arch_init (void)`

Initialize the CYW43 architecture.

`int cyw43_arch_init_with_country (uint32_t country)`

Initialize the CYW43 architecture for use in a specific country.

`void cyw43_arch_deinit (void)`

De-initialize the CYW43 architecture.

`async_context_t * cyw43_arch_async_context (void)`

Return the current async_context currently in use by the cyw43_arch code.

`void cyw43_arch_set_async_context (async_context_t *context)`

Set the async_context to be used by the cyw43_arch_init.

`async_context_t * cyw43_arch_init_default_async_context (void)`

Initialize the default async_context for the current cyw43_arch type.

`void cyw43_arch_poll (void)`

Perform any processing required by the `cyw43_driver` or the TCP/IP stack.

`void cyw43_arch_wait_for_work_until (absolute_time_t until)`

Sleep until there is cyw43_driver work to be done.

`uint32_t cyw43_arch_get_country_code (void)`

Return the country code used to initialize cyw43_arch.

`void cyw43_arch_enable_sta_mode (void)`

Enables Wi-Fi STA (Station) mode.

`void cyw43_arch_disable_sta_mode (void)`

Disables Wi-Fi STA (Station) mode.

`void cyw43_arch_enable_ap_mode (const char *ssid, const char *password, uint32_t auth)`

Enables Wi-Fi AP (Access point) mode.

`void cyw43_arch_disable_ap_mode (void)`

Disables Wi-Fi AP (Access point) mode.

`int cyw43_arch_wifi_connect_blocking (const char *ssid, const char *pw, uint32_t auth)`

Attempt to connect to a wireless access point, blocking until the network is joined or a failure is detected.

`int cyw43_arch_wifi_connect_bssid_blocking (const char *ssid, const uint8_t *bssid, const char *pw, uint32_t auth)`

Attempt to connect to a wireless access point specified by SSID and BSSID, blocking until the network is joined or a failure is detected.

`int cyw43_arch_wifi_connect_timeout_ms (const char *ssid, const char *pw, uint32_t auth, uint32_t timeout)`

Attempt to connect to a wireless access point, blocking until the network is joined, a failure is detected or a timeout occurs.

`int cyw43_arch_wifi_connect_bssid_timeout_ms (const char *ssid, const uint8_t *bssid, const char *pw, uint32_t auth, uint32_t timeout)`

Attempt to connect to a wireless access point specified by SSID and BSSID, blocking until the network is joined, a failure is detected or a timeout occurs.

`int cyw43_arch_wifi_connect_async (const char *ssid, const char *pw, uint32_t auth)`

Start attempting to connect to a wireless access point.

`int cyw43_arch_wifi_connect_bssid_async (const char *ssid, const uint8_t *bssid, const char *pw, uint32_t auth)`

Start attempting to connect to a wireless access point specified by SSID and BSSID.

`void cyw43_arch_gpio_put (uint wl_gpio, bool value)`

Set a GPIO pin on the wireless chip to a given value.

`bool cyw43_arch_gpio_get (uint wl_gpio)`

Read the value of a GPIO pin on the wireless chip.

`static void cyw43_arch_lwip_begin (void)`

Acquire any locks required to call into lwIP.

`static void cyw43_arch_lwip_end (void)`

Release any locks required for calling into lwIP.

`static int cyw43_arch_lwip_protect (int(*func)(void *param), void *param)`

sad Release any locks required for calling into lwIP

4.4.4.5. Macro Definition Documentation

4.4.4.5.1. cyw43_arch_lwip_check

```
#define cyw43_arch_lwip_check(void) cyw43_thread_lock_check()
```

Checks the caller has any locks required for calling into lwIP.

The lwIP API is not thread safe. You should surround calls into the lwIP API with calls to cyw43_arch_lwip_begin and this method. Note these calls are not necessary (but harmless) when you are calling back into the lwIP API from an lwIP callback.

This method will assert in debug mode, if the above conditions are not met (i.e. it is not safe to call into the lwIP API)

> **NOTE**
>
> as of SDK release 1.5.0, this is now equivalent to calling async_context_lock_check on the async_context associated with cyw43_arch and lwIP.

See also

cyw43_arch_lwip_begin

cyw43_arch_lwip_protect

async_context_lock_check

cyw43_arch_async_context

4.4.4.6. Function Documentation

4.4.4.6.1. cyw43_arch_async_context

`async_context_t * cyw43_arch_async_context (void)`

Return the current async_context currently in use by the cyw43_arch code.

Returns

the async_context.

4.4.4.6.2. cyw43_arch_deinit

`void cyw43_arch_deinit (void)`

De-initialize the CYW43 architecture.

This method de-initializes the `cyw43_driver` code and de-initializes the lwIP stack (if it was enabled at build time). Note this method should always be called from the same core (or RTOS task, depending on the environment) as cyw43_arch_init.

Additionally if the cyw43_arch is using its own async_context instance, then that instance is de-initialized.

4.4.4.6.3. cyw43_arch_disable_ap_mode

`void cyw43_arch_disable_ap_mode (void)`

Disables Wi-Fi AP (Access point) mode.

This Disbles the Wi-Fi in *Access Point* mode.

4.4.4.6.4. cyw43_arch_disable_sta_mode

`void cyw43_arch_disable_sta_mode (void)`

Disables Wi-Fi STA (Station) mode.

This disables the Wi-Fi in *Station* mode, disconnecting any active connection. You should subsequently check the status by calling cyw43_wifi_link_status.

4.4.4.6.5. cyw43_arch_enable_ap_mode

`void cyw43_arch_enable_ap_mode (const char * ssid, const char * password, uint32_t auth)`

Enables Wi-Fi AP (Access point) mode.

This enables the Wi-Fi in *Access Point* mode such that connections can be made to the device by other Wi-Fi clients

Parameters

`ssid`	the name for the access point
`password`	the password to use or NULL for no password.
`auth`	the authorization type to use when the password is enabled. Values are CYW43_AUTH_WPA_TKIP_PSK, CYW43_AUTH_WPA2_AES_PSK, or CYW43_AUTH_WPA2_MIXED_PSK (see CYW43_AUTH_)

4.4.4.6.6. cyw43_arch_enable_sta_mode

`void cyw43_arch_enable_sta_mode (void)`

Enables Wi-Fi STA (Station) mode.

This enables the Wi-Fi in *Station* mode such that connections can be made to other Wi-Fi Access Points

4.4.4.6.7. cyw43_arch_get_country_code

`uint32_t cyw43_arch_get_country_code (void)`

Return the country code used to initialize cyw43_arch.

Returns

the country code (see CYW43_COUNTRY_)

4.4.4.6.8. cyw43_arch_gpio_get

`bool cyw43_arch_gpio_get (uint wl_gpio)`

Read the value of a GPIO pin on the wireless chip.

> **NOTE**
>
> this method does not check for errors setting the GPIO. You can use the lower level cyw43_gpio_get instead if you wish to check for errors.

Parameters

`wl_gpio`	the GPIO number on the wireless chip

Returns

true if the GPIO is high, false otherwise

4.4.4.6.9. cyw43_arch_gpio_put

`void cyw43_arch_gpio_put (uint wl_gpio, bool value)`

Set a GPIO pin on the wireless chip to a given value.

> **NOTE**
>
> this method does not check for errors setting the GPIO. You can use the lower level cyw43_gpio_set instead if you wish to check for errors.

Parameters

`wl_gpio`	the GPIO number on the wireless chip
`value`	true to set the GPIO, false to clear it.

4.4.4.6.10. cyw43_arch_init

`int cyw43_arch_init (void)`

Initialize the CYW43 architecture.

This method initializes the `cyw43_driver` code and initializes the lwIP stack (if it was enabled at build time). This method must be called prior to using any other `pico_cyw43_arch`, `cyw43_driver` or lwIP functions.

> **NOTE**
>
> this method initializes wireless with a country code of `PICO_CYW43_ARCH_DEFAULT_COUNTRY_CODE` which defaults to `CYW43_COUNTRY_WORLDWIDE`. Worldwide settings may not give the best performance; consider setting PICO_CYW43_ARCH_DEFAULT_COUNTRY_CODE to a different value or calling cyw43_arch_init_with_country

By default this method initializes the cyw43_arch code's own async_context by calling cyw43_arch_init_default_async_context, however the user can specify use of their own async_context by calling cyw43_arch_set_async_context() before calling this method

Returns

0 if the initialization is successful, an error code otherwise

See also

pico_error_codes

4.4.4.6.11. cyw43_arch_init_default_async_context

`async_context_t * cyw43_arch_init_default_async_context (void)`

Initialize the default async_context for the current cyw43_arch type.

This method initializes and returns a pointer to the static async_context associated with cyw43_arch. This method is called by cyw43_arch_init automatically if a different async_context has not been set by cyw43_arch_set_async_context

Returns

the context or NULL if initialization failed.

4.4.4.6.12. cyw43_arch_init_with_country

`int cyw43_arch_init_with_country (uint32_t country)`

Initialize the CYW43 architecture for use in a specific country.

This method initializes the `cyw43_driver` code and initializes the lwIP stack (if it was enabled at build time). This method must be called prior to using any other `pico_cyw43_arch`, `cyw43_driver` or lwIP functions.

By default this method initializes the cyw43_arch code's own async_context by calling cyw43_arch_init_default_async_context, however the user can specify use of their own async_context by calling cyw43_arch_set_async_context() before calling this method

Parameters

`country`	the country code to use (see CYW43_COUNTRY_)

Returns

0 if the initialization is successful, an error code otherwise

See also

pico_error_codes

4.4.4.6.13. cyw43_arch_lwip_begin

`static void cyw43_arch_lwip_begin (void) [inline], [static]`

Acquire any locks required to call into lwIP.

The lwIP API is not thread safe. You should surround calls into the lwIP API with calls to this method and cyw43_arch_lwip_end. Note these calls are not necessary (but harmless) when you are calling back into the lwIP API from an lwIP callback. If you are using single-core polling only (pico_cyw43_arch_poll) then these calls are no-ops anyway it is good practice to call them anyway where they are necessary.

> **NOTE**
>
> as of SDK release 1.5.0, this is now equivalent to calling async_context_acquire_lock_blocking on the async_context associated with cyw43_arch and lwIP.

See also

cyw43_arch_lwip_end

cyw43_arch_lwip_protect

async_context_acquire_lock_blocking

cyw43_arch_async_context

4.4.4.6.14. cyw43_arch_lwip_end

```
static void cyw43_arch_lwip_end (void) [inline], [static]
```

Release any locks required for calling into lwIP.

The lwIP API is not thread safe. You should surround calls into the lwIP API with calls to cyw43_arch_lwip_begin and this method. Note these calls are not necessary (but harmless) when you are calling back into the lwIP API from an lwIP callback. If you are using single-core polling only (pico_cyw43_arch_poll) then these calls are no-ops anyway it is good practice to call them anyway where they are necessary.

> **NOTE**
>
> as of SDK release 1.5.0, this is now equivalent to calling async_context_release_lock on the async_context associated with cyw43_arch and lwIP.

See also

cyw43_arch_lwip_begin

cyw43_arch_lwip_protect

async_context_release_lock

cyw43_arch_async_context

4.4.4.6.15. cyw43_arch_lwip_protect

```
static int cyw43_arch_lwip_protect (int(*)(void *param) func, void * param) [inline], [static]
```

sad Release any locks required for calling into lwIP

The lwIP API is not thread safe. You can use this method to wrap a function with any locking required to call into the lwIP API. If you are using single-core polling only (pico_cyw43_arch_poll) then there are no locks to required, but it is still good practice to use this function.

Parameters

`func`	the function ta call with any required locks held
`param`	parameter to pass to `func`

Returns

the return value from `func`

See also

cyw43_arch_lwip_begin

cyw43_arch_lwip_end

4.4.4.6.16. cyw43_arch_poll

`void cyw43_arch_poll (void)`

Perform any processing required by the `cyw43_driver` or the TCP/IP stack.

This method must be called periodically from the main loop when using a *polling* style `pico_cyw43_arch` (e.g. `pico_cyw43_arch_lwip_poll`). It may be called in other styles, but it is unnecessary to do so.

4.4.4.6.17. cyw43_arch_set_async_context

`void cyw43_arch_set_async_context (async_context_t * context)`

Set the async_context to be used by the cyw43_arch_init.

NOTE

> This method must be called before calling cyw43_arch_init or cyw43_arch_init_with_country if you wish to use a custom async_context instance.

Parameters

`context`	the async_context to be used

4.4.4.6.18. cyw43_arch_wait_for_work_until

`void cyw43_arch_wait_for_work_until (absolute_time_t until)`

Sleep until there is cyw43_driver work to be done.

This method may be called by code that is waiting for an event to come from the cyw43_driver, and has no work to do, but would like to sleep without blocking any background work associated with the cyw43_driver.

Parameters

`until`	the time to wait until if there is no work to do.

4.4.4.6.19. cyw43_arch_wifi_connect_async

`int cyw43_arch_wifi_connect_async (const char * ssid, const char * pw, uint32_t auth)`

Start attempting to connect to a wireless access point.

This method tells the CYW43 driver to start connecting to an access point. You should subsequently check the status by calling cyw43_wifi_link_status.

Parameters

`ssid`	the network name to connect to
`pw`	the network password or NULL if there is no password required
`auth`	the authorization type to use when the password is enabled. Values are CYW43_AUTH_WPA_TKIP_PSK, CYW43_AUTH_WPA2_AES_PSK, or CYW43_AUTH_WPA2_MIXED_PSK (see CYW43_AUTH_)

Returns

0 if the scan was started successfully, an error code otherwise

See also

pico_error_codes

4.4.4.6.20. cyw43_arch_wifi_connect_blocking

`int cyw43_arch_wifi_connect_blocking (const char * ssid, const char * pw, uint32_t auth)`

Attempt to connect to a wireless access point, blocking until the network is joined or a failure is detected.

Parameters

`ssid`	the network name to connect to
`pw`	the network password or NULL if there is no password required
`auth`	the authorization type to use when the password is enabled. Values are CYW43_AUTH_WPA_TKIP_PSK, CYW43_AUTH_WPA2_AES_PSK, or CYW43_AUTH_WPA2_MIXED_PSK (see CYW43_AUTH_)

Returns

0 if the initialization is successful, an error code otherwise

See also

pico_error_codes

4.4.4.6.21. cyw43_arch_wifi_connect_bssid_async

`int cyw43_arch_wifi_connect_bssid_async (const char * ssid, const uint8_t * bssid, const char * pw, uint32_t auth)`

Start attempting to connect to a wireless access point specified by SSID and BSSID.

This method tells the CYW43 driver to start connecting to an access point. You should subsequently check the status by calling cyw43_wifi_link_status.

Parameters

`ssid`	the network name to connect to
`bssid`	the network BSSID to connect to or NULL if ignored
`pw`	the network password or NULL if there is no password required
`auth`	the authorization type to use when the password is enabled. Values are CYW43_AUTH_WPA_TKIP_PSK, CYW43_AUTH_WPA2_AES_PSK, or CYW43_AUTH_WPA2_MIXED_PSK (see CYW43_AUTH_)

Returns

0 if the scan was started successfully, an error code otherwise

See also

pico_error_codes

4.4.4.6.22. cyw43_arch_wifi_connect_bssid_blocking

`int cyw43_arch_wifi_connect_bssid_blocking (const char * ssid, const uint8_t * bssid, const char * pw, uint32_t auth)`

Attempt to connect to a wireless access point specified by SSID and BSSID, blocking until the network is joined or a failure is detected.

Parameters

`ssid`	the network name to connect to
`bssid`	the network BSSID to connect to or NULL if ignored
`pw`	the network password or NULL if there is no password required
`auth`	the authorization type to use when the password is enabled. Values are CYW43_AUTH_WPA_TKIP_PSK, CYW43_AUTH_WPA2_AES_PSK, or CYW43_AUTH_WPA2_MIXED_PSK (see CYW43_AUTH_)

Returns

0 if the initialization is successful, an error code otherwise

See also

pico_error_codes

4.4.4.6.23. cyw43_arch_wifi_connect_bssid_timeout_ms

`int cyw43_arch_wifi_connect_bssid_timeout_ms (const char * ssid, const uint8_t * bssid, const char * pw, uint32_t auth, uint32_t timeout)`

Attempt to connect to a wireless access point specified by SSID and BSSID, blocking until the network is joined, a failure is detected or a timeout occurs.

Parameters

`ssid`	the network name to connect to
`bssid`	the network BSSID to connect to or NULL if ignored
`pw`	the network password or NULL if there is no password required
`auth`	the authorization type to use when the password is enabled. Values are CYW43_AUTH_WPA_TKIP_PSK, CYW43_AUTH_WPA2_AES_PSK, or CYW43_AUTH_WPA2_MIXED_PSK (see CYW43_AUTH_)
`timeout`	how long to wait in milliseconds for a connection to succeed before giving up

Returns

0 if the initialization is successful, an error code otherwise

See also

pico_error_codes

4.4.4.6.24. cyw43_arch_wifi_connect_timeout_ms

`int cyw43_arch_wifi_connect_timeout_ms (const char * ssid, const char * pw, uint32_t auth, uint32_t timeout)`

Attempt to connect to a wireless access point, blocking until the network is joined, a failure is detected or a timeout occurs.

Parameters

`ssid`	the network name to connect to
`pw`	the network password or NULL if there is no password required
`auth`	the authorization type to use when the password is enabled. Values are CYW43_AUTH_WPA_TKIP_PSK, CYW43_AUTH_WPA2_AES_PSK, or CYW43_AUTH_WPA2_MIXED_PSK (see CYW43_AUTH_)
`timeout`	how long to wait in milliseconds for a connection to succeed before giving up

Returns

0 if the initialization is successful, an error code otherwise

See also

pico_error_codes

4.4.4.7. cyw43_driver

Driver used for Pico W wireless.

4.4.4.7.1. Modules

cyw43_ll

Low Level CYW43 driver interface.

4.4.4.7.2. Macros

- `#define CYW43_DEFAULT_PM (CYW43_PERFORMANCE_PM)`
- `#define CYW43_NONE_PM (cyw43_pm_value(CYW43_NO_POWERSAVE_MODE, 10, 0, 0, 0))`
- `#define CYW43_AGGRESSIVE_PM (cyw43_pm_value(CYW43_PM1_POWERSAVE_MODE, 10, 0, 0, 0))`
- `#define CYW43_PERFORMANCE_PM (cyw43_pm_value(CYW43_PM2_POWERSAVE_MODE, 200, 1, 1, 10))`
- `#define CYW43_COUNTRY(A, B, REV) ((unsigned char)(A) | ((unsigned char)(B) << 8) | ((REV) << 16))`

4.4.4.7.3. Typedefs

`typedef struct _cyw43_t cyw43_t`

4.4.4.7.4. Functions

`void cyw43_init (cyw43_t *self)`

Initialize the driver.

`void cyw43_deinit (cyw43_t *self)`

Shut the driver down.

`int cyw43_ioctl (cyw43_t *self, uint32_t cmd, size_t len, uint8_t *buf, uint32_t iface)`

Send an ioctl command to cyw43.

`int cyw43_send_ethernet (cyw43_t *self, int itf, size_t len, const void *buf, bool is_pbuf)`

Send a raw ethernet packet.

`int cyw43_wifi_pm (cyw43_t *self, uint32_t pm)`

Set the wifi power management mode.

`int cyw43_wifi_get_pm (cyw43_t *self, uint32_t *pm)`

Get the wifi power management mode.

`int cyw43_wifi_link_status (cyw43_t *self, int itf)`

Get the wifi link status.

`void cyw43_wifi_set_up (cyw43_t *self, int itf, bool up, uint32_t country)`

Set up and initialise wifi.

`int cyw43_wifi_get_mac (cyw43_t *self, int itf, uint8_t mac[6])`

Get the mac address of the device.

`int cyw43_wifi_update_multicast_filter (cyw43_t *self, uint8_t *addr, bool add)`

Add/remove multicast group address.

`int cyw43_wifi_scan (cyw43_t *self, cyw43_wifi_scan_options_t *opts, void *env, int(*result_cb)(void *, const cyw43_ev_scan_result_t *))`

Perform a wifi scan for wifi networks.

`static bool cyw43_wifi_scan_active (cyw43_t *self)`

Determine if a wifi scan is in progress.

`int cyw43_wifi_join (cyw43_t *self, size_t ssid_len, const uint8_t *ssid, size_t key_len, const uint8_t *key, uint32_t auth_type, const uint8_t *bssid, uint32_t channel)`

Connect or *join* a wifi network.

`int cyw43_wifi_leave (cyw43_t *self, int itf)`

Disassociate from a wifi network.

`int cyw43_wifi_get_rssi (cyw43_t *self, int32_t *rssi)`

Get the signal strength (RSSI) of the wifi network.

`int cyw43_wifi_get_bssid (cyw43_t *self, uint8_t bssid[6])`

Get the BSSID of the connected wifi network.

`static void cyw43_wifi_ap_get_ssid (cyw43_t *self, size_t *len, const uint8_t **buf)`

Get the ssid for the access point.

`static uint32_t cyw43_wifi_ap_get_auth (cyw43_t *self)`

Get the security authorisation used in AP mode.

`static void cyw43_wifi_ap_set_channel (cyw43_t *self, uint32_t channel)`

Set the the channel for the access point.

`static void cyw43_wifi_ap_set_ssid (cyw43_t *self, size_t len, const uint8_t *buf)`

Set the ssid for the access point.

`static void cyw43_wifi_ap_set_password (cyw43_t *self, size_t len, const uint8_t *buf)`

Set the password for the wifi access point.

`static void cyw43_wifi_ap_set_auth (cyw43_t *self, uint32_t auth)`

Set the security authorisation used in AP mode.

`void cyw43_wifi_ap_get_max_stas (cyw43_t *self, int *max_stas)`

Get the maximum number of devices (STAs) that can be associated with the wifi access point.

`void cyw43_wifi_ap_get_stas (cyw43_t *self, int *num_stas, uint8_t *macs)`

Get the number of devices (STAs) associated with the wifi access point.

`static bool cyw43_is_initialized (cyw43_t *self)`

Determines if the cyw43 driver been initialised.

`void cyw43_cb_tcpip_init (cyw43_t *self, int itf)`

Initialise the IP stack.

`void cyw43_cb_tcpip_deinit (cyw43_t *self, int itf)`

Deinitialise the IP stack.

`void cyw43_cb_tcpip_set_link_up (cyw43_t *self, int itf)`

Notify the IP stack that the link is up.

`void cyw43_cb_tcpip_set_link_down (cyw43_t *self, int itf)`

Notify the IP stack that the link is down.

`int cyw43_tcpip_link_status (cyw43_t *self, int itf)`

Get the link status.

`int cyw43_gpio_set (cyw43_t *self, int gpio, bool val)`

Set the value of the cyw43 gpio.

`int cyw43_gpio_get (cyw43_t *self, int gpio, bool *val)`

Get the value of the cyw43 gpio.

`static uint32_t cyw43_pm_value (uint8_t pm_mode, uint16_t pm2_sleep_ret_ms, uint8_t li_beacon_period, uint8_t li_dtim_period, uint8_t li_assoc)`

Return a power management value to pass to cyw43_wifi_pm.

4.4.4.7.5. Variables

`cyw43_t cyw43_state`

`void(* cyw43_poll)(void)`

`uint32_t cyw43_sleep`

4.4.4.7.6. CYW43 driver version as components

Current version of the CYW43 driver as major/minor/micro components

CYW43_VERSION_MAJOR

```
#define CYW43_VERSION_MAJOR 1
```

CYW43_VERSION_MINOR

```
#define CYW43_VERSION_MINOR 0
```

CYW43_VERSION_MICRO

```
#define CYW43_VERSION_MICRO 3
```

4.4.4.7.7. CYW43 driver version

Combined CYW43 driver version as a 32-bit number

CYW43_VERSION

```
#define CYW43_VERSION (CYW43_VERSION_MAJOR << 16 | CYW43_VERSION_MINOR << 8 | CYW43_VERSION_MICRO)
```

4.4.4.7.8. Trace flags

CYW43_TRACE_ASYNC_EV

```
#define CYW43_TRACE_ASYNC_EV (0x0001)
```

CYW43_TRACE_ETH_TX

```
#define CYW43_TRACE_ETH_TX (0x0002)
```

CYW43_TRACE_ETH_RX

```
#define CYW43_TRACE_ETH_RX (0x0004)
```

CYW43_TRACE_ETH_FULL

```
#define CYW43_TRACE_ETH_FULL (0x0008)
```

CYW43_TRACE_MAC

```
#define CYW43_TRACE_MAC (0x0010)
```

4.4.4.7.9. Link status

See also

status_name() to get a user readable name of the status for debug

cyw43_wifi_link_status() to get the wifi status

cyw43_tcpip_link_status() to get the overall link status

CYW43_LINK_DOWN

```
#define CYW43_LINK_DOWN (0)
```

link is down

CYW43_LINK_JOIN

```
#define CYW43_LINK_JOIN (1)
```

Connected to wifi.

CYW43_LINK_NOIP

```
#define CYW43_LINK_NOIP (2)
```

Connected to wifi, but no IP address.

CYW43_LINK_UP

```
#define CYW43_LINK_UP (3)
```

Connected to wifi with an IP address.

CYW43_LINK_FAIL

```
#define CYW43_LINK_FAIL (-1)
```

Connection failed.

CYW43_LINK_NONET

```
#define CYW43_LINK_NONET (-2)
```

No matching SSID found (could be out of range, or down)

CYW43_LINK_BADAUTH

```
#define CYW43_LINK_BADAUTH (-3)
```

Authenticatation failure

4.4.4.7.10. Country codes

CYW43_COUNTRY_WORLDWIDE

```
#define CYW43_COUNTRY_WORLDWIDE CYW43_COUNTRY('X', 'X', 0)
```

CYW43_COUNTRY_AUSTRALIA

```
#define CYW43_COUNTRY_AUSTRALIA CYW43_COUNTRY('A', 'U', 0)
```

CYW43_COUNTRY_AUSTRIA

```
#define CYW43_COUNTRY_AUSTRIA CYW43_COUNTRY('A', 'T', 0)
```

CYW43_COUNTRY_BELGIUM

```
#define CYW43_COUNTRY_BELGIUM CYW43_COUNTRY('B', 'E', 0)
```

CYW43_COUNTRY_BRAZIL

```
#define CYW43_COUNTRY_BRAZIL CYW43_COUNTRY('B', 'R', 0)
```

CYW43_COUNTRY_CANADA

```
#define CYW43_COUNTRY_CANADA CYW43_COUNTRY('C', 'A', 0)
```

CYW43_COUNTRY_CHILE

```
#define CYW43_COUNTRY_CHILE CYW43_COUNTRY('C', 'L', 0)
```

CYW43_COUNTRY_CHINA

```
#define CYW43_COUNTRY_CHINA CYW43_COUNTRY('C', 'N', 0)
```

CYW43_COUNTRY_COLOMBIA

```
#define CYW43_COUNTRY_COLOMBIA CYW43_COUNTRY('C', 'O', 0)
```

CYW43_COUNTRY_CZECH_REPUBLIC

```
#define CYW43_COUNTRY_CZECH_REPUBLIC CYW43_COUNTRY('C', 'Z', 0)
```

CYW43_COUNTRY_DENMARK

```
#define CYW43_COUNTRY_DENMARK CYW43_COUNTRY('D', 'K', 0)
```

CYW43_COUNTRY_ESTONIA

```
#define CYW43_COUNTRY_ESTONIA CYW43_COUNTRY('E', 'E', 0)
```

CYW43_COUNTRY_FINLAND

```
#define CYW43_COUNTRY_FINLAND CYW43_COUNTRY('F', 'I', 0)
```

CYW43_COUNTRY_FRANCE

```
#define CYW43_COUNTRY_FRANCE CYW43_COUNTRY('F', 'R', 0)
```

CYW43_COUNTRY_GERMANY

```
#define CYW43_COUNTRY_GERMANY CYW43_COUNTRY('D', 'E', 0)
```

CYW43_COUNTRY_GREECE

```
#define CYW43_COUNTRY_GREECE CYW43_COUNTRY('G', 'R', 0)
```

CYW43_COUNTRY_HONG_KONG

```
#define CYW43_COUNTRY_HONG_KONG CYW43_COUNTRY('H', 'K', 0)
```

CYW43_COUNTRY_HUNGARY

```
#define CYW43_COUNTRY_HUNGARY CYW43_COUNTRY('H', 'U', 0)
```

CYW43_COUNTRY_ICELAND

```
#define CYW43_COUNTRY_ICELAND CYW43_COUNTRY('I', 'S', 0)
```

CYW43_COUNTRY_INDIA

```
#define CYW43_COUNTRY_INDIA CYW43_COUNTRY('I', 'N', 0)
```

CYW43_COUNTRY_ISRAEL

```
#define CYW43_COUNTRY_ISRAEL CYW43_COUNTRY('I', 'L', 0)
```

CYW43_COUNTRY_ITALY

```
#define CYW43_COUNTRY_ITALY CYW43_COUNTRY('I', 'T', 0)
```

CYW43_COUNTRY_JAPAN

```
#define CYW43_COUNTRY_JAPAN CYW43_COUNTRY('J', 'P', 0)
```

CYW43_COUNTRY_KENYA

```
#define CYW43_COUNTRY_KENYA CYW43_COUNTRY('K', 'E', 0)
```

CYW43_COUNTRY_LATVIA

```
#define CYW43_COUNTRY_LATVIA CYW43_COUNTRY('L', 'V', 0)
```

CYW43_COUNTRY_LIECHTENSTEIN

```
#define CYW43_COUNTRY_LIECHTENSTEIN CYW43_COUNTRY('L', 'I', 0)
```

CYW43_COUNTRY_LITHUANIA

```
#define CYW43_COUNTRY_LITHUANIA CYW43_COUNTRY('L', 'T', 0)
```

CYW43_COUNTRY_LUXEMBOURG

```
#define CYW43_COUNTRY_LUXEMBOURG CYW43_COUNTRY('L', 'U', 0)
```

CYW43_COUNTRY_MALAYSIA

```
#define CYW43_COUNTRY_MALAYSIA CYW43_COUNTRY('M', 'Y', 0)
```

CYW43_COUNTRY_MALTA

```
#define CYW43_COUNTRY_MALTA CYW43_COUNTRY('M', 'T', 0)
```

CYW43_COUNTRY_MEXICO

```
#define CYW43_COUNTRY_MEXICO CYW43_COUNTRY('M', 'X', 0)
```

CYW43_COUNTRY_NETHERLANDS

```
#define CYW43_COUNTRY_NETHERLANDS CYW43_COUNTRY('N', 'L', 0)
```

CYW43_COUNTRY_NEW_ZEALAND

```
#define CYW43_COUNTRY_NEW_ZEALAND CYW43_COUNTRY('N', 'Z', 0)
```

CYW43_COUNTRY_NIGERIA

```
#define CYW43_COUNTRY_NIGERIA CYW43_COUNTRY('N', 'G', 0)
```

CYW43_COUNTRY_NORWAY

```
#define CYW43_COUNTRY_NORWAY CYW43_COUNTRY('N', 'O', 0)
```

CYW43_COUNTRY_PERU

```
#define CYW43_COUNTRY_PERU CYW43_COUNTRY('P', 'E', 0)
```

CYW43_COUNTRY_PHILIPPINES

```
#define CYW43_COUNTRY_PHILIPPINES CYW43_COUNTRY('P', 'H', 0)
```

CYW43_COUNTRY_POLAND

```
#define CYW43_COUNTRY_POLAND CYW43_COUNTRY('P', 'L', 0)
```

CYW43_COUNTRY_PORTUGAL

```
#define CYW43_COUNTRY_PORTUGAL CYW43_COUNTRY('P', 'T', 0)
```

CYW43_COUNTRY_SINGAPORE

```
#define CYW43_COUNTRY_SINGAPORE CYW43_COUNTRY('S', 'G', 0)
```

CYW43_COUNTRY_SLOVAKIA

```
#define CYW43_COUNTRY_SLOVAKIA CYW43_COUNTRY('S', 'K', 0)
```

CYW43_COUNTRY_SLOVENIA

```
#define CYW43_COUNTRY_SLOVENIA CYW43_COUNTRY('S', 'I', 0)
```

CYW43_COUNTRY_SOUTH_AFRICA

```
#define CYW43_COUNTRY_SOUTH_AFRICA CYW43_COUNTRY('Z', 'A', 0)
```

CYW43_COUNTRY_SOUTH_KOREA

```
#define CYW43_COUNTRY_SOUTH_KOREA CYW43_COUNTRY('K', 'R', 0)
```

CYW43_COUNTRY_SPAIN

```
#define CYW43_COUNTRY_SPAIN CYW43_COUNTRY('E', 'S', 0)
```

CYW43_COUNTRY_SWEDEN

```
#define CYW43_COUNTRY_SWEDEN CYW43_COUNTRY('S', 'E', 0)
```

CYW43_COUNTRY_SWITZERLAND

```
#define CYW43_COUNTRY_SWITZERLAND CYW43_COUNTRY('C', 'H', 0)
```

CYW43_COUNTRY_TAIWAN

```
#define CYW43_COUNTRY_TAIWAN CYW43_COUNTRY('T', 'W', 0)
```

CYW43_COUNTRY_THAILAND

```
#define CYW43_COUNTRY_THAILAND CYW43_COUNTRY('T', 'H', 0)
```

CYW43_COUNTRY_TURKEY

```
#define CYW43_COUNTRY_TURKEY CYW43_COUNTRY('T', 'R', 0)
```

CYW43_COUNTRY_UK

```
#define CYW43_COUNTRY_UK CYW43_COUNTRY('G', 'B', 0)
```

CYW43_COUNTRY_USA

```
#define CYW43_COUNTRY_USA CYW43_COUNTRY('U', 'S', 0)
```

4.4.4.7.11. Macro Definition Documentation

CYW43_DEFAULT_PM

```
#define CYW43_DEFAULT_PM (CYW43_PERFORMANCE_PM)
```

Default power management mode.

CYW43_NONE_PM

```
#define CYW43_NONE_PM (cyw43_pm_value(CYW43_NO_POWERSAVE_MODE, 10, 0, 0, 0))
```

No power management.

CYW43_AGGRESSIVE_PM

```
#define CYW43_AGGRESSIVE_PM (cyw43_pm_value(CYW43_PM1_POWERSAVE_MODE, 10, 0, 0, 0))
```

Aggressive power management mode for optimal power usage at the cost of performance.

CYW43_PERFORMANCE_PM

```
#define CYW43_PERFORMANCE_PM (cyw43_pm_value(CYW43_PM2_POWERSAVE_MODE, 200, 1, 1, 10))
```

Performance power management mode where more power is used to increase performance.

CYW43_COUNTRY

```
#define CYW43_COUNTRY(A, B, REV) ((unsigned char)(A) | ((unsigned char)(B) << 8) | ((REV) << 16))
```

create a country code from the two character country and revision number

4.4.4.7.12. Typedef Documentation

cyw43_t

```
typedef struct _cyw43_t cyw43_t
```

4.4.4.7.13. Function Documentation

cyw43_cb_tcpip_deinit

```
void cyw43_cb_tcpip_deinit (cyw43_t * self, int itf)
```

Deinitialise the IP stack.

This method must be provided by the network stack interface It is called to close the IP stack and free resources.

Parameters

`self`	the driver state object. This should always be `&cyw43_state`
`itf`	the interface used, either CYW43_ITF_STA or CYW43_ITF_AP

cyw43_cb_tcpip_init

```
void cyw43_cb_tcpip_init (cyw43_t * self, int itf)
```

Initialise the IP stack.

This method must be provided by the network stack interface It is called to initialise the IP stack.

Parameters

`self`	the driver state object. This should always be `&cyw43_state`
`itf`	the interface used, either CYW43_ITF_STA or CYW43_ITF_AP

cyw43_cb_tcpip_set_link_down

```
void cyw43_cb_tcpip_set_link_down (cyw43_t * self, int itf)
```

Notify the IP stack that the link is down.

This method must be provided by the network stack interface It is called to notify the IP stack that the link is down.

Parameters

`self`	the driver state object. This should always be `&cyw43_state`

`itf`	the interface used, either CYW43_ITF_STA or CYW43_ITF_AP

cyw43_cb_tcpip_set_link_up

`void cyw43_cb_tcpip_set_link_up (cyw43_t * self, int itf)`

Notify the IP stack that the link is up.

This method must be provided by the network stack interface It is called to notify the IP stack that the link is up. This can, for example be used to request an IP address via DHCP.

Parameters

`self`	the driver state object. This should always be `&cyw43_state`
`itf`	the interface used, either CYW43_ITF_STA or CYW43_ITF_AP

cyw43_deinit

`void cyw43_deinit (cyw43_t * self)`

Shut the driver down.

This method will close the network interfaces, and free up resources

Parameters

`self`	the driver state object. This should always be `&cyw43_state`

cyw43_gpio_get

`int cyw43_gpio_get (cyw43_t * self, int gpio, bool * val)`

Get the value of the cyw43 gpio.

Get the value of a cyw43 gpio.

> **NOTE**
>
> Check the datasheet for the number and purpose of the cyw43 gpios.

Parameters

`self`	the driver state object. This should always be `&cyw43_state`
`gpio`	number of the gpio to get
`val`	Returns the value of the gpio

Returns

0 on success

cyw43_gpio_set

`int cyw43_gpio_set (cyw43_t * self, int gpio, bool val)`

Set the value of the cyw43 gpio.

Set the value of a cyw43 gpio.

> **NOTE**
>
> Check the datasheet for the number and purpose of the cyw43 gpios.

Parameters

`self`	the driver state object. This should always be `&cyw43_state`
`gpio`	number of the gpio to set

`val` value for the gpio

Returns

0 on success

cyw43_init

`void cyw43_init (cyw43_t * self)`

Initialize the driver.

This method must be called before using the driver

Parameters

`self` the driver state object. This should always be `&cyw43_state`

cyw43_ioctl

`int cyw43_ioctl (cyw43_t * self, uint32_t cmd, size_t len, uint8_t * buf, uint32_t iface)`

Send an ioctl command to cyw43.

This method sends a command to cyw43.

Parameters

`self` the driver state object. This should always be `&cyw43_state`

`cmd` the command to send

`len` the amount of data to send with the command

`buf` a buffer containing the data to send

`iface` the interface to use, either CYW43_ITF_STA or CYW43_ITF_AP

Returns

0 on success

cyw43_is_initialized

`static bool cyw43_is_initialized (cyw43_t * self) [inline], [static]`

Determines if the cyw43 driver been initialised.

Returns true if the cyw43 driver has been initialised with a call to cyw43_init

Parameters

`self` the driver state object. This should always be `&cyw43_state`

Returns

True if the cyw43 driver has been initialised

cyw43_pm_value

`static uint32_t cyw43_pm_value (uint8_t pm_mode, uint16_t pm2_sleep_ret_ms, uint8_t li_beacon_period, uint8_t li_dtim_period, uint8_t li_assoc) [inline], [static]`

Return a power management value to pass to cyw43_wifi_pm.

Generate the power management (PM) value to pass to cyw43_wifi_pm

pm_mode	Meaning
CYW43_NO_POWERSAVE_MODE	No power saving
CYW43_PM1_POWERSAVE_MODE	Aggressive power saving which reduces wifi throughput

pm_mode	Meaning
CYW43_PM2_POWERSAVE_MODE	Power saving with High throughput (preferred). Saves power when there is no wifi activity for some time.

See also

CYW43_DEFAULT_PM

CYW43_NONE_PM

CYW43_AGGRESSIVE_PM

CYW43_PERFORMANCE_PM

Parameters

`pm_mode`	Power management mode
`pm2_sleep_ret_ms`	The maximum time to wait before going back to sleep for CYW43_PM2_POWERSAVE_MODE mode. Value measured in milliseconds and must be between 10 and 2000ms and divisible by 10
`li_beacon_period`	Wake period is measured in beacon periods
`li_dtim_period`	Wake interval measured in DTIMs. If this is set to 0, the wake interval is measured in beacon periods
`li_assoc`	Wake interval sent to the access point

cyw43_send_ethernet

```
int cyw43_send_ethernet (cyw43_t * self, int itf, size_t len, const void * buf, bool is_pbuf)
```

Send a raw ethernet packet.

This method sends a raw ethernet packet.

Parameters

`self`	the driver state object. This should always be `&cyw43_state`
`itf`	interface to use, either CYW43_ITF_STA or CYW43_ITF_AP
`len`	the amount of data to send
`buf`	the data to send
`is_pbuf`	true if buf points to an lwip struct pbuf

Returns

0 on success

cyw43_tcpip_link_status

```
int cyw43_tcpip_link_status (cyw43_t * self, int itf)
```

Get the link status.

Returns the status of the link which is a superset of the wifi link status returned by cyw43_wifi_link_status

> **NOTE**
>
> If the link status is negative it indicates an error

link status	Meaning
CYW43_LINK_DOWN	Wifi down
CYW43_LINK_JOIN	Connected to wifi
CYW43_LINK_NOIP	Connected to wifi, but no IP address
CYW43_LINK_UP	Connect to wifi with an IP address
CYW43_LINK_FAIL	Connection failed
CYW43_LINK_NONET	No matching SSID found (could be out of range, or down)
CYW43_LINK_BADAUTH	Authenticatation failure

Parameters

`self` the driver state object. This should always be `&cyw43_state`

`itf` the interface for which to return the link status, should be CYW43_ITF_STA or CYW43_ITF_AP

Returns

A value representing the link status

cyw43_wifi_ap_get_auth

`static uint32_t cyw43_wifi_ap_get_auth (cyw43_t * self) [inline], [static]`

Get the security authorisation used in AP mode.

For access point (AP) mode, this method can be used to get the security authorisation mode.

Parameters

`self` the driver state object. This should always be `&cyw43_state`

Returns

the current security authorisation mode for the access point

cyw43_wifi_ap_get_max_stas

`void cyw43_wifi_ap_get_max_stas (cyw43_t * self, int * max_stas)`

Get the maximum number of devices (STAs) that can be associated with the wifi access point.

For access point (AP) mode, this method can be used to get the maximum number of devices that can be connected to the wifi access point.

Parameters

`self` the driver state object. This should always be `&cyw43_state`

`max_stas` Returns the maximum number of devices (STAs) that can be connected to the access point (set to 0 on error)

cyw43_wifi_ap_get_ssid

`static void cyw43_wifi_ap_get_ssid (cyw43_t * self, size_t * len, const uint8_t ** buf) [inline], [static]`

Get the ssid for the access point.

For access point (AP) mode, this method can be used to get the SSID name of the wifi access point.

Parameters

`self` the driver state object. This should always be `&cyw43_state`

`len` Returns the length of the AP SSID name

`buf` Returns a pointer to an internal buffer containing the AP SSID name

cyw43_wifi_ap_get_stas

```
void cyw43_wifi_ap_get_stas (cyw43_t * self, int * num_stas, uint8_t * macs)
```

Get the number of devices (STAs) associated with the wifi access point.

For access point (AP) mode, this method can be used to get the number of devices and mac addresses of devices connected to the wifi access point.

Parameters

`self` the driver state object. This should always be `&cyw43_state`

`num_stas` Caller must provide the number of MACs that will fit in the macs buffer; The supplied buffer should have enough room for 6 bytes per MAC address. Returns the number of devices (STA) connected to the access point.

`macs` Returns up to num_stas MAC addresses of devices (STA) connected to the access point. Call cyw43_wifi_ap_get_max_stas to determine how many mac addresses can be returned.

cyw43_wifi_ap_set_auth

```
static void cyw43_wifi_ap_set_auth (cyw43_t * self, uint32_t auth) [inline], [static]
```

Set the security authorisation used in AP mode.

For access point (AP) mode, this method can be used to set how access to the access point is authorised.

Auth mode	Meaning
CYW43_AUTH_OPEN	Use an open access point with no authorisation required
CYW43_AUTH_WPA_TKIP_PSK	Use WPA authorisation
CYW43_AUTH_WPA2_AES_PSK	Use WPA2 (preferred)
CYW43_AUTH_WPA2_MIXED_PSK	Use WPA2/WPA mixed (currently treated the same as CYW43_AUTH_WPA2_AES_PSK)

Parameters

`self` the driver state object. This should always be `&cyw43_state`

`auth` Auth mode for the access point

cyw43_wifi_ap_set_channel

```
static void cyw43_wifi_ap_set_channel (cyw43_t * self, uint32_t channel) [inline], [static]
```

Set the the channel for the access point.

For access point (AP) mode, this method can be used to set the channel used for the wifi access point.

Parameters

`self` the driver state object. This should always be `&cyw43_state`

`channel` Wifi channel to use for the wifi access point

cyw43_wifi_ap_set_password

```
static void cyw43_wifi_ap_set_password (cyw43_t * self, size_t len, const uint8_t * buf) [inline], [static]
```

Set the password for the wifi access point.

For access point (AP) mode, this method can be used to set the password for the wifi access point.

Parameters

`self`	the driver state object. This should always be `&cyw43_state`
`len`	The length of the AP password
`buf`	A buffer containing the AP password

cyw43_wifi_ap_set_ssid

`static void cyw43_wifi_ap_set_ssid (cyw43_t * self, size_t len, const uint8_t * buf) [inline], [static]`

Set the ssid for the access point.

For access point (AP) mode, this method can be used to set the SSID name of the wifi access point.

Parameters

`self`	the driver state object. This should always be `&cyw43_state`
`len`	The length of the AP SSID name
`buf`	A buffer containing the AP SSID name

cyw43_wifi_get_bssid

`int cyw43_wifi_get_bssid (cyw43_t * self, uint8_t bssid)`

Get the BSSID of the connected wifi network.

Parameters

`self`	the driver state object. This should always be `&cyw43_state`
`bssid`	a buffer to receive the BSSID

Returns

0 on success

cyw43_wifi_get_mac

`int cyw43_wifi_get_mac (cyw43_t * self, int itf, uint8_t mac)`

Get the mac address of the device.

This method returns the mac address of the interface.

Parameters

`self`	the driver state object. This should always be `&cyw43_state`
`itf`	the interface to use, either CYW43_ITF_STA or CYW43_ITF_AP
`mac`	a buffer to receive the mac address

Returns

0 on success

cyw43_wifi_get_pm

`int cyw43_wifi_get_pm (cyw43_t * self, uint32_t * pm)`

Get the wifi power management mode.

This method gets the power management mode used by cyw43. The value is expressed as an unsigned integer 0x00adbrrm where, m = pm_mode Power management mode rr = pm2_sleep_ret (in units of 10ms) b = li_beacon_period d = li_dtim_period a = li_assoc

See also

cyw43_pm_value for an explanation of these values This should be called after cyw43_wifi_set_up

Parameters

`self`	the driver state object. This should always be `&cyw43_state`
`pm`	Power management value

Returns

0 on success

cyw43_wifi_get_rssi

`int cyw43_wifi_get_rssi (cyw43_t * self, int32_t * rssi)`

Get the signal strength (RSSI) of the wifi network.

For STA (client) mode, returns the signal strength or RSSI of the wifi network. An RSSI value of zero is returned if you call this function before a network is connected.

Parameters

`self`	the driver state object. This should always be `&cyw43_state`
`rssi`	a pointer to which the returned RSSI value is stored.

Returns

0 on success

cyw43_wifi_join

`int cyw43_wifi_join (cyw43_t * self, size_t ssid_len, const uint8_t * ssid, size_t key_len, const uint8_t * key, uint32_t auth_type, const uint8_t * bssid, uint32_t channel)`

Connect or *join* a wifi network.

Connect to a wifi network in STA (client) mode After success is returned, periodically call cyw43_wifi_link_status or cyw43_tcpip_link_status, to query the status of the link. It can take a many seconds to connect to fully join a network.

NOTE

> Call cyw43_wifi_leave to disassociate from a wifi network.

Parameters

`self`	the driver state object. This should always be `&cyw43_state`
`ssid_len`	the length of the wifi network name
`ssid`	A buffer containing the wifi network name
`key_len`	The length of the wifi *password*
`key`	A buffer containing the wifi *password*
`auth_type`	Auth type,

See also

CYW43_AUTH_

Parameters

`bssid`	the mac address of the access point to connect to. This can be NULL.
`channel`	Used to set the band of the connection. This is only used if bssid is non NULL.

Returns

0 on success

cyw43_wifi_leave

`int cyw43_wifi_leave (cyw43_t * self, int itf)`

Disassociate from a wifi network.

This method disassociates from a wifi network.

Parameters

- `self` the driver state object. This should always be `&cyw43_state`
- `itf` The interface to disconnect, either CYW43_ITF_STA or CYW43_ITF_AP

Returns

0 on success

cyw43_wifi_link_status

`int cyw43_wifi_link_status (cyw43_t * self, int itf)`

Get the wifi link status.

Returns the status of the wifi link.

link status	Meaning
CYW43_LINK_DOWN	Wifi down
CYW43_LINK_JOIN	Connected to wifi
CYW43_LINK_FAIL	Connection failed
CYW43_LINK_NONET	No matching SSID found (could be out of range, or down)
CYW43_LINK_BADAUTH	Authenticatation failure

NOTE

If the link status is negative it indicates an error The wifi link status for the interface CYW43_ITF_AP is always CYW43_LINK_DOWN

Parameters

- `self` the driver state object. This should always be `&cyw43_state`
- `itf` the interface to use, should be CYW43_ITF_STA or CYW43_ITF_AP

Returns

A integer value representing the link status

cyw43_wifi_pm

`int cyw43_wifi_pm (cyw43_t * self, uint32_t pm)`

Set the wifi power management mode.

This method sets the power management mode used by cyw43. This should be called after cyw43_wifi_set_up

See also

cyw43_pm_value

CYW43_DEFAULT_PM

CYW43_NONE_PM

CYW43_AGGRESSIVE_PM

CYW43_PERFORMANCE_PM

Parameters

`self`	the driver state object. This should always be `&cyw43_state`
`pm`	Power management value

Returns

0 on success

cyw43_wifi_scan

`int cyw43_wifi_scan (cyw43_t * self, cyw43_wifi_scan_options_t * opts, void * env, int(*)(void *, const cyw43_ev_scan_result_t *) result_cb)`

Perform a wifi scan for wifi networks.

Start a scan for wifi networks. Results are returned via the callback.

> **NOTE**
>
> The scan is complete when cyw43_wifi_scan_active return false

Parameters

`self`	the driver state object. This should always be `&cyw43_state`
`opts`	An instance of cyw43_wifi_scan_options_t. Values in here are currently ignored.
`env`	Pointer passed back in the callback
`result_cb`	Callback for wifi scan results, see cyw43_ev_scan_result_t

Returns

0 on success

cyw43_wifi_scan_active

`static bool cyw43_wifi_scan_active (cyw43_t * self) [inline], [static]`

Determine if a wifi scan is in progress.

This method tells you if the scan is still in progress

Parameters

`self`	the driver state object. This should always be `&cyw43_state`

Returns

true if a wifi scan is in progress

cyw43_wifi_set_up

`void cyw43_wifi_set_up (cyw43_t * self, int itf, bool up, uint32_t country)`

Set up and initialise wifi.

This method turns on wifi and sets the country for regulation purposes. The power management mode is initialised to CYW43_DEFAULT_PM For CYW43_ITF_AP, the access point is enabled. For CYW43_ITF_STA, the TCP/IP stack is reinitialised

Parameters

`self`	the driver state object. This should always be `&cyw43_state`
`itf`	the interface to use either CYW43_ITF_STA or CYW43_ITF_AP
`up`	true to enable the link. Set to false to disable AP mode. Setting the *up* parameter to false for CYW43_ITF_STA is ignored.
`country`	the country code, see CYW43_COUNTRY_

cyw43_wifi_update_multicast_filter

`int cyw43_wifi_update_multicast_filter (cyw43_t * self, uint8_t * addr, bool add)`

Add/remove multicast group address.

This method adds/removes an address from the multicast filter, allowing frames sent to this group to be received

Parameters

`self`	the driver state object. This should always be `&cyw43_state`
`addr`	a buffer containing a group mac address
`add`	true to add the address, false to remove it

Returns

0 on success

4.4.4.7.14. Variable Documentation

cyw43_state

`cyw43_t cyw43_state`

cyw43_poll

`void(* cyw43_poll) (void)`

cyw43_sleep

`uint32_t cyw43_sleep`

4.4.4.7.15. cyw43_ll

Low Level CYW43 driver interface.

Macros

- `#define CYW43_IOCTL_GET_SSID (0x32)`
- `#define CYW43_IOCTL_GET_CHANNEL (0x3a)`
- `#define CYW43_IOCTL_SET_DISASSOC (0x69)`
- `#define CYW43_IOCTL_GET_ANTDIV (0x7e)`
- `#define CYW43_IOCTL_SET_ANTDIV (0x81)`
- `#define CYW43_IOCTL_SET_MONITOR (0xd9)`
- `#define CYW43_IOCTL_GET_RSSI (0xfe)`
- `#define CYW43_IOCTL_GET_VAR (0x20c)`
- `#define CYW43_IOCTL_SET_VAR (0x20f)`
- `#define CYW43_EV_SET_SSID (0)`
- `#define CYW43_EV_JOIN (1)`
- `#define CYW43_EV_AUTH (3)`
- `#define CYW43_EV_DEAUTH (5)`
- `#define CYW43_EV_DEAUTH_IND (6)`
- `#define CYW43_EV_ASSOC (7)`

- #define CYW43_EV_DISASSOC (11)
- #define CYW43_EV_DISASSOC_IND (12)
- #define CYW43_EV_LINK (16)
- #define CYW43_EV_PRUNE (23)
- #define CYW43_EV_PSK_SUP (46)
- #define CYW43_EV_ICV_ERROR (49)
- #define CYW43_EV_ESCAN_RESULT (69)
- #define CYW43_EV_CSA_COMPLETE_IND (80)
- #define CYW43_EV_ASSOC_REQ_IE (87)
- #define CYW43_EV_ASSOC_RESP_IE (88)
- #define CYW43_STATUS_SUCCESS (0)
- #define CYW43_STATUS_FAIL (1)
- #define CYW43_STATUS_TIMEOUT (2)
- #define CYW43_STATUS_NO_NETWORKS (3)
- #define CYW43_STATUS_ABORT (4)
- #define CYW43_STATUS_NO_ACK (5)
- #define CYW43_STATUS_UNSOLICITED (6)
- #define CYW43_STATUS_ATTEMPT (7)
- #define CYW43_STATUS_PARTIAL (8)
- #define CYW43_STATUS_NEWSCAN (9)
- #define CYW43_STATUS_NEWASSOC (10)
- #define CYW43_SUP_DISCONNECTED (0)
- #define CYW43_SUP_CONNECTING (1)
- #define CYW43_SUP_IDREQUIRED (2)
- #define CYW43_SUP_AUTHENTICATING (3)
- #define CYW43_SUP_AUTHENTICATED (4)
- #define CYW43_SUP_KEYXCHANGE (5)
- #define CYW43_SUP_KEYED (6)
- #define CYW43_SUP_TIMEOUT (7)
- #define CYW43_SUP_LAST_BASIC_STATE (8)
- #define CYW43_SUP_KEYXCHANGE_WAIT_M1 CYW43_SUP_AUTHENTICATED
- #define CYW43_SUP_KEYXCHANGE_PREP_M2 CYW43_SUP_KEYXCHANGE
- #define CYW43_SUP_KEYXCHANGE_WAIT_M3 CYW43_SUP_LAST_BASIC_STATE
- #define CYW43_SUP_KEYXCHANGE_PREP_M4 (9)
- #define CYW43_SUP_KEYXCHANGE_WAIT_G1 (10)
- #define CYW43_SUP_KEYXCHANGE_PREP_G2 (11)
- #define CYW43_REASON_INITIAL_ASSOC (0)

- #define CYW43_REASON_LOW_RSSI (1)
- #define CYW43_REASON_DEAUTH (2)
- #define CYW43_REASON_DISASSOC (3)
- #define CYW43_REASON_BCNS_LOST (4)
- #define CYW43_REASON_FAST_ROAM_FAILED (5)
- #define CYW43_REASON_DIRECTED_ROAM (6)
- #define CYW43_REASON_TSPEC_REJECTED (7)
- #define CYW43_REASON_BETTER_AP (8)
- #define CYW43_REASON_PRUNE_ENCR_MISMATCH (1)
- #define CYW43_REASON_PRUNE_BCAST_BSSID (2)
- #define CYW43_REASON_PRUNE_MAC_DENY (3)
- #define CYW43_REASON_PRUNE_MAC_NA (4)
- #define CYW43_REASON_PRUNE_REG_PASSV (5)
- #define CYW43_REASON_PRUNE_SPCT_MGMT (6)
- #define CYW43_REASON_PRUNE_RADAR (7)
- #define CYW43_REASON_RSN_MISMATCH (8)
- #define CYW43_REASON_PRUNE_NO_COMMON_RATES (9)
- #define CYW43_REASON_PRUNE_BASIC_RATES (10)
- #define CYW43_REASON_PRUNE_CCXFAST_PREVAP (11)
- #define CYW43_REASON_PRUNE_CIPHER_NA (12)
- #define CYW43_REASON_PRUNE_KNOWN_STA (13)
- #define CYW43_REASON_PRUNE_CCXFAST_DROAM (14)
- #define CYW43_REASON_PRUNE_WDS_PEER (15)
- #define CYW43_REASON_PRUNE_QBSS_LOAD (16)
- #define CYW43_REASON_PRUNE_HOME_AP (17)
- #define CYW43_REASON_PRUNE_AP_BLOCKED (18)
- #define CYW43_REASON_PRUNE_NO_DIAG_SUPPORT (19)
- #define CYW43_REASON_SUP_OTHER (0)
- #define CYW43_REASON_SUP_DECRYPT_KEY_DATA (1)
- #define CYW43_REASON_SUP_BAD_UCAST_WEP128 (2)
- #define CYW43_REASON_SUP_BAD_UCAST_WEP40 (3)
- #define CYW43_REASON_SUP_UNSUP_KEY_LEN (4)
- #define CYW43_REASON_SUP_PW_KEY_CIPHER (5)
- #define CYW43_REASON_SUP_MSG3_TOO_MANY_IE (6)
- #define CYW43_REASON_SUP_MSG3_IE_MISMATCH (7)
- #define CYW43_REASON_SUP_NO_INSTALL_FLAG (8)
- #define CYW43_REASON_SUP_MSG3_NO_GTK (9)

- #define CYW43_REASON_SUP_GRP_KEY_CIPHER (10)
- #define CYW43_REASON_SUP_GRP_MSG1_NO_GTK (11)
- #define CYW43_REASON_SUP_GTK_DECRYPT_FAIL (12)
- #define CYW43_REASON_SUP_SEND_FAIL (13)
- #define CYW43_REASON_SUP_DEAUTH (14)
- #define CYW43_REASON_SUP_WPA_PSK_TMO (15)
- #define CYW43_NO_POWERSAVE_MODE (0)
- #define CYW43_PM1_POWERSAVE_MODE (1)
- #define CYW43_PM2_POWERSAVE_MODE (2)
- #define CYW43_BUS_MAX_BLOCK_SIZE 64
- #define CYW43_BACKPLANE_READ_PAD_LEN_BYTES 16
- #define CYW43_LL_STATE_SIZE_WORDS 526 + 5 + ((CYW43_BACKPLANE_READ_PAD_LEN_BYTES / 4) + 1)
- #define CYW43_CHANNEL_NONE (0xffffffff)

Typedefs

typedef struct _cyw43_async_event_t cyw43_async_event_t

typedef struct _cyw43_ll_t cyw43_ll_t

Functions

void cyw43_ll_init (cyw43_ll_t *self, void *cb_data)

void cyw43_ll_deinit (cyw43_ll_t *self)

int cyw43_ll_bus_init (cyw43_ll_t *self, const uint8_t *mac)

void cyw43_ll_bus_sleep (cyw43_ll_t *self, bool can_sleep)

void cyw43_ll_process_packets (cyw43_ll_t *self)

int cyw43_ll_ioctl (cyw43_ll_t *self, uint32_t cmd, size_t len, uint8_t *buf, uint32_t iface)

int cyw43_ll_send_ethernet (cyw43_ll_t *self, int itf, size_t len, const void *buf, bool is_pbuf)

int cyw43_ll_wifi_on (cyw43_ll_t *self, uint32_t country)

int cyw43_ll_wifi_pm (cyw43_ll_t *self, uint32_t pm, uint32_t pm_sleep_ret, uint32_t li_bcn, uint32_t li_dtim, uint32_t li_assoc)

int cyw43_ll_wifi_get_pm (cyw43_ll_t *self, uint32_t *pm, uint32_t *pm_sleep_ret, uint32_t *li_bcn, uint32_t *li_dtim, uint32_t *li_assoc) int cyw43_ll_wifi_scan (cyw43_ll_t *self, cyw43_wifi_scan_options_t *opts)

int cyw43_ll_wifi_join (cyw43_ll_t *self, size_t ssid_len, const uint8_t *ssid, size_t key_len, const uint8_t *key, uint32_t auth_type, const uint8_t *bssid, uint32_t channel)

void cyw43_ll_wifi_set_wpa_auth (cyw43_ll_t *self)

void cyw43_ll_wifi_rejoin (cyw43_ll_t *self)

int cyw43_ll_wifi_get_bssid (cyw43_ll_t *self_in, uint8_t *bssid)

int cyw43_ll_wifi_ap_init (cyw43_ll_t *self, size_t ssid_len, const uint8_t *ssid, uint32_t auth, size_t key_len, const uint8_t *key, uint32_t channel)

int cyw43_ll_wifi_ap_set_up (cyw43_ll_t *self, bool up)

int cyw43_ll_wifi_ap_get_stas (cyw43_ll_t *self, int *num_stas, uint8_t *macs)

int cyw43_ll_gpio_set (cyw43_ll_t *self, int gpio_n, bool gpio_en)

int cyw43_ll_gpio_get (cyw43_ll_t *self_in, int gpio_n, bool *gpio_en)

int cyw43_ll_wifi_get_mac (cyw43_ll_t *self_in, uint8_t *addr)

int cyw43_ll_wifi_update_multicast_filter (cyw43_ll_t *self_in, uint8_t *addr, bool add)

bool cyw43_ll_has_work (cyw43_ll_t *self)

bool cyw43_ll_bt_has_work (cyw43_ll_t *self)

int cyw43_cb_read_host_interrupt_pin (void *cb_data)

void cyw43_cb_ensure_awake (void *cb_data)

void cyw43_cb_process_async_event (void *cb_data, const cyw43_async_event_t *ev)

void cyw43_cb_process_ethernet (void *cb_data, int itf, size_t len, const uint8_t *buf)

void cyw43_ll_write_backplane_reg (cyw43_ll_t *self_in, uint32_t addr, uint32_t val)

```
uint32_t cyw43_ll_read_backplane_reg (cyw43_ll_t *self_in, uint32_t addr)
```

```
int cyw43_ll_write_backplane_mem (cyw43_ll_t *self_in, uint32_t addr, uint32_t len, const uint8_t *buf)
```

```
int cyw43_ll_read_backplane_mem (cyw43_ll_t *self_in, uint32_t addr, uint32_t len, uint8_t *buf)
```

anonymous enum

```
anonymous enum
```

Network interface types .

Table 32. Enumerator

CYW43_ITF_STA	Client interface STA mode.
CYW43_ITF_AP	Access point (AP) interface mode.

cyw43_ev_scan_result_t

```
typedef struct _cyw43_ev_scan_result_t cyw43_ev_scan_result_t
```

Structure to return wifi scan results.

cyw43_wifi_scan_options_t

```
typedef struct _cyw43_wifi_scan_options_t cyw43_wifi_scan_options_t
```

wifi scan options passed to cyw43_wifi_scan

Authorization types

Used when setting up an access point, or connecting to an access point

CYW43_AUTH_OPEN

```
#define CYW43_AUTH_OPEN (0)
```

No authorisation required (open)

CYW43_AUTH_WPA_TKIP_PSK

```
#define CYW43_AUTH_WPA_TKIP_PSK (0x00200002)
```

WPA authorisation.

CYW43_AUTH_WPA2_AES_PSK

```
#define CYW43_AUTH_WPA2_AES_PSK (0x00400004)
```

WPA2 authorisation (preferred)

CYW43_AUTH_WPA2_MIXED_PSK

```
#define CYW43_AUTH_WPA2_MIXED_PSK (0x00400006)
```

WPA2/WPA mixed authorisation.

CYW43_AUTH_WPA3_SAE_AES_PSK

```
#define CYW43_AUTH_WPA3_SAE_AES_PSK (0x01000004)
```

WPA3 AES authorisation.

CYW43_AUTH_WPA3_WPA2_AES_PSK

```
#define CYW43_AUTH_WPA3_WPA2_AES_PSK (0x01400004)
```

WPA2/WPA3 authorisation

Macro Definition Documentation

CYW43_IOCTL_GET_SSID

`#define CYW43_IOCTL_GET_SSID (0x32)`

CYW43_IOCTL_GET_CHANNEL

`#define CYW43_IOCTL_GET_CHANNEL (0x3a)`

CYW43_IOCTL_SET_DISASSOC

`#define CYW43_IOCTL_SET_DISASSOC (0x69)`

CYW43_IOCTL_GET_ANTDIV

`#define CYW43_IOCTL_GET_ANTDIV (0x7e)`

CYW43_IOCTL_SET_ANTDIV

`#define CYW43_IOCTL_SET_ANTDIV (0x81)`

CYW43_IOCTL_SET_MONITOR

`#define CYW43_IOCTL_SET_MONITOR (0xd9)`

CYW43_IOCTL_GET_RSSI

`#define CYW43_IOCTL_GET_RSSI (0xfe)`

CYW43_IOCTL_GET_VAR

`#define CYW43_IOCTL_GET_VAR (0x20c)`

CYW43_IOCTL_SET_VAR

`#define CYW43_IOCTL_SET_VAR (0x20f)`

CYW43_EV_SET_SSID

`#define CYW43_EV_SET_SSID (0)`

CYW43_EV_JOIN

`#define CYW43_EV_JOIN (1)`

CYW43_EV_AUTH

`#define CYW43_EV_AUTH (3)`

CYW43_EV_DEAUTH

`#define CYW43_EV_DEAUTH (5)`

CYW43_EV_DEAUTH_IND

`#define CYW43_EV_DEAUTH_IND (6)`

CYW43_EV_ASSOC

`#define CYW43_EV_ASSOC (7)`

CYW43_EV_DISASSOC

`#define CYW43_EV_DISASSOC (11)`

CYW43_EV_DISASSOC_IND

`#define CYW43_EV_DISASSOC_IND (12)`

CYW43_EV_LINK

`#define CYW43_EV_LINK (16)`

CYW43_EV_PRUNE

`#define CYW43_EV_PRUNE (23)`

CYW43_EV_PSK_SUP

`#define CYW43_EV_PSK_SUP (46)`

CYW43_EV_ICV_ERROR

`#define CYW43_EV_ICV_ERROR (49)`

CYW43_EV_ESCAN_RESULT

`#define CYW43_EV_ESCAN_RESULT (69)`

CYW43_EV_CSA_COMPLETE_IND

`#define CYW43_EV_CSA_COMPLETE_IND (80)`

CYW43_EV_ASSOC_REQ_IE

`#define CYW43_EV_ASSOC_REQ_IE (87)`

CYW43_EV_ASSOC_RESP_IE

`#define CYW43_EV_ASSOC_RESP_IE (88)`

CYW43_STATUS_SUCCESS

`#define CYW43_STATUS_SUCCESS (0)`

CYW43_STATUS_FAIL

`#define CYW43_STATUS_FAIL (1)`

CYW43_STATUS_TIMEOUT

`#define CYW43_STATUS_TIMEOUT (2)`

CYW43_STATUS_NO_NETWORKS

`#define CYW43_STATUS_NO_NETWORKS (3)`

CYW43_STATUS_ABORT

`#define CYW43_STATUS_ABORT (4)`

CYW43_STATUS_NO_ACK

`#define CYW43_STATUS_NO_ACK (5)`

CYW43_STATUS_UNSOLICITED

`#define CYW43_STATUS_UNSOLICITED (6)`

CYW43_STATUS_ATTEMPT

`#define CYW43_STATUS_ATTEMPT (7)`

CYW43_STATUS_PARTIAL

`#define CYW43_STATUS_PARTIAL (8)`

CYW43_STATUS_NEWSCAN

`#define CYW43_STATUS_NEWSCAN (9)`

CYW43_STATUS_NEWASSOC

`#define CYW43_STATUS_NEWASSOC (10)`

CYW43_SUP_DISCONNECTED

```
#define CYW43_SUP_DISCONNECTED (0)
```

CYW43_SUP_CONNECTING

```
#define CYW43_SUP_CONNECTING (1)
```

CYW43_SUP_IDREQUIRED

```
#define CYW43_SUP_IDREQUIRED (2)
```

CYW43_SUP_AUTHENTICATING

```
#define CYW43_SUP_AUTHENTICATING (3)
```

CYW43_SUP_AUTHENTICATED

```
#define CYW43_SUP_AUTHENTICATED (4)
```

CYW43_SUP_KEYXCHANGE

```
#define CYW43_SUP_KEYXCHANGE (5)
```

CYW43_SUP_KEYED

```
#define CYW43_SUP_KEYED (6)
```

CYW43_SUP_TIMEOUT

```
#define CYW43_SUP_TIMEOUT (7)
```

CYW43_SUP_LAST_BASIC_STATE

```
#define CYW43_SUP_LAST_BASIC_STATE (8)
```

CYW43_SUP_KEYXCHANGE_WAIT_M1

```
#define CYW43_SUP_KEYXCHANGE_WAIT_M1 CYW43_SUP_AUTHENTICATED
```

CYW43_SUP_KEYXCHANGE_PREP_M2

```
#define CYW43_SUP_KEYXCHANGE_PREP_M2 CYW43_SUP_KEYXCHANGE
```

CYW43_SUP_KEYXCHANGE_WAIT_M3

```
#define CYW43_SUP_KEYXCHANGE_WAIT_M3 CYW43_SUP_LAST_BASIC_STATE
```

CYW43_SUP_KEYXCHANGE_PREP_M4

```
#define CYW43_SUP_KEYXCHANGE_PREP_M4 (9)
```

CYW43_SUP_KEYXCHANGE_WAIT_G1

```
#define CYW43_SUP_KEYXCHANGE_WAIT_G1 (10)
```

CYW43_SUP_KEYXCHANGE_PREP_G2

```
#define CYW43_SUP_KEYXCHANGE_PREP_G2 (11)
```

CYW43_REASON_INITIAL_ASSOC

```
#define CYW43_REASON_INITIAL_ASSOC (0)
```

CYW43_REASON_LOW_RSSI

```
#define CYW43_REASON_LOW_RSSI (1)
```

CYW43_REASON_DEAUTH

```
#define CYW43_REASON_DEAUTH (2)
```

CYW43_REASON_DISASSOC

```
#define CYW43_REASON_DISASSOC (3)
```

CYW43_REASON_BCNS_LOST

```
#define CYW43_REASON_BCNS_LOST (4)
```

CYW43_REASON_FAST_ROAM_FAILED

```
#define CYW43_REASON_FAST_ROAM_FAILED (5)
```

CYW43_REASON_DIRECTED_ROAM

```
#define CYW43_REASON_DIRECTED_ROAM (6)
```

CYW43_REASON_TSPEC_REJECTED

```
#define CYW43_REASON_TSPEC_REJECTED (7)
```

CYW43_REASON_BETTER_AP

```
#define CYW43_REASON_BETTER_AP (8)
```

CYW43_REASON_PRUNE_ENCR_MISMATCH

```
#define CYW43_REASON_PRUNE_ENCR_MISMATCH (1)
```

CYW43_REASON_PRUNE_BCAST_BSSID

```
#define CYW43_REASON_PRUNE_BCAST_BSSID (2)
```

CYW43_REASON_PRUNE_MAC_DENY

```
#define CYW43_REASON_PRUNE_MAC_DENY (3)
```

CYW43_REASON_PRUNE_MAC_NA

```
#define CYW43_REASON_PRUNE_MAC_NA (4)
```

CYW43_REASON_PRUNE_REG_PASSV

```
#define CYW43_REASON_PRUNE_REG_PASSV (5)
```

CYW43_REASON_PRUNE_SPCT_MGMT

```
#define CYW43_REASON_PRUNE_SPCT_MGMT (6)
```

CYW43_REASON_PRUNE_RADAR

```
#define CYW43_REASON_PRUNE_RADAR (7)
```

CYW43_REASON_RSN_MISMATCH

```
#define CYW43_REASON_RSN_MISMATCH (8)
```

CYW43_REASON_PRUNE_NO_COMMON_RATES

```
#define CYW43_REASON_PRUNE_NO_COMMON_RATES (9)
```

CYW43_REASON_PRUNE_BASIC_RATES

```
#define CYW43_REASON_PRUNE_BASIC_RATES (10)
```

CYW43_REASON_PRUNE_CCXFAST_PREVAP

```
#define CYW43_REASON_PRUNE_CCXFAST_PREVAP (11)
```

CYW43_REASON_PRUNE_CIPHER_NA

```
#define CYW43_REASON_PRUNE_CIPHER_NA (12)
```

CYW43_REASON_PRUNE_KNOWN_STA

```
#define CYW43_REASON_PRUNE_KNOWN_STA (13)
```

CYW43_REASON_PRUNE_CCXFAST_DROAM

```
#define CYW43_REASON_PRUNE_CCXFAST_DROAM (14)
```

CYW43_REASON_PRUNE_WDS_PEER

```
#define CYW43_REASON_PRUNE_WDS_PEER (15)
```

CYW43_REASON_PRUNE_QBSS_LOAD

```
#define CYW43_REASON_PRUNE_QBSS_LOAD (16)
```

CYW43_REASON_PRUNE_HOME_AP

```
#define CYW43_REASON_PRUNE_HOME_AP (17)
```

CYW43_REASON_PRUNE_AP_BLOCKED

```
#define CYW43_REASON_PRUNE_AP_BLOCKED (18)
```

CYW43_REASON_PRUNE_NO_DIAG_SUPPORT

```
#define CYW43_REASON_PRUNE_NO_DIAG_SUPPORT (19)
```

CYW43_REASON_SUP_OTHER

```
#define CYW43_REASON_SUP_OTHER (0)
```

CYW43_REASON_SUP_DECRYPT_KEY_DATA

```
#define CYW43_REASON_SUP_DECRYPT_KEY_DATA (1)
```

CYW43_REASON_SUP_BAD_UCAST_WEP128

```
#define CYW43_REASON_SUP_BAD_UCAST_WEP128 (2)
```

CYW43_REASON_SUP_BAD_UCAST_WEP40

```
#define CYW43_REASON_SUP_BAD_UCAST_WEP40 (3)
```

CYW43_REASON_SUP_UNSUP_KEY_LEN

```
#define CYW43_REASON_SUP_UNSUP_KEY_LEN (4)
```

CYW43_REASON_SUP_PW_KEY_CIPHER

```
#define CYW43_REASON_SUP_PW_KEY_CIPHER (5)
```

CYW43_REASON_SUP_MSG3_TOO_MANY_IE

```
#define CYW43_REASON_SUP_MSG3_TOO_MANY_IE (6)
```

CYW43_REASON_SUP_MSG3_IE_MISMATCH

```
#define CYW43_REASON_SUP_MSG3_IE_MISMATCH (7)
```

CYW43_REASON_SUP_NO_INSTALL_FLAG

```
#define CYW43_REASON_SUP_NO_INSTALL_FLAG (8)
```

CYW43_REASON_SUP_MSG3_NO_GTK

```
#define CYW43_REASON_SUP_MSG3_NO_GTK (9)
```

CYW43_REASON_SUP_GRP_KEY_CIPHER

```
#define CYW43_REASON_SUP_GRP_KEY_CIPHER (10)
```

CYW43_REASON_SUP_GRP_MSG1_NO_GTK

```
#define CYW43_REASON_SUP_GRP_MSG1_NO_GTK (11)
```

CYW43_REASON_SUP_GTK_DECRYPT_FAIL

```
#define CYW43_REASON_SUP_GTK_DECRYPT_FAIL (12)
```

CYW43_REASON_SUP_SEND_FAIL

`#define CYW43_REASON_SUP_SEND_FAIL (13)`

CYW43_REASON_SUP_DEAUTH

`#define CYW43_REASON_SUP_DEAUTH (14)`

CYW43_REASON_SUP_WPA_PSK_TMO

`#define CYW43_REASON_SUP_WPA_PSK_TMO (15)`

CYW43_NO_POWERSAVE_MODE

`#define CYW43_NO_POWERSAVE_MODE (0)`

Power save mode parameter passed to cyw43_ll_wifi_pm.

No Powersave mode

CYW43_PM1_POWERSAVE_MODE

`#define CYW43_PM1_POWERSAVE_MODE (1)`

Powersave mode on specified interface without regard for throughput reduction.

CYW43_PM2_POWERSAVE_MODE

`#define CYW43_PM2_POWERSAVE_MODE (2)`

Powersave mode on specified interface with High throughput.

CYW43_BUS_MAX_BLOCK_SIZE

`#define CYW43_BUS_MAX_BLOCK_SIZE 64`

CYW43_BACKPLANE_READ_PAD_LEN_BYTES

`#define CYW43_BACKPLANE_READ_PAD_LEN_BYTES 16`

CYW43_LL_STATE_SIZE_WORDS

`#define CYW43_LL_STATE_SIZE_WORDS 526 + 5 + ((CYW43_BACKPLANE_READ_PAD_LEN_BYTES / 4) + 1)`

CYW43_CHANNEL_NONE

`#define CYW43_CHANNEL_NONE (0xffffffff)`

To indicate no specific channel when calling cyw43_ll_wifi_join with bssid specified.

No Channel specified (use the AP's channel)

Typedef Documentation

cyw43_async_event_t

`typedef struct _cyw43_async_event_t cyw43_async_event_t`

cyw43_ll_t

`typedef struct _cyw43_ll_t cyw43_ll_t`

Function Documentation

cyw43_cb_ensure_awake

`void cyw43_cb_ensure_awake (void * cb_data)`

cyw43_cb_process_async_event

`void cyw43_cb_process_async_event (void * cb_data, const cyw43_async_event_t * ev)`

cyw43_cb_process_ethernet

void cyw43_cb_process_ethernet (void * cb_data, int itf, size_t len, const uint8_t * buf)

cyw43_cb_read_host_interrupt_pin

int cyw43_cb_read_host_interrupt_pin (void * cb_data)

cyw43_ll_bt_has_work

bool cyw43_ll_bt_has_work (cyw43_ll_t * self)

cyw43_ll_bus_init

int cyw43_ll_bus_init (cyw43_ll_t * self, const uint8_t * mac)

cyw43_ll_bus_sleep

void cyw43_ll_bus_sleep (cyw43_ll_t * self, bool can_sleep)

cyw43_ll_deinit

void cyw43_ll_deinit (cyw43_ll_t * self)

cyw43_ll_gpio_get

int cyw43_ll_gpio_get (cyw43_ll_t * self_in, int gpio_n, bool * gpio_en)

cyw43_ll_gpio_set

int cyw43_ll_gpio_set (cyw43_ll_t * self, int gpio_n, bool gpio_en)

cyw43_ll_has_work

bool cyw43_ll_has_work (cyw43_ll_t * self)

cyw43_ll_init

void cyw43_ll_init (cyw43_ll_t * self, void * cb_data)

cyw43_ll_ioctl

int cyw43_ll_ioctl (cyw43_ll_t * self, uint32_t cmd, size_t len, uint8_t * buf, uint32_t iface)

cyw43_ll_process_packets

void cyw43_ll_process_packets (cyw43_ll_t * self)

cyw43_ll_read_backplane_mem

int cyw43_ll_read_backplane_mem (cyw43_ll_t * self_in, uint32_t addr, uint32_t len, uint8_t * buf)

cyw43_ll_read_backplane_reg

uint32_t cyw43_ll_read_backplane_reg (cyw43_ll_t * self_in, uint32_t addr)

cyw43_ll_send_ethernet

int cyw43_ll_send_ethernet (cyw43_ll_t * self, int itf, size_t len, const void * buf, bool is_pbuf)

cyw43_ll_wifi_ap_get_stas

int cyw43_ll_wifi_ap_get_stas (cyw43_ll_t * self, int * num_stas, uint8_t * macs)

cyw43_ll_wifi_ap_init

int cyw43_ll_wifi_ap_init (cyw43_ll_t * self, size_t ssid_len, const uint8_t * ssid, uint32_t auth, size_t key_len, const uint8_t * key, uint32_t channel)

cyw43_ll_wifi_ap_set_up

int cyw43_ll_wifi_ap_set_up (cyw43_ll_t * self, bool up)

cyw43_ll_wifi_get_bssid

int cyw43_ll_wifi_get_bssid (cyw43_ll_t * self_in, uint8_t * bssid)

cyw43_ll_wifi_get_mac

```
int cyw43_ll_wifi_get_mac (cyw43_ll_t * self_in, uint8_t * addr)
```

cyw43_ll_wifi_get_pm

```
int cyw43_ll_wifi_get_pm (cyw43_ll_t * self, uint32_t * pm, uint32_t * pm_sleep_ret, uint32_t * li_bcn, uint32_t * li_dtim, uint32_t * li_assoc)
```

cyw43_ll_wifi_join

```
int cyw43_ll_wifi_join (cyw43_ll_t * self, size_t ssid_len, const uint8_t * ssid, size_t key_len, const uint8_t * key, uint32_t auth_type, const uint8_t * bssid, uint32_t channel)
```

cyw43_ll_wifi_on

```
int cyw43_ll_wifi_on (cyw43_ll_t * self, uint32_t country)
```

cyw43_ll_wifi_pm

```
int cyw43_ll_wifi_pm (cyw43_ll_t * self, uint32_t pm, uint32_t pm_sleep_ret, uint32_t li_bcn, uint32_t li_dtim, uint32_t li_assoc)
```

cyw43_ll_wifi_rejoin

```
void cyw43_ll_wifi_rejoin (cyw43_ll_t * self)
```

cyw43_ll_wifi_scan

```
int cyw43_ll_wifi_scan (cyw43_ll_t * self, cyw43_wifi_scan_options_t * opts)
```

cyw43_ll_wifi_set_wpa_auth

```
void cyw43_ll_wifi_set_wpa_auth (cyw43_ll_t * self)
```

cyw43_ll_wifi_update_multicast_filter

```
int cyw43_ll_wifi_update_multicast_filter (cyw43_ll_t * self_in, uint8_t * addr, bool add)
```

cyw43_ll_write_backplane_mem

```
int cyw43_ll_write_backplane_mem (cyw43_ll_t * self_in, uint32_t addr, uint32_t len, const uint8_t * buf)
```

cyw43_ll_write_backplane_reg

```
void cyw43_ll_write_backplane_reg (cyw43_ll_t * self_in, uint32_t addr, uint32_t val)
```

4.5. Runtime Infrastructure

Libraries that are used to provide efficient implementation of certain language level and C library functions, as well as CMake INTERFACE libraries abstracting the compilation and link steps in the SDK

boot_stage2	Second stage boot loaders responsible for setting up external flash.
pico_atomic	Helper implementations for C11 atomics.
pico_base	Core types and macros for the Raspberry Pi Pico SDK.
pico_binary_info	Binary info is intended for embedding machine readable information with the binary in FLASH.
pico_bootrom	Access to functions and data in the bootrom.
pico_bit_ops	Optimized bit manipulation functions.
pico_cxx_options	non-code library controlling C++ related compile options
pico_clib_interface	Provides the necessary glue code required by the particular C/C++ runtime being used.

pico_crt0	Provides the default linker scripts and the program entry/exit point.
pico_divider	Optimized 32 and 64 bit division functions accelerated by the RP2040 hardware divider.
pico_double	Optimized double-precision floating point functions.
pico_float	Optimized single-precision floating point functions.
pico_int64_ops	Optimized replacement implementations of the compiler built-in 64 bit multiplication.
pico_malloc	Multi-core safety for malloc, calloc and free.
pico_mem_ops	Provides optimized replacement implementations of the compiler built-in memcpy, memset and related functions.
pico_platform	Macros and definitions (and functions when included by non assembly code) for the RP2 family device / architecture to provide a common abstraction over low level compiler / platform specifics.
pico_printf	Compact replacement for printf by Marco Paland (info@paland.com)
pico_runtime	Basic runtime support for running pre-main initializers provided by other libraries.
pico_runtime_init	Main runtime initialization functions required to set up the runtime environment before entering main.
pico_stdio	Customized stdio support allowing for input and output from UART, USB, semi-hosting etc.
pico_stdio_semihosting	Experimental support for stdout using RAM semihosting.
pico_stdio_uart	Support for stdin/stdout using UART.
pico_stdio_rtt	Support for stdin/stdout using SEGGER RTT.
pico_stdio_usb	Support for stdin/stdout over USB serial (CDC)
pico_standard_binary_info	Includes default information about the binary that can be displayed by picotool.
pico_standard_link	Setup for link options for a standard SDK executable.

4.5.1. boot_stage2

Second stage boot loaders responsible for setting up external flash.

4.5.2. pico_atomic

Helper implementations for C11 atomics.

4.5.2.1. Detailed Description

On RP2040 a spin lock is used as protection for all atomic operations, since there is no C library support.

On RP2350 the C-library provides implementations for all 1-byte, 2-byte and 4-byte atomics using processor exclusive operations. This library provides a spin-lock protected version for arbitrary-sized atomics (including 64-bit).

4.5.3. pico_base

Core types and macros for the Raspberry Pi Pico SDK.

4.5.3.1. Detailed Description

This header is intended to be included by all source code as it includes configuration headers and overrides in the correct order

This header may be included by assembly code

4.5.3.2. Enumerations

```
enum pico_error_codes { PICO_OK = 0, PICO_ERROR_NONE = 0, PICO_ERROR_GENERIC = -1, PICO_ERROR_TIMEOUT = -2,
PICO_ERROR_NO_DATA = -3, PICO_ERROR_NOT_PERMITTED = -4, PICO_ERROR_INVALID_ARG = -5, PICO_ERROR_IO = -6,
PICO_ERROR_BADAUTH = -7, PICO_ERROR_CONNECT_FAILED = -8, PICO_ERROR_INSUFFICIENT_RESOURCES = -9,
PICO_ERROR_INVALID_ADDRESS = -10, PICO_ERROR_BAD_ALIGNMENT = -11, PICO_ERROR_INVALID_STATE = -12,
PICO_ERROR_BUFFER_TOO_SMALL = -13, PICO_ERROR_PRECONDITION_NOT_MET = -14, PICO_ERROR_MODIFIED_DATA = -15,
PICO_ERROR_INVALID_DATA = -16, PICO_ERROR_NOT_FOUND = -17, PICO_ERROR_UNSUPPORTED_MODIFICATION = -18,
PICO_ERROR_LOCK_REQUIRED = -19, PICO_ERROR_VERSION_MISMATCH = -20, PICO_ERROR_RESOURCE_IN_USE = -21 }
```

Common return codes from pico_sdk methods that return a status.

4.5.3.3. Enumeration Type Documentation

4.5.3.3.1. pico_error_codes

`enum pico_error_codes`

Common return codes from pico_sdk methods that return a status.

All `PICO_ERROR_` values are negative so they can be returned from functions that also want to return a zero or positive value on success.

Note these error codes may be returned via bootrom functions too.

Table 33. Enumerator

PICO_OK	No error; the operation succeeded.
PICO_ERROR_NONE	No error; the operation succeeded.
PICO_ERROR_GENERIC	An unspecified error occurred.
PICO_ERROR_TIMEOUT	The function failed due to timeout.
PICO_ERROR_NO_DATA	Attempt for example to read from an empty buffer/FIFO.
PICO_ERROR_NOT_PERMITTED	Permission violation e.g. write to read-only flash partition, or security violation.
PICO_ERROR_INVALID_ARG	Argument is outside of range of supported values`.
PICO_ERROR_IO	An I/O error occurred.
PICO_ERROR_BADAUTH	The authorization failed due to bad credentials.
PICO_ERROR_CONNECT_FAILED	The connection failed.
PICO_ERROR_INSUFFICIENT_RESOURCES	Dynamic allocation of resources failed.

PICO_ERROR_INVALID_ADDRESS	Address argument was out-of-bounds or was determined to be an address that the caller may not access.
PICO_ERROR_BAD_ALIGNMENT	Address was mis-aligned (usually not on word boundary)
PICO_ERROR_INVALID_STATE	Something happened or failed to happen in the past, and consequently we (currently) can't service the request.
PICO_ERROR_BUFFER_TOO_SMALL	A user-allocated buffer was too small to hold the result or working state of this function.
PICO_ERROR_PRECONDITION_NOT_MET	The call failed because another function must be called first.
PICO_ERROR_MODIFIED_DATA	Cached data was determined to be inconsistent with the actual version of the data.
PICO_ERROR_INVALID_DATA	A data structure failed to validate.
PICO_ERROR_NOT_FOUND	Attempted to access something that does not exist; or, a search failed.
PICO_ERROR_UNSUPPORTED_MODIFICATION	Write is impossible based on previous writes; e.g. attempted to clear an OTP bit.
PICO_ERROR_LOCK_REQUIRED	A required lock is not owned.
PICO_ERROR_VERSION_MISMATCH	A version mismatch occurred (e.g. trying to run PIO version 1 code on RP2040)
PICO_ERROR_RESOURCE_IN_USE	The call could not proceed because requires resourcesw were unavailable.

4.5.4. pico_binary_info

Binary info is intended for embedding machine readable information with the binary in FLASH.

4.5.4.1. Detailed Description

Example uses include:

- Program identification / information
- Pin layouts
- Included features
- Identifying flash regions used as block devices/storage

4.5.4.2. Macros

- `#define bi_decl(_decl) __bi_mark_enclosure _decl; __bi_decl(__bi_ptr_lineno_var_name, &__bi_lineno_var_name.core, ".binary_info.keep.", __used);`
- `#define bi_decl_if_func_used(_decl) ({__bi_mark_enclosure _decl; __bi_decl(__bi_ptr_lineno_var_name, &__bi_lineno_var_name.core, ".binary_info.", ); *(const volatile uint8_t *)&__bi_ptr_lineno_var_name;});`

4.5.4.3. Macro Definition Documentation

4.5.4.3.1. bi_decl

```
#define bi_decl(_decl) __bi_mark_enclosure _decl; __bi_decl(__bi_ptr_lineno_var_name, &__bi_lineno_var_name.core,
".binary_info.keep.", __used);
```

Declare some binary information that will be included if the contain source file/line is compiled into the binary.

4.5.4.3.2. bi_decl_if_func_used

```
#define bi_decl_if_func_used(_decl) ({__bi_mark_enclosure _decl; __bi_decl(__bi_ptr_lineno_var_name,
&__bi_lineno_var_name.core, ".binary_info.", ); *(const volatile uint8_t *)&__bi_ptr_lineno_var_name;});
```

Declare some binary information that will be included if the function containing the decl is linked into the binary. The SDK uses –gc-sections, so functions that are never called will be removed by the linker, and any associated binary information declared this way will also be stripped.

4.5.5. pico_bootrom

Access to functions and data in the bootrom.

4.5.5.1. Detailed Description

This header may be included by assembly code

4.5.5.2. Macros

- `#define ROM_TABLE_CODE(c1, c2) ((c1) | ((c2) << 8))`

4.5.5.3. Functions

`static uint32_t rom_table_code (uint8_t c1, uint8_t c2)`

Return a bootrom lookup code based on two ASCII characters.

`void * rom_func_lookup (uint32_t code)`

Lookup a bootrom function by its code.

`void * rom_data_lookup (uint32_t code)`

Lookup a bootrom data address by its code.

`bool rom_funcs_lookup (uint32_t *table, unsigned int count)`

Helper function to lookup the addresses of multiple bootrom functions.

`static __force_inline void * rom_func_lookup_inline (uint32_t code)`

Lookup a bootrom function by code. This method is forcibly inlined into the caller for FLASH/RAM sensitive code usage.

`static __force_inline void * rom_data_lookup_inline (uint32_t code)`

Lookup a bootrom data address by its code. This method is forcibly inlined into the caller for FLASH/RAM sensitive code usage.

`void rom_reset_usb_boot (uint32_t usb_activity_gpio_pin_mask, uint32_t disable_interface_mask)`

Reboot the device into BOOTSEL mode.

`void rom_reset_usb_boot_extra (int usb_activity_gpio_pin, uint32_t disable_interface_mask, bool usb_activity_gpio_pin_active_low)`

Reboot the device into BOOTSEL mode.

`static void rom_connect_internal_flash (void)`

Connect the SSI/QMI to the QSPI pads.

`static void rom_flash_exit_xip (void)`

Return the QSPI device from its XIP state to a serial command state.

`static void rom_flash_range_erase (uint32_t addr, size_t count, uint32_t block_size, uint8_t block_cmd)`

Erase bytes in flash.

`static void rom_flash_range_program (uint32_t addr, const uint8_t *data, size_t count)`

Program bytes in flash.

`static void rom_flash_flush_cache (void)`

Flush the XIP cache.

`static void rom_flash_enter_cmd_xip (void)`

Configure the SSI/QMI with a standard command.

`static int rom_reboot (uint32_t flags, uint32_t delay_ms, uint32_t p0, uint32_t p1)` RP2350

Reboot using the watchdog.

`static void rom_bootrom_state_reset (uint32_t flags)` RP2350

Reset bootrom state.

`static void rom_flash_reset_address_trans (void)` RP2350

Reset address translation.

`static void rom_flash_select_xip_read_mode (bootrom_xip_mode_t mode, uint8_t clkdiv)` RP2350

Configure QMI in a XIP read mode.

`static int rom_flash_op (cflash_flags_t flags, uintptr_t addr, uint32_t size_bytes, uint8_t *buf)` RP2350

Perform a flash read, erase, or program operation.

`static int rom_func_otp_access (uint8_t *buf, uint32_t buf_len, otp_cmd_t cmd)` RP2350

Writes data from a buffer into OTP, or reads data from OTP into a buffer.

`static int rom_get_partition_table_info (uint32_t *out_buffer, uint32_t out_buffer_word_size, uint32_t partition_and_flags)` RP2350

Fills a buffer with information from the partition table.

`static int rom_load_partition_table (uint8_t *workarea_base, uint32_t workarea_size, bool force_reload)` RP2350

Loads the current partition table from flash, if present.

`static int rom_pick_ab_partition (uint8_t *workarea_base, uint32_t workarea_size, uint partition_a_num, uint32_t flash_update_boot_window_base)` RP2350

Pick a partition from an A/B pair.

`static int rom_get_b_partition (uint pi_a)` RP2350

Get B partition.

`static int rom_get_uf2_target_partition (uint8_t *workarea_base, uint32_t workarea_size, uint32_t family_id, resident_partition_t *partition_out)` RP2350

Get UF2 Target Partition.

`static intptr_t rom_flash_runtime_to_storage_addr (uintptr_t flash_runtime_addr)` RP2350

Translate runtime to storage address.

`static int rom_chain_image (uint8_t *workarea_base, uint32_t workarea_size, uint32_t region_base, uint32_t region_size)` RP2350

Chain into a launchable image.

`static int rom_explicit_buy (uint8_t *buffer, uint32_t buffer_size)` RP2350

Buy an image.

`static int rom_set_ns_api_permission (uint ns_api_num, bool allowed)` RP2350

Set NS API Permission.

`static void * rom_validate_ns_buffer (const void *addr, uint32_t size, uint32_t write, uint32_t *ok)` RP2350

Validate NS Buffer.

`static intptr_t rom_set_rom_callback (uint callback_num, bootrom_api_callback_generic_t funcptr)` RP2350

Set ROM callback function.

`static int rom_get_sys_info (uint32_t *out_buffer, uint32_t out_buffer_word_size, uint32_t flags)` RP2350

Get system information.

`int rom_add_flash_runtime_partition (uint32_t start_offset, uint32_t size, uint32_t permissions)` RP2350

Add a runtime partition to the partition table to specify flash permissions.

4.5.5.4. Macro Definition Documentation

4.5.5.4.1. ROM_TABLE_CODE

```
#define ROM_TABLE_CODE(c1, c2) ((c1) | ((c2) << 8))
```

Return a bootrom lookup code based on two ASCII characters.

These codes are uses to lookup data or function addresses in the bootrom

Parameters

`c1`	the first character
`c2`	the second character

Returns

the 'code' to use in rom_func_lookup() or rom_data_lookup()

4.5.5.5. Function Documentation

4.5.5.5.1. rom_add_flash_runtime_partition RP2350

```
int rom_add_flash_runtime_partition (uint32_t start_offset, uint32_t size, uint32_t permissions)
```

Add a runtime partition to the partition table to specify flash permissions.

Note that a partition is added to the runtime view of the partition table maintained by the bootrom if there is space to do so

Note that these permissions cannot override the permissions for any pre-existing partitions, as permission matches are made on a first partition found basis.

Parameters

`start_offset`	the start_offset into flash in bytes (must be a multiple of 4K)

`size`	the size in byte (must be a multiple of 4K)
`permissions`	the bitwise OR of permissions from PICOBIN_PARTITION_PERMISSION_ constants, e.g. PICOBIN_PARTITION_PERMISSION_S_R_BITS from boot/picobin.h

Returns

>= 0 the partition number added if PICO_ERROR_BAD_ALIGNMENT if the start_offset or size aren't multiples of 4K. PICO_ERROR_INVALID_ARG if the start_offset or size are out of range, or invalid permission bits are set.

4.5.5.5.2. rom_bootrom_state_reset RP2350

```
static void rom_bootrom_state_reset (uint32_t flags) [inline], [static]
```

Reset bootrom state.

Resets internal bootrom state, based on the following flags:

STATE_RESET_CURRENT_CORE - Resets any internal bootrom state for the current core into a clean state. This method should be called prior to calling any other bootrom APIs on the current core, and is called automatically by the bootrom during normal boot of core 0 and launch of code on core 1.

STATE_RESET_OTHER_CORE - Resets any internal bootrom state for the other core into a clean state. This is generally called by a debugger when resetting the state of one core via code running on the other.

STATE_RESET_GLOBAL_STATE - Resets all non core-specific state, including: Disables access to bootrom APIs from ARM-NS Unlocks all BOOT spinlocks Clears any secure code callbacks

Note: the sdk calls this method on runtime initialisation to put the bootrom into a known state. This allows the program to function correctly if it is entered (e.g. from a debugger) without taking the usual boot path (which resets the state appropriately itself).

Parameters

`flags`	flags, as detailed above

4.5.5.5.3. rom_chain_image RP2350

```
static int rom_chain_image (uint8_t * workarea_base, uint32_t workarea_size, uint32_t region_base, uint32_t region_size)
[inline], [static]
```

Chain into a launchable image.

Searches a memory region for a launchable image, and executes it if possible.

The region_base and region_size specify a word-aligned, word-multiple-sized area of RAM, XIP RAM or flash to search. The first 4 kiB of the region must contain the start of a Block Loop with an IMAGE_DEF. If the new image is launched, the call does not return otherwise an error is returned.

The region_base is signed, as a negative value can be passed, which indicates that the (negated back to positive value) is both the region_base and the base of the "flash update" region.

This method potentially requires similar complexity to the boot path in terms of picking amongst versions, checking signatures etc. As a result it requires a user provided memory buffer as a work area. The work area should be word aligned, and of sufficient size or BOOTROM_ERROR_INSUFFICIENT_RESOURCES will be returned. The work area size currently required is 3264, so 3.25K is a good choice.

> **NOTE**
>
> This method is primarily expected to be used when implementing bootloaders.

> **NOTE**
>
> When chaining into an image, the OTP_DATA_BOOT_FLAGS0_ROLLBACK_REQUIRED flag will not be set, to prevent invalidating a bootloader without a rollback version by booting a binary which has one.

Parameters

`workarea_base`	base address of work area
`workarea_size`	size of work area
`region_base`	base address of image
`region_size`	size of window containing image

4.5.5.5.4. rom_connect_internal_flash

`static void rom_connect_internal_flash (void) [inline], [static]`

Connect the SSI/QMI to the QSPI pads.

Restore all QSPI pad controls to their default state, and connect the SSI/QMI peripheral to the QSPI pads.

On RP2350 if a secondary flash chip select GPIO has been configured via OTP OTP_DATA_FLASH_DEVINFO, or by writing to the runtime copy of FLASH_DEVINFO in bootram, then this bank 0 GPIO is also initialised and the QMI peripheral is connected. Otherwise, bank 0 IOs are untouched.

4.5.5.5.5. rom_data_lookup

`void * rom_data_lookup (uint32_t code)`

Lookup a bootrom data address by its code.

Parameters

`code`	the code

Returns

a pointer to the data, or NULL if the code does not match any bootrom function

4.5.5.5.6. rom_data_lookup_inline

`static __force_inline void * rom_data_lookup_inline (uint32_t code) [static]`

Lookup a bootrom data address by its code. This method is forcibly inlined into the caller for FLASH/RAM sensitive code usage.

Parameters

`code`	the code

Returns

a pointer to the data, or NULL if the code does not match any bootrom data

4.5.5.5.7. rom_explicit_buy RP2350

`static int rom_explicit_buy (uint8_t * buffer, uint32_t buffer_size) [inline], [static]`

Buy an image.

Perform an "explicit" buy of an executable launched via an IMAGE_DEF which was "explicit buy" flagged. A "flash update" boot of such an image is a way to have the image execute once, but only become the "current" image if it calls back into the bootrom via this call.

This call may perform the following:

- Erase and rewrite the part of flash containing the "explicit buy" flag in order to clear said flag.
- Erase the first sector of the other partition in an A/B partition scenario, if this new IMAGE_DEF is a version downgrade (so this image will boot again when not doing a "flash update" boot)
- Update the rollback version in OTP if the chip is secure, and a rollback version is present in the image.

> **NOTE**
>
> The device may reboot while updating the rollback version, if multiple rollback rows need to be written - this occurs when the version crosses a multiple of 24 (for example upgrading from version 23 to 25 requires a reboot, but 23 to 24 or 24 to 25 doesn't). The application should therefore be prepared to reboot when calling this function, if rollback versions are in use.

Note that the first of the above requires 4 kiB of scratch space, so you should pass a word aligned buffer of at least 4 kiB to this method, or it will return BOOTROM_ERROR_INSUFFICIENT_RESOURCES if the "explicit buy" flag needs to be cleared.

Parameters

`buffer`	base address of scratch space
`buffer_size`	size of scratch space

4.5.5.5.8. rom_flash_enter_cmd_xip

`static void rom_flash_enter_cmd_xip (void) [inline], [static]`

Configure the SSI/QMI with a standard command.

Configure the SSI/QMI to generate a standard 03h serial read command, with 24 address bits, upon each XIP access. This is a slow XIP configuration, but is widely supported. CLKDIV is set to 12 on RP2350. The debugger may call this function to ensure that flash is readable following a program/erase operation.

Note that the same setup is performed by flash_exit_xip(), and the RP2350 flash program/erase functions do not leave XIP in an inaccessible state, so calls to this function are largely redundant on RP2350. It is provided on RP2350 for compatibility with RP2040.

4.5.5.5.9. rom_flash_exit_xip

`static void rom_flash_exit_xip (void) [inline], [static]`

Return the QSPI device from its XIP state to a serial command state.

On RP2040, first set up the SSI for serial-mode operations, then issue the fixed XIP exit sequence described in Section 2.8.1.2 of the datasheet. Note that the bootrom code uses the IO forcing logic to drive the CS pin, which must be cleared before returning the SSI to XIP mode (e.g. by a call to _flash_flush_cache). This function configures the SSI with a fixed SCK clock divisor of /6.

On RP2350, Initialise the QMI for serial operations (direct mode), and also initialise a basic XIP mode, where the QMI will perform 03h serial read commands at low speed (CLKDIV=12) in response to XIP reads.

Then, issue a sequence to the QSPI device on chip select 0, designed to return it from continuous read mode ("XIP mode") and/or QPI mode to a state where it will accept serial commands. This is necessary after system reset to restore the QSPI device to a known state, because resetting RP2350 does not reset attached QSPI devices. It is also necessary when user code, having already performed some continuous-read-mode or QPI-mode accesses, wishes to return the QSPI device to a state where it will accept the serial erase and programming commands issued by the bootrom's flash access functions.

If a GPIO for the secondary chip select is configured via FLASH_DEVINFO, then the XIP exit sequence is also issued to chip select 1.

The QSPI device should be accessible for XIP reads after calling this function; the name flash_exit_xip refers to returning the QSPI device from its XIP state to a serial command state.

4.5.5.5.10. rom_flash_flush_cache

```
static void rom_flash_flush_cache (void) [inline], [static]
```

Flush the XIP cache.

Flush and enable the XIP cache. Also clears the IO forcing on QSPI CSn, so that the SSI can drive the flash chip select as normal.

Flush the entire XIP cache, by issuing an invalidate by set/way maintenance operation to every cache line. This ensures that flash program/erase operations are visible to subsequent cached XIP reads.

Note that this unpins pinned cache lines, which may interfere with cache-as-SRAM use of the XIP cache.

No other operations are performed.

4.5.5.5.11. rom_flash_op RP2350

```
static int rom_flash_op (cflash_flags_t flags, uintptr_t addr, uint32_t size_bytes, uint8_t * buf) [inline], [static]
```

Perform a flash read, erase, or program operation.

The flash operation is bounds-checked against the known flash devices specified by the runtime value of FLASH_DEVINFO, stored in bootram. This is initialised by the bootrom to the OTP value OTP_DATA_FLASH_DEVINFO, if OTP_DATA_BOOT_FLAGS0_FLASH_DEVINFO_ENABLE is set; otherwise it is initialised to 16 MiB for chip select 0 and 0 bytes for chip select 1. FLASH_DEVINFO can be updated at runtime by writing to its location in bootram, the pointer to which can be looked up in the ROM table.

If a resident partition table is in effect, then the flash operation is also checked against the partition permissions. The Secure version of this function can specify the caller's effective security level (Secure, Non-secure, bootloader) using the CFLASH_SECLEVEL_BITS bitfield of the flags argument, whereas the Non-secure function is always checked against the Non-secure permissions for the partition. Flash operations which span two partitions are not allowed, and will fail address validation.

If OTP_DATA_FLASH_DEVINFO_D8H_ERASE_SUPPORTED is set, erase operations will use a D8h 64 kiB block erase command where possible (without erasing outside the specified region), for faster erase time. Otherwise, only 20h 4 kiB sector erase commands are used.

Optionally, this API can translate addr from flash runtime addresses to flash storage addresses, according to the translation currently configured by QMI address translation registers, QMI_ATRANS0 through QMI_ATRANS7. For example, an image stored at a +2 MiB offset in flash (but mapped at XIP address 0 at runtime), writing to an offset of +1 MiB into the image, will write to a physical flash storage address of 3 MiB. Translation is enabled by setting the CFLASH_ASPACE_BITS bitfield in the flags argument.

When translation is enabled, flash operations which cross address holes in the XIP runtime address space (created by non-maximum ATRANSx_SIZE) will return an error response. This check may tear: the transfer may be partially performed before encountering an address hole and ultimately returning failure.

When translation is enabled, flash operations are permitted to cross chip select boundaries, provided this does not span an ATRANS address hole. When translation is disabled, the entire operation must target a single flash chip select (as determined by bits 24 and upward of the address), else address validation will fail.

Parameters

`flags`	controls the security level, address space, and flash operation
`addr`	the address of the first flash byte to be accessed, ranging from XIP_BASE to XIP_BASE + 0x1ffffff
`size_bytes`	size of buf, in bytes
`buf`	contains data to be written to flash, for program operations, and data read back from flash, for read operations

4.5.5.5.12. rom_flash_range_erase

```
static void rom_flash_range_erase (uint32_t addr, size_t count, uint32_t block_size, uint8_t block_cmd) [inline],
[static]
```

Erase bytes in flash.

Erase count bytes, starting at addr (offset from start of flash). Optionally, pass a block erase command e.g. D8h block erase, and the size of the block erased by this command - this function will use the larger block erase where possible, for much higher erase speed. addr must be aligned to a 4096-byte sector, and count must be a multiple of 4096 bytes.

This is a low-level flash API, and no validation of the arguments is performed.

See rom_flash_op on RP2350 for a higher-level API which checks alignment, flash bounds and partition permissions, and can transparently apply a runtime-to-storage address translation.

The QSPI device must be in a serial command state before calling this API, which can be achieved by calling rom_connect_internal_flash() followed by rom_flash_exit_xip(). After the erase, the flash cache should be flushed via rom_flash_flush_cache() to ensure the modified flash data is visible to cached XIP accesses.

Finally, the original XIP mode should be restored by copying the saved XIP setup function from bootram into SRAM, and executing it: the bootrom provides a default function which restores the flash mode/clkdiv discovered during flash scanning, and user programs can override this with their own XIP setup function.

For the duration of the erase operation, QMI is in direct mode and attempting to access XIP from DMA, the debugger or the other core will return a bus fault. XIP becomes accessible again once the function returns.

Parameters

`addr`	the offset from start of flash to be erased
`count`	number of bytes to erase
`block_size`	optional size of block erased by block_cmd
`block_cmd`	optional block erase command e.g. D8h block erase

4.5.5.5.13. rom_flash_range_program

```
static void rom_flash_range_program (uint32_t addr, const uint8_t * data, size_t count) [inline], [static]
```

Program bytes in flash.

Program data to a range of flash addresses starting at addr (offset from the start of flash) and count bytes in size. addr must be aligned to a 256-byte boundary, and count must be a multiple of 256.

This is a low-level flash API, and no validation of the arguments is performed.

See rom_flash_op on RP2350 for a higher-level API which checks alignment, flash bounds and partition permissions, and can transparently apply a runtime-to-storage address translation.

The QSPI device must be in a serial command state before calling this API - see notes on rom_flash_range_erase

Parameters

`addr`	the offset from start of flash to be erased
`data`	buffer containing the data to be written
`count`	number of bytes to erase

4.5.5.5.14. rom_flash_reset_address_trans RP2350

`static void rom_flash_reset_address_trans (void) [inline], [static]`

Reset address translation.

Restore the QMI address translation registers, QMI_ATRANS0 through QMI_ATRANS7, to their reset state. This makes the runtime-to-storage address map an identity map, i.e. the mapped and unmapped address are equal, and the entire space is fully mapped.

4.5.5.5.15. rom_flash_runtime_to_storage_addr RP2350

`static intptr_t rom_flash_runtime_to_storage_addr (uintptr_t flash_runtime_addr) [inline], [static]`

Translate runtime to storage address.

Applies the address translation currently configured by QMI address translation registers.

Translating an address outside of the XIP runtime address window, or beyond the bounds of an ATRANSx_SIZE field, returns BOOTROM_ERROR_INVALID_ADDRESS, which is not a valid flash storage address. Otherwise, return the storage address which QMI would access when presented with the runtime address addr. This is effectively a virtual-to-physical address translation for QMI.

Parameters

`flash_runtime_addr`	the address to translate

4.5.5.5.16. rom_flash_select_xip_read_mode RP2350

`static void rom_flash_select_xip_read_mode (bootrom_xip_mode_t mode, uint8_t clkdiv) [inline], [static]`

Configure QMI in a XIP read mode.

Configure QMI for one of a small menu of XIP read modes supported by the bootrom. This mode is configured for both memory windows (both chip selects), and the clock divisor is also applied to direct mode.

Parameters

`mode`	bootrom_xip_mode_t mode to use
`clkdiv`	clock divider

4.5.5.5.17. rom_func_lookup

`void * rom_func_lookup (uint32_t code)`

Lookup a bootrom function by its code.

Parameters

`code`	the code

Returns

a pointer to the function, or NULL if the code does not match any bootrom function

4.5.5.5.18. rom_func_lookup_inline

`static __force_inline void * rom_func_lookup_inline (uint32_t code) [static]`

Lookup a bootrom function by code. This method is forcibly inlined into the caller for FLASH/RAM sensitive code usage.

Parameters

`code`	the code

Returns

a pointer to the function, or NULL if the code does not match any bootrom function

4.5.5.5.19. rom_func_otp_access RP2350

`static int rom_func_otp_access (uint8_t * buf, uint32_t buf_len, otp_cmd_t cmd) [inline], [static]`

Writes data from a buffer into OTP, or reads data from OTP into a buffer.

The buffer must be aligned to 2 bytes or 4 bytes according to the IS_ECC flag.

This method will read and write rows until the first row it encounters that fails a key or permission check at which it will return BOOTROM_ERROR_NOT_PERMITTED.

Writing will also stop at the first row where an attempt is made to set an OTP bit from a 1 to a 0, and BOOTROM_ERROR_UNSUPPORTED_MODIFICATION will be returned.

If all rows are read/written successfully, then BOOTROM_OK will be returned.

Parameters

`buf` buffer to read to/write from

`buf_len` size of buf

`cmd` OTP command to execute

- 0x0000ffff - ROW_NUMBER: 16 low bits are row number (0-4095)
- 0x00010000 - IS_WRITE: if set, do a write (not a read)
- 0x00020000 - IS_ECC: if this bit is set, each value in the buffer is 2 bytes and ECC is used when read/writing from 24 bit value in OTP. If this bit is not set, each value in the buffer is 4 bytes, the low 24-bits of which are written to or read from OTP.

4.5.5.5.20. rom_funcs_lookup

`bool rom_funcs_lookup (uint32_t * table, unsigned int count)`

Helper function to lookup the addresses of multiple bootrom functions.

This method looks up the 'codes' in the table, and convert each table entry to the looked up function pointer, if there is a function for that code in the bootrom.

Parameters

`table`	an IN/OUT array, elements are codes on input, function pointers on success.
`count`	the number of elements in the table

Returns

true if all the codes were found, and converted to function pointers, false otherwise

4.5.5.5.21. rom_get_b_partition RP2350

`static int rom_get_b_partition (uint pi_a) [inline], [static]`

Get B partition.

Returns the index of the B partition of partition A if a partition table is present and loaded, and there is a partition A with a B partition; otherwise returns BOOTROM_ERROR_NOT_FOUND.

Parameters

`pi_a`	the A partition number

4.5.5.5.22. rom_get_partition_table_info RP2350

`static int rom_get_partition_table_info (uint32_t * out_buffer, uint32_t out_buffer_word_size, uint32_t partition_and_flags) [inline], [static]`

Fills a buffer with information from the partition table.

Fills a buffer with information from the partition table. Note that this API is also used to return information over the picoboot interface.

On success, the buffer is filled, and the number of words filled in the buffer is returned. If the partition table has not been loaded (e.g. from a watchdog or RAM boot), then this method will return BOOTROM_ERROR_NO_DATA, and you should load the partition table via load_partition_table() first.

Note that not all data from the partition table is kept resident in memory by the bootrom due to size constraints. To protect against changes being made in flash after the bootrom has loaded the resident portion, the bootrom keeps a hash of the partition table as of the time it loaded it. If the hash has changed by the time this method is called, then it will return BOOTROM_ERROR_INVALID_STATE.

The information returned is chosen by the partition_and_flags parameter; the first word in the returned buffer, is the (sub)set of those flags that the API supports. You should always check this value before interpreting the buffer.

Following the first word, returns words of data for each present flag in order. With the exception of PT_INFO, all the flags select "per partition" information, so each field is returned in flag order for one partition after the next. The special SINGLE_PARTITION flag indicates that data for only a single partition is required.

Parameters

`out_buffer`	buffer to write data to
`out_buffer_word_size`	size of out_buffer, in words
`partition_and_flags`	partition number and flags

4.5.5.5.23. rom_get_sys_info RP2350

`static int rom_get_sys_info (uint32_t * out_buffer, uint32_t out_buffer_word_size, uint32_t flags) [inline], [static]`

Get system information.

Fills a buffer with various system information. Note that this API is also used to return information over the picoboot interface.

On success, the buffer is filled, and the number of words filled in the buffer is returned.

The information returned is chosen by the flags parameter; the first word in the returned buffer, is the (sub)set of those flags that the API supports. You should always check this value before interpreting the buffer.

"Boot Diagnostic" information is intended to help identify the cause of a failed boot, or booting into an unexpected binary. This information can be retrieved via picoboot after a watchdog reboot, however it will not survive a reset via the RUN pin or POWMAN reset.

There is only one word of diagnostic information. What it records is based on the pp selection above, which is itself set as a parameter when rebooting programmatically into a normal boot.

To get diagnostic info, pp must refer to a slot or an "A" partition; image diagnostics are automatically selected on boot from OTP or RAM image, or when chain_image() is called.)

The diagnostic word thus contains data for either slot 0 and slot 1, or the "A" partition (and its "B" partition if it has one). The low half word of the diagnostic word contains information from slot 0 or partition A; the high half word contains information from slot 1 or partition B.

To get a full picture of a failed boot involving slots and multiple partitions, the device can be rebooted multiple times to gather the information.

Parameters

`out_buffer`	buffer to write data to
`out_buffer_word_size`	size of out_buffer, in words
`flags`	flags

4.5.5.5.24. rom_get_uf2_target_partition RP2350

`static int rom_get_uf2_target_partition (uint8_t * workarea_base, uint32_t workarea_size, uint32_t family_id, resident_partition_t * partition_out) [inline], [static]`

Get UF2 Target Partition.

This method performs the same operation to decide on a target partition for a UF2 family ID as when a UF2 is dragged onto the USB drive in BOOTSEL mode.

This method potentially requires similar complexity to the boot path in terms of picking amongst versions, checking signatures etc. As a result it requires a user provided memory buffer as a work area. The work area should byte word-aligned and of sufficient size or `BOOTROM_ERROR_INSUFFICIENT_RESOURCES` will be returned. The work area size currently required is 3264, so 3.25K is a good choice.

If the partition table has not been loaded (e.g. from a watchdog or RAM boot), then this method will return `BOOTROM_ERROR_PRECONDITION_NOT_MET`, and you should load the partition table via load_partition_table() first.

Parameters

`workarea_base`	base address of work area
`workarea_size`	size of work area
`family_id`	the family ID to place
`partition_out`	pointer to the resident_partition_t to fill with the partition data

4.5.5.5.25. rom_load_partition_table RP2350

`static int rom_load_partition_table (uint8_t * workarea_base, uint32_t workarea_size, bool force_reload) [inline], [static]`

Loads the current partition table from flash, if present.

This method potentially requires similar complexity to the boot path in terms of picking amongst versions, checking signatures etc. As a result it requires a user provided memory buffer as a work area. The work area should byte word-aligned and of sufficient size or BOOTROM_ERROR_INSUFFICIENT_RESOURCES will be returned. The work area size currently required is 3264, so 3.25K is a good choice.

If force_reload is false, then this method will return BOOTROM_OK immediately if the bootrom is loaded, otherwise it will reload the partition table if it has been loaded already, allowing for the partition table to be updated in a running program.

Parameters

`workarea_base`	base address of work area
`workarea_size`	size of work area
`force_reload`	force reloading of the partition table

4.5.5.5.26. rom_pick_ab_partition RP2350

`static int rom_pick_ab_partition (uint8_t * workarea_base, uint32_t workarea_size, uint partition_a_num, uint32_t flash_update_boot_window_base) [inline], [static]`

Pick a partition from an A/B pair.

Determines which of the partitions has the "better" IMAGE_DEF. In the case of executable images, this is the one that would be booted

This method potentially requires similar complexity to the boot path in terms of picking amongst versions, checking signatures etc. As a result it requires a user provided memory buffer as a work area. The work area should bye word aligned, and of sufficient size or BOOTROM_ERROR_INSUFFICIENT_RESOURCES will be returned. The work area size currently required is 3264, so 3.25K is a good choice.

The passed partition number can be any valid partition number other than the "B" partition of an A/B pair.

This method returns a negative error code, or the partition number of the picked partition if (i.e. partition_a_num or the number of its "B" partition if any).

> **NOTE**
>
> This method does not look at owner partitions, only the A partition passed and it's corresponding B partition.

Parameters

`workarea_base`	base address of work area
`workarea_size`	size of work area
`partition_a_num`	the A partition of the pair
`flash_update_boot_window_base`	the flash update base, to pick that partition instead of the normally "better" partition

4.5.5.5.27. rom_reboot RP2350

`static int rom_reboot (uint32_t flags, uint32_t delay_ms, uint32_t p0, uint32_t p1) [inline], [static]`

Reboot using the watchdog.

Resets the chip and uses the watchdog facility to restart.

The delay_ms is the millisecond delay before the reboot occurs. Note: by default this method is asynchronous (unless NO_RETURN_ON_SUCCESS is set - see below), so the method will return and the reboot will happen this many milliseconds later.

The flags field contains one of the following values:

REBOOT_TYPE_NORMAL - reboot into the normal boot path.

REBOOT_TYPE_BOOTSEL - reboot into BOOTSEL mode. p0 - the GPIO number to use as an activity indicator (enabled by flag in p1). p1 - a set of flags: 0x01 : DISABLE_MSD_INTERFACE - Disable the BOOTSEL USB drive (see [section_bootrom_mass_storage]) 0x02 : DISABLE_PICOBOOT_INTERFACE - Disable the PICOBOOT interface (see [section_bootrom_picoboot]). 0x10 : GPIO_PIN_ACTIVE_LOW - The GPIO in p0 is active low. 0x20 : GPIO_PIN_ENABLED - Enable the activity indicator on the specified GPIO.

REBOOT_TYPE_RAM_IMAGE - reboot into an image in RAM. The region of RAM or XIP RAM is searched for an image to run. This is the type of reboot used when a RAM UF2 is dragged onto the BOOTSEL USB drive. p0 - the region start address (word-aligned). p1 - the region size (word-aligned).

REBOOT_TYPE_FLASH_UPDATE - variant of REBOOT_TYPE_NORMAL to use when flash has been updated. This is the type of reboot used after dragging a flash UF2 onto the BOOTSEL USB drive. p0 - the address of the start of the region of flash that was updated. If this address matches the start address of a partition or slot, then that partition or slot is treated preferentially during boot (when there is a choice). This type of boot facilitates TBYB and version downgrades.

REBOOT_TYPE_PC_SP - reboot to a specific PC and SP. Note: this is not allowed in the ARM-NS variant. p0 - the initial program counter (PC) to start executing at. This must have the lowest bit set for Arm and clear for RISC-V p1 - the initial stack pointer (SP).

All of the above, can have optional flags ORed in:

REBOOT_TO_ARM - switch both cores to the Arm architecture (rather than leaving them as is). The call will fail with BOOTROM_ERROR_INVALID_STATE if the Arm architecture is not supported. REBOOT_TO_RISCV - switch both cores to the RISC-V architecture (rather than leaving them as is). The call will fail with BOOTROM_ERROR_INVALID_STATE if the RISC-V architecture is not supported. NO_RETURN_ON_SUCCESS - the watchdog h/w is asynchronous. Setting this bit forces this method not to return if the reboot is successfully initiated.

Parameters

`flags`	the reboot flags, as detailed above
`delay_ms`	millisecond delay before the reboot occurs
`p0`	parameter 0, depends on flags
`p1`	parameter 1, depends on flags

4.5.5.5.28. rom_reset_usb_boot

`void rom_reset_usb_boot (uint32_t usb_activity_gpio_pin_mask, uint32_t disable_interface_mask)`

Reboot the device into BOOTSEL mode.

This function reboots the device into the BOOTSEL mode ('usb boot"). Facilities are provided to enable an "activity light" via GPIO attached LED for the USB Mass Storage Device, and to limit the USB interfaces exposed.

Parameters

`usb_activity_gpio_pin_mask`	0 No pins are used as per a cold boot. Otherwise a single bit set indicating which GPIO pin should be set to output and raised whenever there is mass storage activity from the host.
`disable_interface_mask`	value to control exposed interfaces • 0 To enable both interfaces (as per a cold boot) • 1 To disable the USB Mass Storage Interface • 2 To disable the USB PICOBOOT Interface

4.5.5.5.29. rom_reset_usb_boot_extra

`void rom_reset_usb_boot_extra (int usb_activity_gpio_pin, uint32_t disable_interface_mask, bool usb_activity_gpio_pin_active_low)`

Reboot the device into BOOTSEL mode.

This function reboots the device into the BOOTSEL mode ('usb boot"). Facilities are provided to enable an "activity light" via GPIO attached LED for the USB Mass Storage Device, and to limit the USB interfaces exposed.

Parameters

`usb_activity_gpio_pin`	GPIO pin to be used as an activitiy pin, or -1 for none from the host.
`disable_interface_mask`	value to control exposed interfaces • 0 To enable both interfaces (as per a cold boot) • 1 To disable the USB Mass Storage Interface • 2 To disable the USB PICOBOOT Interface
`usb_activity_gpio_pin_active_low`	Activity GPIO is active low (ignored on RP2040)

4.5.5.5.30. rom_set_ns_api_permission RP2350

`static int rom_set_ns_api_permission (uint ns_api_num, bool allowed) [inline], [static]`

Set NS API Permission.

Allow or disallow the specific NS API (note all NS APIs default to disabled).

ns_api_num configures ARM-NS access to the given API. When an NS API is disabled, calling it will return BOOTROM_ERROR_NOT_PERMITTED.

> **NOTE**
>
> All permissions default to disallowed after a reset.

Parameters

`ns_api_num`	ns api number
`allowed`	permission

4.5.5.5.31. rom_set_rom_callback RP2350

`static intptr_t rom_set_rom_callback (uint callback_num, bootrom_api_callback_generic_t funcptr) [inline], [static]`

Set ROM callback function.

The only currently supported callback_number is 0 which sets the callback used for the secure_call API.

A callback pointer of 0 deletes the callback function, a positive callback pointer (all valid function pointers are on RP2350) sets the callback function, but a negative callback pointer can be passed to get the old value without setting a new value.

If successful, returns >=0 (the existing value of the function pointer on entry to the function).

Parameters

`callback_num`	the callback number to set - only 0 is supported on RP2350
`funcptr`	pointer to the callback function

4.5.5.5.32. rom_table_code

`static uint32_t rom_table_code (uint8_t c1, uint8_t c2) [inline], [static]`

Return a bootrom lookup code based on two ASCII characters.

These codes are uses to lookup data or function addresses in the bootrom

Parameters

`c1`	the first character

c2 the second character

Returns

the 'code' to use in rom_func_lookup() or rom_data_lookup()

4.5.5.5.33. rom_validate_ns_buffer RP2350

```
static void * rom_validate_ns_buffer (const void * addr, uint32_t size, uint32_t write, uint32_t * ok) [inline], [static]
```

Validate NS Buffer.

Utility method that can be used by secure ARM code to validate a buffer passed to it from Non-secure code.

Both the write parameter and the (out) result parameter ok are RCP booleans, so 0xa500a500 for true, and 0x00c300c3 for false. This enables hardening of this function, and indeed the write parameter must be one of these values or the RCP will hang the system.

For success, the entire buffer must fit in range XIP_BASE -> SRAM_END, and must be accessible by the Non-secure caller according to SAU + NS MPU (privileged or not based on current processor IPSR and NS CONTROL flag). Buffers in USB RAM are also allowed if access is granted to NS via ACCESSCTRL.

Parameters

addr	buffer address
size	buffer size
write	rcp boolean, true if writeable
ok	rcp boolean result

4.5.6. pico_bit_ops

Optimized bit manipulation functions.

4.5.6.1. Detailed Description

Additionally provides replacement implementations of the compiler built-ins __builtin_popcount, __builtin_clz and __bulitin_ctz

4.5.6.2. Functions

`uint32_t __rev (uint32_t bits)`

Reverse the bits in a 32 bit word.

`uint64_t __revll (uint64_t bits)`

Reverse the bits in a 64 bit double word.

4.5.6.3. Function Documentation

4.5.6.3.1. __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

4.5.6.3.2. __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

4.5.7. pico_cxx_options

non-code library controlling C++ related compile options

4.5.8. pico_clib_interface

Provides the necessary glue code required by the particular C/C++ runtime being used.

4.5.9. pico_crt0

Provides the default linker scripts and the program entry/exit point.

4.5.10. pico_divider

Optimized 32 and 64 bit division functions accelerated by the RP2040 hardware divider.

4.5.10.1. Detailed Description

Additionally provides integration with the C `/` and `%` operators

4.5.10.2. Functions

`int32_t div_s32s32 (int32_t a, int32_t b)`

Integer divide of two signed 32-bit values.

`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.

`divmod_result_t divmod_s32s32 (int32_t a, int32_t b)`

Integer divide of two signed 32-bit values.

`uint32_t div_u32u32 (uint32_t a, uint32_t b)`

Integer divide of two unsigned 32-bit values.

`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.

`divmod_result_t divmod_u32u32 (uint32_t a, uint32_t b)`

Integer divide of two unsigned 32-bit values.

`int64_t div_s64s64 (int64_t a, int64_t b)`

Integer divide of two signed 64-bit values.

`int64_t divmod_s64s64_rem (int64_t a, int64_t b, int64_t *rem)`

Integer divide of two signed 64-bit values, with remainder.

`int64_t divmod_s64s64 (int64_t a, int64_t b)`

Integer divide of two signed 64-bit values.

`uint64_t div_u64u64 (uint64_t a, uint64_t b)`

Integer divide of two unsigned 64-bit values.

`uint64_t divmod_u64u64_rem (uint64_t a, uint64_t b, uint64_t *rem)`

Integer divide of two unsigned 64-bit values, with remainder.

`uint64_t divmod_u64u64 (uint64_t a, uint64_t b)`

Integer divide of two signed 64-bit values.

`int32_t div_s32s32_unsafe (int32_t a, int32_t b)`

Unsafe integer divide of two signed 32-bit values.

`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.

`divmod_result_t divmod_s32s32_unsafe (int32_t a, int32_t b)`

Unsafe integer divide of two unsigned 32-bit values.

`uint32_t div_u32u32_unsafe (uint32_t a, uint32_t b)`

Unsafe integer divide of two unsigned 32-bit values.

`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.

`divmod_result_t divmod_u32u32_unsafe (uint32_t a, uint32_t b)`

Unsafe integer divide of two unsigned 32-bit values.

`int64_t div_s64s64_unsafe (int64_t a, int64_t b)`

Unsafe integer divide of two signed 64-bit values.

`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.

`int64_t divmod_s64s64_unsafe (int64_t a, int64_t b)`

Unsafe integer divide of two signed 64-bit values.

`uint64_t div_u64u64_unsafe (uint64_t a, uint64_t b)`

Unsafe integer divide of two unsigned 64-bit values.

`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.

`uint64_t divmod_u64u64_unsafe (uint64_t a, uint64_t b)`

Unsafe integer divide of two signed 64-bit values.

4.5.10.3. Function Documentation

4.5.10.3.1. 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

4.5.10.3.2. div_s32s32_unsafe

`int32_t div_s32s32_unsafe (int32_t a, int32_t b)`

Unsafe integer divide of two signed 32-bit values.

Parameters

`a` Dividend

`b` Divisor

Returns

quotient

Do not use in interrupts

4.5.10.3.3. 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

4.5.10.3.4. div_s64s64_unsafe

`int64_t div_s64s64_unsafe (int64_t a, int64_t b)`

Unsafe integer divide of two signed 64-bit values.

Parameters

`a` Dividend

`b` Divisor

Returns

Quotient

Do not use in interrupts

4.5.10.3.5. 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

4.5.10.3.6. div_u32u32_unsafe

`uint32_t div_u32u32_unsafe (uint32_t a, uint32_t b)`

Unsafe integer divide of two unsigned 32-bit values.

Parameters

`a` Dividend

`b` Divisor

Returns

Quotient

Do not use in interrupts

4.5.10.3.7. 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

4.5.10.3.8. div_u64u64_unsafe

`uint64_t div_u64u64_unsafe (uint64_t a, uint64_t b)`

Unsafe integer divide of two unsigned 64-bit values.

Parameters

`a` Dividend

`b` Divisor

Returns

Quotient

Do not use in interrupts

4.5.10.3.9. 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

4.5.10.3.10. divmod_s32s32_rem

`static int32_t divmod_s32s32_rem (int32_t a, int32_t b, int32_t * rem) [inline], [static]`

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

4.5.10.3.11. 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.

Parameters

`a` Dividend

`b` Divisor

`rem` The remainder of dividend/divisor

Returns

Quotient result of dividend/divisor

Do not use in interrupts

4.5.10.3.12. divmod_s32s32_unsafe

`divmod_result_t divmod_s32s32_unsafe (int32_t a, int32_t b)`

Unsafe integer divide of two unsigned 32-bit values.

Parameters

`a`	Dividend
`b`	Divisor

Returns

quotient in low word/r0, remainder in high word/r1

Do not use in interrupts

4.5.10.3.13. 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)

4.5.10.3.14. 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

4.5.10.3.15. 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.

Parameters

`a`	Dividend
`b`	Divisor
`rem`	The remainder of dividend/divisor

Returns

Quotient result of dividend/divisor

Do not use in interrupts

4.5.10.3.16. divmod_s64s64_unsafe

`int64_t divmod_s64s64_unsafe (int64_t a, int64_t b)`

Unsafe integer divide of two signed 64-bit values.

Parameters

`a`	Dividend
`b`	Divisor

Returns

quotient in result (r0,r1), remainder in regs (r2, r3)

Do not use in interrupts

4.5.10.3.17. 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

4.5.10.3.18. divmod_u32u32_rem

`static uint32_t divmod_u32u32_rem (uint32_t a, uint32_t b, uint32_t * rem) [inline], [static]`

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

4.5.10.3.19. 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.

Parameters

`a`	Dividend

`b`	Divisor
`rem`	The remainder of dividend/divisor

Returns

Quotient result of dividend/divisor

Do not use in interrupts

4.5.10.3.20. divmod_u32u32_unsafe

`divmod_result_t divmod_u32u32_unsafe (uint32_t a, uint32_t b)`

Unsafe integer divide of two unsigned 32-bit values.

Parameters

`a`	Dividend
`b`	Divisor

Returns

quotient in low word/r0, remainder in high word/r1

Do not use in interrupts

4.5.10.3.21. 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)

4.5.10.3.22. 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

4.5.10.3.23. 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.

Parameters

`a`	Dividend
`b`	Divisor
`rem`	The remainder of dividend/divisor

Returns

Quotient result of dividend/divisor

Do not use in interrupts

4.5.10.3.24. divmod_u64u64_unsafe

`uint64_t divmod_u64u64_unsafe (uint64_t a, uint64_t b)`

Unsafe integer divide of two signed 64-bit values.

Parameters

`a`	Dividend
`b`	Divisor

Returns

quotient in result (r0,r1), remainder in regs (r2, r3)

Do not use in interrupts

4.5.11. pico_double

Optimized double-precision floating point functions.

4.5.11.1. Detailed Description

An application can take control of the floating point routines used in the application over and above what is provided by the compiler, by depending on the pico_double library. A user might want to do this:

1. To use optimized software implementations provided by the RP2-series device's bootrom or the SDK
2. To use optimized combined software/hardware implementations utilizing custom RP2-series hardware for acceleration
3. To control the amount of C compiler/library code bloat
4. To make sure no floating point is called at all

The pico_double library comes in three main flavors:

1. `pico_double_none` - all floating point operations cause a panic - no double-precision floating point code is included
2. `pico_double_compiler` - no custom functions are provided; all double-precision floating point is handled by the C compiler/library
3. `pico_double_pico` - the smallest and fastest available for the platform, along with additional functionality (e.g. fixed point conversions) which are detailed below

The user can control which version they want (e.g. **pico_double_xxx** by either setting the CMake global variable `PICO_DEFAULT_DOUBLE_IMPL=xxx`, or by using the CMake function `pico_set_double_implementation(<TARGET> xxx)`. Note that in the absence of either, pico_double_pico is used by default.

On RP2040, `pico_double_pico` uses optimized hand coded implementations from the bootrom and the SDK for both basic double-precision floating point operations and floating point math library functions. These implementations are generally faster and smaller than those provided by the C compiler/library, though they don't support all the features of a fully compliant floating point implementation; they are however usually fine for the majority of cases

On RP2350, `pico_double_pico` uses RP2350 DCP instructions (double co-processor) to implement fast version of the basic arithmetic functions, and provides optimized M33 implementations of trignometric and scientific functions. These implementations are generally faster and smaller than those provided by the C compiler/library, though they don't support all the features of a fully compliant floating point implementation; they are however usually fine for the majority of cases

On Arm, (replacement) optimized implementations are provided for the following compiler built-ins and math library functions when using `pico_double_pico`:

- basic arithmetic:

  __aeabi_dadd, __aeabi_ddiv, __aeabi_dmul, __aeabi_drsub, __aeabi_dsub

- comparison:

  __aeabi_cfcmpeq, __aeabi_cfrcmple, __aeabi_cfcmple, __aeabi_dcmpeq, __aeabi_dcmplt, __aeabi_dcmple, __aeabi_dcmpge, __aeabi_dcmpgt, __aeabi_dcmpun

- (u)int32 <-> double:

  __aeabi_i2d, __aeabi_ui2d, __aeabi_d2iz, __aeabi_d2uiz

- (u)int64 <-> double:

  __aeabi_l2d, __aeabi_ul2d, __aeabi_d2lz, __aeabi_d2ulz

- double -> float:

  __aeabi_d2d

- basic trigonometric:

  sqrt, cos, sin, tan, atan2, exp, log

- trigonometric and scientific

  ldexp, copysign, trunc, floor, ceil, round, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh, exp2, log2, exp10, log10, pow, hypot, cbrt, fmod, drem, remainder, remquo, expm1, log1p, fma

- GNU exetnsions:

  powint, sincos

On Arm, the following additional optimized functions are also provided when using `pico_double_pico`:

- Conversions to/from integer types:
  - (u)int -> double (round to nearest):

    int2double, uint2double, int642double, uint642double

  - (u)double -> int (round towards zero):

    double2int_z, double2uint_z, double2int64_z, double2uint64_z

  - (u)double -> int (round towards -infinity):

    double2int, double2uint, double2int64, double2uint64
- Conversions to/from fixed point integers:
  - (u)fix -> double (round to nearest):

    fix2double, ufix2double, fix642double, ufix642double

  - double -> (u)fix (round towards zero):

    double2fix_z, double2ufix_z, double2fix64_z, double2ufix64_z

    - double -> (u)fix (round towards -infinity):

      double2fix, double2ufix, double2fix64, double2ufix64

- Even faster versions of divide and square-root functions that do not round correctly:

  ddiv_fast, sqrt_fast (these do not round correctly)

- Faster unfused multiply and accumulate:

  mla (fast fma)

  On RISC-V there is no custom double-precision floating point support, so `pico_double_pico` is equivalent to `pico_double_compiler`

4.5.12. pico_float

Optimized single-precision floating point functions.

4.5.12.1. Detailed Description

An application can take control of the floating point routines used in the application over and above what is provided by the compiler, by depending on the pico_float library. A user might want to do this

1. To use optimized software implementations provided by the RP2-series device's bootrom or the SDK
2. To use optimized combined software/hardware implementations utilizing custom RP2-series hardware for acceleration
3. To control the amount of C compiler/library code bloat
4. To make sure no floating point is called at all

The pico_float library comes in three main flavors:

1. `pico_float_none` - all floating point operations cause a panic - no single-precision floating point code is included
2. `pico_float_compiler` - no custom functions are provided; all single-precision floating point is handled by the C compiler/library
3. `pico_float_pico` - the smallest and fastest available for the platform, along with additional functionality (e.g. fixed point conversions) which are detailed below

The user can control which version they want (e.g. **pico_float_xxx** by either setting the CMake global variable `PICO_DEFAULT_FLOAT_IMPL=xxx`, or by using the CMake function `pico_set_float_implementation(<TARGET> xxx)`. Note that in the absence of either, pico_float_pico is used by default.

On RP2040, `pico_float_pico` uses optimized hand coded implementations from the bootrom and the SDK for both basic single-precision floating point operations and floating point math library functions. These implementations are generally faster and smaller than those provided by the C compiler/library, though they don't support all the features of a fully compliant floating point implementation; they are however usually fine for the majority of cases

On Arm on RP2350, there are multiple options for `pico_float_pico`:

1. `pico_float_pico_vfp` - this library leaves basic C single-precision floating point operations to the compiler which can use inlined VFP (Arm FPU) code. Custom optimized versions of trigonometric and scientific functions are provided. No DCP (RP2350 Double co-processor) instructions are used.
2. `pico_float_pico_dcp` - this library prevents the compiler injecting inlined VFP code, and also implements all single-precision floating point operations in optimized DCP or M33 code. This option is not quite as fast as pico_float_pico_vfp, however it allows floating point operations without enabling the floating point co-processor on the CPU; this can be beneficial in certain circumstances, e.g. where leaving stack in tasks or interrupts for the floating point state is undesirable.

Note: `pico_float_pico` is equivalent to `pico_float_pico_vfp` on RP2350, as this is the most sensible default

On Arm, (replacement) optimized implementations are provided for the following compiler built-ins and math library functions when using `_pico` variants of `pico_float`:

- basic arithmetic: (except `pico_float_pico_vfp`)

  __aeabi_fadd, __aeabi_fdiv, __aeabi_fmul, __aeabi_frsub, __aeabi_fsub

- comparison: (except `pico_float_pico_vfp`)

  __aeabi_cfcmpeq, __aeabi_cfrcmple, __aeabi_cfcmple, __aeabi_fcmpeq, __aeabi_fcmplt, __aeabi_fcmple, __aeabi_fcmpge, __aeabi_fcmpgt, __aeabi_fcmpun

- (u)int32 <-> float: (except `pico_float_pico_vfp`)

  __aeabi_i2f, __aeabi_ui2f, __aeabi_f2iz, __aeabi_f2uiz

- (u)int64 <-> float: (except `pico_float_pico_vfp`)

  __aeabi_l2f, __aeabi_ul2f, __aeabi_f2lz, __aeabi_f2ulz

- float -> double: (except `pico_float_pico_vfp`)

  __aeabi_f2d

- basic trigonometric:

  sqrtf, cosf, sinf, tanf, atan2f, expf, logf

- trigonometric and scientific

  ldexpf, copysignf, truncf, floorf, ceilf, roundf, asinf, acosf, atanf, sinhf, coshf, tanhf, asinhf, acoshf, atanhf, exp2f, log2f, exp10f, log10f, powf, hypotf, cbrtf, fmodf, dremf, remainderf, remquof, expm1f, log1pf, fmaf

- GNU exetnsions:

  powintf, sincosf

On Arm, the following additional optimized functions are also provided (when using `_pico` variants of `pico_float`):

- Conversions to/from integer types:
  - (u)int -> float (round to nearest):

    int2float, uint2float, int642float, uint642float

    note: on `pico_float_pico_vfp` the 32-bit functions are also provided as C macros since they map to inline VFP code

  - (u)float -> int (round towards zero):

    float2int_z, float2uint_z, float2int64_z, float2uint64_z

    note: on `pico_float_pico_vfp` the 32-bit functions are also provided as C macros since they map to inline VFP code

  - (u)float -> int (round towards -infinity):

    float2int, float2uint, float2int64, float2uint64

- Conversions to/from fixed point integers:
  - (u)fix -> float (round to nearest):

    fix2float, ufix2float, fix642float, ufix642float

  - float -> (u)fix (round towards zero):

    float2fix_z, float2ufix_z, float2fix64_z, float2ufix64_z

    note: on `pico_float_pico_vfp` the 32-bit functions are also provided as C macros since they can map to inline VFP code when the number of fractional bits is a compile time constant between 1 and 32

  - float -> (u)fix (round towards -infinity):

    float2fix, float2ufix, float2fix64, float2ufix64

    note: on `pico_float_pico_vfp` the 32-bit functions are also provided as C macros since they can map to inline

VFP code when the number of fractional bits is a compile time constant between 1 and 32

- Even faster versions of divide and square-root functions that do not round correctly: (`pico_float_pico_dcp` only)

  fdiv_fast, sqrtf_fast

  On RISC-V, (replacement) optimized implementations are provided for the following compiler built-ins when using the `pico_float_pico` library (note that there are no variants of this library like there are on Arm):

- basic arithmetic:

  __addsf3, __subsf3, __mulsf3

4.5.13. pico_int64_ops

Optimized replacement implementations of the compiler built-in 64 bit multiplication.

4.5.13.1. Detailed Description

This library does not provide any additional functions

4.5.14. pico_malloc

Multi-core safety for malloc, calloc and free.

4.5.14.1. Detailed Description

This library does not provide any additional functions

4.5.15. pico_mem_ops

Provides optimized replacement implementations of the compiler built-in memcpy, memset and related functions.

4.5.15.1. Detailed Description

The functions include:

- memset, memcpy
- __aeabi_memset, __aeabi_memset4, __aeabi_memset8, __aeabi_memcpy, __aeabi_memcpy4, __aeabi_memcpy8

This library does not provide any additional functions

4.5.16. pico_platform

Macros and definitions (and functions when included by non assembly code) for the RP2 family device / architecture to provide a common abstraction over low level compiler / platform specifics.

4.5.16.1. Detailed Description

Macros and definitions (and functions when included by non assembly code) to adapt for different compilers.

Macros and definitions for accessing the CPU registers.

This header may be included by assembly code

4.5.16.2. Macros

- `#define __fast_mul(a, b)`
- `#define __isr`
- `#define __force_inline __always_inline`
- `#define count_of(a) (sizeof(a)/sizeof((a)[0]))`
- `#define MAX(a, b) ((a)>(b)?(a):(b))`
- `#define MIN(a, b) ((b)>(a)?(a):(b))`
- `#define __check_type_compatible(type_a, type_b) static_assert(__builtin_types_compatible_p(type_a, type_b), __STRING(type_a) " is not compatible with " __STRING(type_b));`
- `#define __after_data(group) __attribute__((section(".after_data." group)))`
- `#define __scratch_x(group) __attribute__((section(".scratch_x." group)))`
- `#define __scratch_y(group) __attribute__((section(".scratch_y." group)))`
- `#define __uninitialized_ram(group) __attribute__((section(".uninitialized_data." #group))) group`
- `#define __in_flash(group) __attribute__((section(".flashdata." group)))`
- `#define __no_inline_not_in_flash_func(func_name) __noinline __not_in_flash_func(func_name)`

4.5.16.3. Functions

`static __force_inline void tight_loop_contents (void)`

No-op function for the body of tight loops.

`static void busy_wait_at_least_cycles (uint32_t minimum_cycles)`

Helper method to busy-wait for at least the given number of cycles.

`static __force_inline void __breakpoint (void)`

Execute a breakpoint instruction.

`static __force_inline uint get_core_num (void)`

Get the current core number.

`static __force_inline uint __get_current_exception (void)`

Get the current exception level on this core.

`static __force_inline bool pico_processor_state_is_nonsecure (void)` RP2350

Return true if executing in the NonSecure state (Arm-only)

`uint8_t rp2350_chip_version (void)` RP2350

Returns the RP2350 chip revision number.

`static uint8_t rp2040_chip_version (void)` RP2350

Returns the RP2040 chip revision number for compatibility.

`static uint8_t rp2040_rom_version (void)`

Returns the RP2040 rom version number.

`static __force_inline int32_t __mul_instruction (int32_t a, int32_t b)`

Multiply two integers using an assembly `MUL` instruction.

`static __always_inline void __compiler_memory_barrier (void)`

Ensure that the compiler does not move memory access across this method call.

`void panic_unsupported (void)`

Panics with the message "Unsupported".

`void panic (const char *fmt,…)`

Displays a panic message and halts execution.

4.5.16.4. Macro Definition Documentation

4.5.16.4.1. __fast_mul

```
#define __fast_mul(a, b) __builtin_choose_expr(__builtin_constant_p(b) &&
!__builtin_constant_p(a), \
    (__builtin_popcount(b) >= 2 ? __mul_instruction(a,b) : (a)*(b)), \
    (a)*(b))
```

multiply two integer values using the fastest method possible

Efficiently multiplies value a by possibly constant value b.

If b is known to be constant and not zero or a power of 2, then a mul instruction is used rather than gcc's default which is often a slow combination of shifts and adds. If b is a power of 2 then a single shift is of course preferable and will be used

Parameters

- `a` the first operand
- `b` the second operand

Returns

a * b

4.5.16.4.2. __isr

`#define __isr`

Marker for an interrupt handler.

For example an IRQ handler function called my_interrupt_handler:

```
void __isr my_interrupt_handler(void) {
```

4.5.16.4.3. __force_inline

`#define __force_inline __always_inline`

Attribute to force inlining of a function regardless of optimization level.

For example my_function here will always be inlined:

```
int __force_inline my_function(int x) {
```

4.5.16.4.4. count_of

`#define count_of(a) (sizeof(a)/sizeof((a)[0]))`

Macro to determine the number of elements in an array.

4.5.16.4.5. MAX

`#define MAX(a, b) ((a)>(b)?(a):(b))`

Macro to return the maximum of two comparable values.

4.5.16.4.6. MIN

`#define MIN(a, b) ((b)>(a)?(a):(b))`

Macro to return the minimum of two comparable values.

4.5.16.4.7. __check_type_compatible

`#define __check_type_compatible(type_a, type_b) static_assert(__builtin_types_compatible_p(type_a, type_b), __STRING(type_a) " is not compatible with " __STRING(type_b));`

Utility macro to assert two types are equivalent.

This macro can be useful in other macros along with `typeof` to assert that two parameters are of equivalent type (or that a single parameter is of an expected type)

4.5.16.4.8. __after_data

`#define __after_data(group) __attribute__((section(".after_data." group)))`

Section attribute macro for placement in RAM after the `.data` section.

For example a 400 element `uint32_t` array placed after the .data section

```
uint32_t __after_data("my_group_name") a_big_array[400];
```

The section attribute is `.after_data.<group>`

Parameters

`group`	a string suffix to use in the section name to distinguish groups that can be linker garbage-collected independently

4.5.16.4.9. __scratch_x

`#define __scratch_x(group) __attribute__((section(".scratch_x." group)))`

Section attribute macro for placement not in flash (i.e in RAM)

For example a 3 element `uint32_t` array placed in RAM (even though it is `static const`)

```
static const uint32_t __not_in_flash("my_group_name") an_array[3];
```

The section attribute is `.time_critical.<group>`

Parameters

`group` a string suffix to use in the section name to distinguish groups that can be linker garbage-collected independently

Section attribute macro for placement in the SRAM bank 4 (known as "scratch X")

Scratch X is commonly used for critical data and functions accessed only by one core (when only one core is accessing the RAM bank, there is no opportunity for stalls)

For example a `uint32_t` variable placed in "scratch X"

```
uint32_t __scratch_x("my_group_name") foo = 23;
```

The section attribute is `.scratch_x.<group>`

Parameters

`group` a string suffix to use in the section name to distinguish groups that can be linker garbage-collected independently

4.5.16.4.10. __scratch_y

`#define __scratch_y(group) __attribute__((section(".scratch_y." group)))`

Section attribute macro for placement in the SRAM bank 5 (known as "scratch Y")

Scratch Y is commonly used for critical data and functions accessed only by one core (when only one core is accessing the RAM bank, there is no opportunity for stalls)

For example a `uint32_t` variable placed in "scratch Y"

```
uint32_t __scratch_y("my_group_name") foo = 23;
```

The section attribute is `.scratch_y.<group>`

Parameters

`group` a string suffix to use in the section name to distinguish groups that can be linker garbage-collected independently

4.5.16.4.11. __uninitialized_ram

`#define __uninitialized_ram(group) __attribute__((section(".uninitialized_data." #group))) group`

Section attribute macro for data that is to be left uninitialized.

Data marked this way will retain its value across a reset (normally uninitialized data - in the .bss section) is initialized to zero during runtime initialization

For example a `uint32_t` foo that will retain its value if the program is restarted by reset.

```
uint32_t __uninitialized_ram(foo);
```

The section attribute is `.uninitialized_data.<group>`

Parameters

`group` a string suffix to use in the section name to distinguish groups that can be linker garbage-collected independently

4.5.16.4.12. __in_flash

`#define __in_flash(group) __attribute__((section(".flashdata." group)))`

Section attribute macro for placement in flash even in a COPY_TO_RAM binary.

For example a `uint32_t` variable explicitly placed in flash (it will hard fault if you attempt to write it!)

```
uint32_t __in_flash("my_group_name") foo = 23;
```

The section attribute is `.flashdata.<group>`

Parameters

`group` a string suffix to use in the section name to distinguish groups that can be linker garbage-collected independently

4.5.16.4.13. __no_inline_not_in_flash_func

`#define __no_inline_not_in_flash_func(func_name) __noinline __not_in_flash_func(func_name)`

Indicates a function should not be stored in flash.

Decorates a function name, such that the function will execute from RAM (assuming it is not inlined into a flash function by the compiler)

For example a function called my_func taking an int parameter:

```
void __not_in_flash_func(my_func)(int some_arg) {
```

The function is placed in the `.time_critical.<func_name>` linker section

See also

__no_inline_not_in_flash_func

Indicates a function is time/latency critical and should not run from flash

Decorates a function name, such that the function will execute from RAM (assuming it is not inlined into a flash function by the compiler) to avoid possible flash latency. Currently this macro is identical in implementation to `__not_in_flash_func`, however the semantics are distinct and a `__time_critical_func` may in the future be treated more specially to reduce the overhead when calling such function from a flash function.

For example a function called my_func taking an int parameter:

```
void __time_critical_func(my_func)(int some_arg) {
```

The function is placed in the `.time_critical.<func_name>` linker section

See also

__not_in_flash_func

Indicate a function should not be stored in flash and should not be inlined

Decorates a function name, such that the function will execute from RAM, explicitly marking it as noinline to prevent it being inlined into a flash function by the compiler

For example a function called my_func taking an int parameter:

```
void __no_inline_not_in_flash_func(my_func)(int some_arg) {
```

The function is placed in the `.time_critical.<func_name>` linker section

4.5.16.5. Function Documentation

4.5.16.5.1. __breakpoint

`static __force_inline void __breakpoint (void) [static]`

Execute a breakpoint instruction.

4.5.16.5.2. __compiler_memory_barrier

`static __always_inline void __compiler_memory_barrier (void) [static]`

Ensure that the compiler does not move memory access across this method call.

For example in the following code:

```
*some_memory_location = var_a;
__compiler_memory_barrier();
uint32_t var_b = *some_other_memory_location
```

The compiler will not move the load from `some_other_memory_location` above the memory barrier (which it otherwise might - even above the memory store!)

4.5.16.5.3. __get_current_exception

`static __force_inline uint __get_current_exception (void) [static]`

Get the current exception level on this core.

On Cortex-M this is the exception number defined in the architecture reference, which is equal to VTABLE_FIRST_IRQ + irq num if inside an interrupt handler. (VTABLE_FIRST_IRQ is defined in platform_defs.h).

On Hazard3, this function returns VTABLE_FIRST_IRQ + irq num if inside of an external IRQ handler (or a fault from such a handler), and 0 otherwise, generally aligning with the Cortex-M values.

Returns

the exception number if the CPU is handling an exception, or 0 otherwise

4.5.16.5.4. __mul_instruction

`static __force_inline int32_t __mul_instruction (int32_t a, int32_t b) [static]`

Multiply two integers using an assembly `MUL` instruction.

This multiplies a by b using multiply instruction using the ARM mul instruction regardless of values (the compiler might otherwise choose to perform shifts/adds), i.e. this is a 1 cycle operation.

Parameters

`a`	the first operand
`b`	the second operand

Returns

a * b

4.5.16.5.5. busy_wait_at_least_cycles

`static void busy_wait_at_least_cycles (uint32_t minimum_cycles) [inline], [static]`

Helper method to busy-wait for at least the given number of cycles.

This method is useful for introducing very short delays.

This method busy-waits in a tight loop for the given number of system clock cycles. The total wait time is only accurate to within 2 cycles, and this method uses a loop counter rather than a hardware timer, so the method will always take longer than expected if an interrupt is handled on the calling core during the busy-wait; you can of course disable interrupts to prevent this.

You can use clock_get_hz(clk_sys) to determine the number of clock cycles per second if you want to convert an actual time duration to a number of cycles.

Parameters

`minimum_cycles`	the minimum number of system clock cycles to delay for

4.5.16.5.6. get_core_num

`static __force_inline uint get_core_num (void) [static]`

Get the current core number.

Returns

The core number the call was made from

4.5.16.5.7. panic

`void panic (const char * fmt, …)`

Displays a panic message and halts execution.

An attempt is made to output the message to all registered STDOUT drivers after which this method executes a BKPT instruction.

Parameters

`fmt`	format string (printf-like)
`...`	printf-like arguments

4.5.16.5.8. panic_unsupported

`void panic_unsupported (void)`

Panics with the message "Unsupported".

See also

panic

4.5.16.5.9. pico_processor_state_is_nonsecure RP2350

`static __force_inline bool pico_processor_state_is_nonsecure (void) [static]`

Return true if executing in the NonSecure state (Arm-only)

Returns

True if currently executing in the NonSecure state on an Arm processor

4.5.16.5.10. rp2040_chip_version RP2350

`static uint8_t rp2040_chip_version (void) [inline], [static]`

Returns the RP2040 chip revision number for compatibility.

Returns

2 RP2040 errata fixed in B2 are fixed in RP2350

4.5.16.5.11. rp2040_rom_version

`static uint8_t rp2040_rom_version (void) [inline], [static]`

Returns the RP2040 rom version number.

Returns

the RP2040 rom version number (1 for RP2040-B0, 2 for RP2040-B1, 3 for RP2040-B2)

4.5.16.5.12. rp2350_chip_version RP2350

`uint8_t rp2350_chip_version (void)`

Returns the RP2350 chip revision number.

Returns

the RP2350 chip revision number (1 for B0/B1, 2 for B2)

4.5.16.5.13. tight_loop_contents

`static __force_inline void tight_loop_contents (void) [static]`

No-op function for the body of tight loops.

No-op function intended to be called by any tight hardware polling loop. Using this ubiquitously makes it much easier to find tight loops, but also in the future #ifdef-ed support for lockup debugging might be added

4.5.17. pico_printf

Compact replacement for printf by Marco Paland (info@paland.com)

4.5.18. pico_runtime

Basic runtime support for running pre-main initializers provided by other libraries.

4.5.18.1. Detailed Description

This library aggregates the following other libraries (if available):

- hardware_uart
- pico_bit_ops
- pico_divider
- pico_double
- pico_int64_ops
- pico_float
- pico_malloc
- pico_mem_ops
- pico_atomic
- pico_cxx_options
- pico_standard_binary_info
- pico_standard_link
- pico_sync
- pico_printf
- pico_crt0
- pico_clib_interface
- pico_stdio

4.5.18.2. Functions

`void runtime_init (void)`

Run all the initializations that are usually called by crt0.S before entering main.

4.5.18.3. Function Documentation

4.5.18.3.1. runtime_init

`void runtime_init (void)`

Run all the initializations that are usually called by crt0.S before entering main.

This method is useful to set up the runtime after performing a watchdog or powman reboot via scratch vector.

4.5.19. pico_runtime_init

Main runtime initialization functions required to set up the runtime environment before entering main.

4.5.19.1. Detailed Description

The runtime initialization is registration based:

For each step of the initialization there is a 5 digit ordinal which indicates the ordering (alphabetic increasing sort of the 5 digits) of the steps.

e.g. for the step "bootrom_reset", there is:

```
#ifndef PICO_RUNTIME_INIT_BOOTROM_RESET
#define PICO_RUNTIME_INIT_BOOTROM_RESET   "00050"
#endif
```

The user can override the order if they wish, by redefining PICO_RUNTIME_INIT_BOOTROM_RESET

For each step, the automatic initialization may be skipped by defining (in this case) PICO_RUNTIME_SKIP_INIT_BOOTROM_RESET = 1. The user can then choose to either omit the step completely or register their own replacement initialization.

The default method used to perform the initialization is provided, in case the user wishes to call it manually; in this case:

```
void runtime_init_bootrom_reset(void);
```

If PICO_RUNTIME_NO_INIT_BOOTOROM_RESET define is set (NO vs SKIP above), then the function is not defined, allowing the user to provide a replacement (and also avoiding cases where the default implementation won't compile due to missing dependencies)

4.5.19.2. Functions

`static void clocks_init (void)`

Initialise the clock hardware.

4.5.19.3. Function Documentation

4.5.19.3.1. clocks_init

`static void clocks_init (void) [inline], [static]`

Initialise the clock hardware.

Must be called before any other clock function.

4.5.20. pico_stdio

Customized stdio support allowing for input and output from UART, USB, semi-hosting etc.

4.5.20.1. Detailed Description

Note the API for adding additional input output devices is not yet considered stable

4.5.20.2. Modules

pico_stdio_semihosting

Experimental support for stdout using RAM semihosting.

pico_stdio_uart

Support for stdin/stdout using UART.

pico_stdio_rtt

Support for stdin/stdout using SEGGER RTT.

pico_stdio_usb

Support for stdin/stdout over USB serial (CDC)

4.5.20.3. Functions

`bool stdio_init_all (void)`

Initialize all of the present standard stdio types that are linked into the binary.

`bool stdio_deinit_all (void)`

Deinitialize all of the present standard stdio types that are linked into the binary.

`void stdio_flush (void)`

Flushes any buffered output.

`int stdio_getchar_timeout_us (uint32_t timeout_us)`

Return a character from stdin if there is one available within a timeout.

`static int getchar_timeout_us (uint32_t timeout_us)`

Alias for stdio_getchar_timeout_us for backwards compatibility.

`void stdio_set_driver_enabled (stdio_driver_t *driver, bool enabled)`

Adds or removes a driver from the list of active drivers used for input/output.

`void stdio_filter_driver (stdio_driver_t *driver)`

Control limiting of output to a single driver.

`void stdio_set_translate_crlf (stdio_driver_t *driver, bool translate)`

control conversion of line feeds to carriage return on transmissions

`int stdio_putchar_raw (int c)`

putchar variant that skips any CR/LF conversion if enabled

`static int putchar_raw (int c)`

Alias for stdio_putchar_raw for backwards compatibility.

`int stdio_puts_raw (const char *s)`

puts variant that skips any CR/LF conversion if enabled `static int puts_raw (const char *s)`

Alias for stdio_puts_raw for backwards compatibility.

`void stdio_set_chars_available_callback (void(*fn)(void *), void *param)`

get notified when there are input characters available

`int stdio_get_until (char *buf, int len, absolute_time_t until)`

Waits until a timeout to reard at least one character into a buffer.

`int stdio_put_string (const char *s, int len, bool newline, bool cr_translation)`

Prints a buffer to stdout with optional newline and carriage return insertion.

`int stdio_getchar (void)`

stdio_getchar Alias for getchar that definitely does not go thru the implementation in the standard C library even when PICO_STDIO_SHORT_CIRCUIT_CLIB_FUNCS == 0

`int stdio_putchar (int)`

stdio_getchar Alias for putchar that definitely does not go thru the implementation in the standard C library even when PICO_STDIO_SHORT_CIRCUIT_CLIB_FUNCS == 0

`int stdio_puts (const char *s)`

stdio_getchar Alias for puts that definitely does not go thru the implementation in the standard C library even when PICO_STDIO_SHORT_CIRCUIT_CLIB_FUNCS == 0

`int stdio_vprintf (const char *format, va_list va)`

stdio_getchar Alias for vprintf that definitely does not go thru the implementation in the standard C library even when PICO_STDIO_SHORT_CIRCUIT_CLIB_FUNCS == 0

`int __printflike (1, 0) stdio_printf(const char *format`

stdio_getchar Alias for printf that definitely does not go thru the implementation in the standard C library even when PICO_STDIO_SHORT_CIRCUIT_CLIB_FUNCS == 0

4.5.20.4. Function Documentation

4.5.20.4.1. __printflike

`int __printflike (1, 0)`

stdio_getchar Alias for printf that definitely does not go thru the implementation in the standard C library even when PICO_STDIO_SHORT_CIRCUIT_CLIB_FUNCS == 0

4.5.20.4.2. getchar_timeout_us

`static int getchar_timeout_us (uint32_t timeout_us) [inline], [static]`

Alias for stdio_getchar_timeout_us for backwards compatibility.

4.5.20.4.3. putchar_raw

`static int putchar_raw (int c) [inline], [static]`

Alias for stdio_putchar_raw for backwards compatibility.

4.5.20.4.4. puts_raw

`static int puts_raw (const char * s) [inline], [static]`

Alias for stdio_puts_raw for backwards compatibility.

4.5.20.4.5. stdio_deinit_all

`bool stdio_deinit_all (void)`

Deinitialize all of the present standard stdio types that are linked into the binary.

This method currently only supports stdio_uart and stdio_semihosting

Returns

true if all outputs was successfully deinitialized, false otherwise.

See also

stdio_uart, stdio_usb, stdio_semihosting, stdio_rtt

4.5.20.4.6. stdio_filter_driver

`void stdio_filter_driver (stdio_driver_t * driver)`

Control limiting of output to a single driver.

> **NOTE**
>
> this method should always be called on an initialized driver

Parameters

`driver`	if non-null then output only that driver will be used for input/output (assuming it is in the list of enabled drivers). if NULL then all enabled drivers will be used

4.5.20.4.7. stdio_flush

`void stdio_flush (void)`

Flushes any buffered output.

4.5.20.4.8. stdio_get_until

`int stdio_get_until (char * buf, int len, absolute_time_t until)`

Waits until a timeout to reard at least one character into a buffer.

This method returns as soon as input is available, but more characters may be returned up to the end of the buffer.

Parameters

`buf`	the buffer to read into
`len`	the length of the buffer

Returns

the number of characters read or PICO_ERROR_TIMEOUT

Parameters

`until`	the time after which to return PICO_ERROR_TIMEOUT if no characters are available

4.5.20.4.9. stdio_getchar

`int stdio_getchar (void)`

stdio_getchar Alias for getchar that definitely does not go thru the implementation in the standard C library even when PICO_STDIO_SHORT_CIRCUIT_CLIB_FUNCS == 0

4.5.20.4.10. stdio_getchar_timeout_us

`int stdio_getchar_timeout_us (uint32_t timeout_us)`

Return a character from stdin if there is one available within a timeout.

Parameters

`timeout_us`	the timeout in microseconds, or 0 to not wait for a character if none available.

Returns

the character from 0-255 or PICO_ERROR_TIMEOUT if timeout occurs

4.5.20.4.11. stdio_init_all

`bool stdio_init_all (void)`

Initialize all of the present standard stdio types that are linked into the binary.

Call this method once you have set up your clocks to enable the stdio support for UART, USB, semihosting, and RTT based on the presence of the respective libraries in the binary.

When stdio_usb is configured, this method can be optionally made to block, waiting for a connection via the variables specified in stdio_usb_init (i.e. PICO_STDIO_USB_CONNECT_WAIT_TIMEOUT_MS)

Returns

true if at least one output was successfully initialized, false otherwise.

See also

stdio_uart, stdio_usb, stdio_semihosting, stdio_rtt

4.5.20.4.12. stdio_put_string

`int stdio_put_string (const char * s, int len, bool newline, bool cr_translation)`

Prints a buffer to stdout with optional newline and carriage return insertion.

This method returns as soon as input is available, but more characters may be returned up to the end of the buffer.

Parameters

`s`	the characters to print
`len`	the length of s
`newline`	true if a newline should be added after the string
`cr_translation`	true if line feed to carriage return translation should be performed

Returns

the number of characters written

4.5.20.4.13. stdio_putchar

`int stdio_putchar (int c)`

stdio_getchar Alias for putchar that definitely does not go thru the implementation in the standard C library even when PICO_STDIO_SHORT_CIRCUIT_CLIB_FUNCS == 0

4.5.20.4.14. stdio_putchar_raw

`int stdio_putchar_raw (int c)`

putchar variant that skips any CR/LF conversion if enabled

4.5.20.4.15. stdio_puts

`int stdio_puts (const char * s)`

stdio_getchar Alias for puts that definitely does not go thru the implementation in the standard C library even when PICO_STDIO_SHORT_CIRCUIT_CLIB_FUNCS == 0

4.5.20.4.16. stdio_puts_raw

`int stdio_puts_raw (const char * s)`

puts variant that skips any CR/LF conversion if enabled

4.5.20.4.17. stdio_set_chars_available_callback

`void stdio_set_chars_available_callback (void(*)(void *) fn, void * param)`

get notified when there are input characters available

Parameters

`fn`	Callback function to be called when characters are available. Pass NULL to cancel any existing callback
`param`	Pointer to pass to the callback

4.5.20.4.18. stdio_set_driver_enabled

`void stdio_set_driver_enabled (stdio_driver_t * driver, bool enabled)`

Adds or removes a driver from the list of active drivers used for input/output.

NOTE

> this method should always be called on an initialized driver and is not re-entrant

Parameters

`driver`	the driver
`enabled`	true to add, false to remove

4.5.20.4.19. stdio_set_translate_crlf

`void stdio_set_translate_crlf (stdio_driver_t * driver, bool translate)` control conversion of line feeds to carriage return on transmissions

NOTE

> this method should always be called on an initialized driver

Parameters

`driver`	the driver
`translate`	If true, convert line feeds to carriage return on transmissions

4.5.20.4.20. stdio_vprintf

`int stdio_vprintf (const char * format, va_list va)`

stdio_getchar Alias for vprintf that definitely does not go thru the implementation in the standard C library even when PICO_STDIO_SHORT_CIRCUIT_CLIB_FUNCS == 0

4.5.20.5. pico_stdio_semihosting

Experimental support for stdout using RAM semihosting.

4.5.20.5.1. Detailed Description

Linking this library or calling `pico_enable_stdio_semihosting(TARGET ENABLED)` in the CMake (which achieves the same thing) will add semihosting to the drivers used for standard output

4.5.20.5.2. Functions

`void stdio_semihosting_init (void)`

Explicitly initialize stdout over semihosting and add it to the current set of stdout targets.

`void stdio_semihosting_deinit (void)`

Explicitly deinitialize stdout over semihosting and add it to the current set of stdout targets.

4.5.20.5.3. Function Documentation

stdio_semihosting_deinit

`void stdio_semihosting_deinit (void)`

Explicitly deinitialize stdout over semihosting and add it to the current set of stdout targets.

NOTE

> this method is automatically called by stdio_deinit_all() if `pico_stdio_semihosting` is included in the build

stdio_semihosting_init

`void stdio_semihosting_init (void)`

Explicitly initialize stdout over semihosting and add it to the current set of stdout targets.

> **NOTE**
>
> this method is automatically called by stdio_init_all() if `pico_stdio_semihosting` is included in the build

4.5.20.6. pico_stdio_uart

Support for stdin/stdout using UART.

4.5.20.6.1. Detailed Description

Linking this library or calling `pico_enable_stdio_uart(TARGET ENABLED)` in the CMake (which achieves the same thing) will add UART to the drivers used for standard input/output

4.5.20.6.2. Functions

`void stdio_uart_init (void)`

Explicitly initialize stdin/stdout over UART and add it to the current set of stdin/stdout drivers.

`void stdout_uart_init (void)`

Explicitly initialize stdout only (no stdin) over UART and add it to the current set of stdout drivers.

`void stdin_uart_init (void)`

Explicitly initialize stdin only (no stdout) over UART and add it to the current set of stdin drivers.

`void stdio_uart_init_full (uart_inst_t *uart, uint baud_rate, int tx_pin, int rx_pin)`

Perform custom initialization initialize stdin/stdout over UART and add it to the current set of stdin/stdout drivers.

`void stdio_uart_deinit (void)`

Explicitly deinitialize stdin/stdout over UART and remove it from the current set of stdin/stdout drivers.

`void stdout_uart_deinit (void)`

Explicitly deinitialize stdout only (no stdin) over UART and remove it from the current set of stdout drivers.

`void stdin_uart_deinit (void)`

Explicitly deinitialize stdin only (no stdout) over UART and remove it from the current set of stdin drivers.

`void stdio_uart_deinit_full (uart_inst_t *uart, int tx_pin, int rx_pin)`

Perform custom deinitialization deinitialize stdin/stdout over UART and remove it from the current set of stdin/stdout drivers.

4.5.20.6.3. Function Documentation

stdin_uart_deinit

```
void stdin_uart_deinit (void)
```

Explicitly deinitialize stdin only (no stdout) over UART and remove it from the current set of stdin drivers.

This method disables PICO_DEFAULT_UART_RX_PIN for UART input (if defined), and leaves the pads isolated

stdin_uart_init

```
void stdin_uart_init (void)
```

Explicitly initialize stdin only (no stdout) over UART and add it to the current set of stdin drivers.

This method sets up PICO_DEFAULT_UART_RX_PIN for UART input (if defined) , and configures the baud rate as

PICO_DEFAULT_UART_BAUD_RATE

stdio_uart_deinit

```
void stdio_uart_deinit (void)
```

Explicitly deinitialize stdin/stdout over UART and remove it from the current set of stdin/stdout drivers.

This method disables PICO_DEFAULT_UART_TX_PIN for UART output (if defined), PICO_DEFAULT_UART_RX_PIN for input (if defined) and leaves the pads isolated.

NOTE

> this method is automatically called by stdio_deinit_all() if `pico_stdio_uart` is included in the build

stdio_uart_deinit_full

```
void stdio_uart_deinit_full (uart_inst_t * uart, int tx_pin, int rx_pin)
```

Perform custom deinitialization deinitialize stdin/stdout over UART and remove it from the current set of stdin/stdout drivers.

Parameters

`uart`	the uart instance to use, uart0 or uart1
`tx_pin`	the UART pin to use for stdout (or -1 for no stdout)
`rx_pin`	the UART pin to use for stdin (or -1 for no stdin)

stdio_uart_init

```
void stdio_uart_init (void)
```

Explicitly initialize stdin/stdout over UART and add it to the current set of stdin/stdout drivers.

This method sets up PICO_DEFAULT_UART_TX_PIN for UART output (if defined), PICO_DEFAULT_UART_RX_PIN for input (if defined) and configures the baud rate as PICO_DEFAULT_UART_BAUD_RATE.

NOTE

> this method is automatically called by stdio_init_all() if `pico_stdio_uart` is included in the build

stdio_uart_init_full

```
void stdio_uart_init_full (uart_inst_t * uart, uint baud_rate, int tx_pin, int rx_pin)
```

Perform custom initialization initialize stdin/stdout over UART and add it to the current set of stdin/stdout drivers.

Parameters

`uart`	the uart instance to use, uart0 or uart1
`baud_rate`	the baud rate in Hz
`tx_pin`	the UART pin to use for stdout (or -1 for no stdout)
`rx_pin`	the UART pin to use for stdin (or -1 for no stdin)

stdout_uart_deinit

```
void stdout_uart_deinit (void)
```

Explicitly deinitialize stdout only (no stdin) over UART and remove it from the current set of stdout drivers.

This method disables PICO_DEFAULT_UART_TX_PIN for UART output (if defined), and leaves the pad isolated

stdout_uart_init

```
void stdout_uart_init (void)
```

Explicitly initialize stdout only (no stdin) over UART and add it to the current set of stdout drivers.

This method sets up PICO_DEFAULT_UART_TX_PIN for UART output (if defined) , and configures the baud rate as PICO_DEFAULT_UART_BAUD_RATE

4.5.20.7. pico_stdio_rtt

Support for stdin/stdout using SEGGER RTT.

4.5.20.7.1. Detailed Description

Linking this library or calling `pico_enable_stdio_rtt(TARGET)` in the CMake (which achieves the same thing) will add RTT to the drivers used for standard output

4.5.20.7.2. Functions

`void stdio_rtt_init (void)`

Explicitly initialize stdin/stdout over RTT and add it to the current set of stdin/stdout drivers.

`void stdio_rtt_deinit (void)`

Explicitly deinitialize stdin/stdout over RTT and remove it from the current set of stdin/stdout drivers.

4.5.20.7.3. Function Documentation

stdio_rtt_deinit

`void stdio_rtt_deinit (void)`

Explicitly deinitialize stdin/stdout over RTT and remove it from the current set of stdin/stdout drivers.

NOTE

> this method is automatically called by stdio_deinit_all() if `pico_stdio_rtt` is included in the build

stdio_rtt_init

`void stdio_rtt_init (void)`

Explicitly initialize stdin/stdout over RTT and add it to the current set of stdin/stdout drivers.

NOTE

> this method is automatically called by stdio_init_all() if `pico_stdio_rtt` is included in the build

4.5.20.8. pico_stdio_usb

Support for stdin/stdout over USB serial (CDC)

4.5.20.8.1. Detailed Description

Linking this library or calling `pico_enable_stdio_usb(TARGET ENABLED)` in the CMake (which achieves the same thing) will add USB CDC to the drivers used for standard input/output

Note this library is a developer convenience. It is not applicable in all cases; for one it takes full control of the USB device precluding your use of the USB in device or host mode. For this reason, this library will automatically disengage if you try to using it alongside tinyusb_device or tinyusb_host. It also takes control of a lower level IRQ and sets up a periodic background task.

This library also includes (by default) functionality to enable the RP-series microcontroller to be reset over the USB interface.

4.5.20.8.2. Functions

`bool stdio_usb_init (void)`

Explicitly initialize USB stdio and add it to the current set of stdin drivers.

`bool stdio_usb_deinit (void)`

Explicitly deinitialize USB stdio and remove it from the current set of stdin drivers.

`bool stdio_usb_connected (void)`

Check if there is an active stdio CDC connection to a host.

4.5.20.8.3. Function Documentation

stdio_usb_connected

`bool stdio_usb_connected (void)`

Check if there is an active stdio CDC connection to a host.

Returns

true if stdio is connected over CDC

stdio_usb_deinit

`bool stdio_usb_deinit (void)`

Explicitly deinitialize USB stdio and remove it from the current set of stdin drivers.

Returns

true if the USB CDC was deinitialized, false if an error occurred

stdio_usb_init

`bool stdio_usb_init (void)`

Explicitly initialize USB stdio and add it to the current set of stdin drivers.

PICO_STDIO_USB_CONNECT_WAIT_TIMEOUT_MS can be set to cause this method to wait for a CDC connection from the host before returning, which is useful if you don't want any initial stdout output to be discarded before the connection is established.

Returns

true if the USB CDC was initialized, false if an error occurred

4.5.21. pico_standard_binary_info

Includes default information about the binary that can be displayed by picotool.

4.5.21.1. Detailed Description

Information is included only if PICO_NO_BINARY_INFO and PICO_NO_PROGRAM_INFO are both false.

This library adds the following information to the binary:

- The program name if defined (unless PICO_NO_BINARY_SIZE=1). The value is PICO_PROGRAM_NAME or PICO_TARGET_NAME if the former isn't defined
- The value of PICO_BOARD (unless PICO_NO_BI_PICO_BOARD=1)
- The SDK version (unless PICO_NO_BI_SDK_VERSION=1)
- The program version string if defined (unless PICO_NO_BI_PROGRAM_VERSION_STRING=1). The value is PICO_PROGRAM_VERSION_STRING
- The program description if defined (unless PICO_NO_BI_PROGRAM_DESCRIPTION=1). The value is PICO_PROGRAM_DESCRIPTION
- The program url if defined (unless PICO_NO_BI_PROGRAM_URL=1). The value is PICO_PROGRAM_URL
- The boot stage 2 used if any (unless PICO_NO_BI_BOOT_STAGE2_NAME=1). The value is PICO_BOOT_STAGE2_NAME
- The program build date (unless PICO_NO_BI_PROGRAM_BUILD_DATE=1). The value defaults to the C preprocessor value __DATE__, but can be overridden with PICO_PROGRAM_BUILD_DATE. Note you should do a clean build if you want to be sure this value is up to date.
- The program build type (unless PICO_NO_BI_BUILD_TYPE=1). The value is PICO_CMAKE_BUILD_TYPE which comes from the CMake build - e.g. Release, Debug, RelMinSize
- The binary size (unless PICO_NO_BI_BINARY_SIZE=1)

4.5.22. pico_standard_link

Setup for link options for a standard SDK executable.

4.5.22.1. Detailed Description

This includes

- C runtime initialization
- Linker scripts for 'default', 'no_flash', 'blocked_ram' and 'copy_to_ram' binaries
- 'Binary Information' support
- Linker option control

4.6. External API Headers

Headers for interfaces that are shared with code outside of the SDK

boot_picobin_headers	Constants for PICOBIN format.
boot_picoboot_headers	Header file for the PICOBOOT USB interface exposed by an RP2xxx chip in BOOTSEL mode.
boot_uf2_headers	Header file for the UF2 format supported by a RP2xxx chip in BOOTSEL mode.
pico_usb_reset_interface_headers	Definition for the reset interface that may be exposed by the pico_stdio_usb library.

4.6.1. boot_picobin_headers

Constants for PICOBIN format.

4.6.2. boot_picoboot_headers

Header file for the PICOBOOT USB interface exposed by an RP2xxx chip in BOOTSEL mode.

4.6.3. boot_uf2_headers

Header file for the UF2 format supported by a RP2xxx chip in BOOTSEL mode.

4.6.4. pico_usb_reset_interface_headers

Definition for the reset interface that may be exposed by the pico_stdio_usb library.

Chapter 5. SDK configuration

SDK configuration is the process of customising the SDK code for your particular build/application. As the parts of the SDK that you use are recompiled as part of your build, configuration options can be chosen at compile time resulting in smaller and more efficient customized versions of the code.

This chapter will show what configuration parameters are available, and how they can be changed.

SDK configuration parameters are passed as C preprocessor definitions to the build. The most common way to override them is to specify them in your `CMakeLists.txt` when you define your executable or library:

e.g.

```
add_executable(my_program main.c)
...
target_compile_definitions(my_program PRIVATE
    PICO_STACK_SIZE=4096
)
```

or if you are creating a library, and you want to add compile definitions whenever your library is included:

```
add_library(my_library INTERFACE)
...
target_compile_definitions(my_library INTERFACE
    PICO_STDIO_DEFAULT_CRLF=0
    PICO_DEFAULT_UART=1
)
```

The definitions can also be specified in header files, as is commonly done for board configuration (see Chapter 8).

For example,. the Pimoroni Tiny2040 board header configures the following to specify appropriate board settings for the default I2C channel exposed on that board.

```
// --- I2C ---
#ifndef PICO_DEFAULT_I2C
#define PICO_DEFAULT_I2C 1
#endif
#ifndef PICO_DEFAULT_I2C_SDA_PIN
#define PICO_DEFAULT_I2C_SDA_PIN 2
#endif
#ifndef PICO_DEFAULT_I2C_SCL_PIN
#define PICO_DEFAULT_I2C_SCL_PIN 3
#endif
```

NOTE

The `#ifdef` still allows the values to be by the build (i.e. in `CMakeLists.txt`)

If you would rather set values in your own header file rather than via `CMake`, then you must make sure the header is included by all compilation (including the SDK sources). Using a custom `PICO_BOARD` header is one way of doing this, but a more advanced way is to have the SDK include your header via `pico/config.h` which itself is included by every SDK source file.

This can be done by adding the following before the `pico_sdk_init()` in your `CMakeLists.txt`:

```
list(APPEND PICO_CONFIG_HEADER_FILES path/to/your/header.h)
```

5.1. Full List of SDK Configuration Defines

Table 34. SDK and Board Configuration Defines

Name / Description	Default
CYW43_DEFAULT_PIN_WL_CLOCK gpio pin for the spi clock line to the cyw43 chip	
CYW43_DEFAULT_PIN_WL_CS gpio pin for the spi chip select to the cyw43 chip	
CYW43_DEFAULT_PIN_WL_DATA_IN gpio pin for spi data in from the cyw43 chip	
CYW43_DEFAULT_PIN_WL_DATA_OUT gpio pin for spi data out to the cyw43 chip	
CYW43_DEFAULT_PIN_WL_HOST_WAKE gpio (irq) pin for the irq line from the cyw43 chip	
CYW43_DEFAULT_PIN_WL_REG_ON gpio pin to power up the cyw43 chip	
CYW43_PIN_WL_DYNAMIC flag to indicate if cyw43 SPI pins can be changed at runtime	
CYW43_PIO_CLOCK_DIV_DYNAMIC Enable runtime configuration of the clock divider for communication with the wireless chip	0
CYW43_PIO_CLOCK_DIV_FRAC8 Fractional part of the clock divider for communication with the wireless chip 0-255	0
CYW43_PIO_CLOCK_DIV_INT Integer part of the clock divider for communication with the wireless chip	2
CYW43_TASK_PRIORITY Priority for the CYW43 FreeRTOS task	tskIDLE_PRIORITY + 4
CYW43_TASK_STACK_SIZE Stack size for the CYW43 FreeRTOS task in 4-byte words	1024
GPIO_IRQ_CALLBACK_ORDER_PRIORITY IRQ priority order of the default IRQ callback	PICO_SHARED_IRQ_HANDLER_LOWEST_ORDER_PRIORITY
GPIO_RAW_IRQ_HANDLER_DEFAULT_ORDER_PRIORITY IRQ priority order of raw IRQ handlers if the priority is not specified	PICO_SHARED_IRQ_HANDLER_DEFAULT_ORDER_PRIORITY
PARAM_ASSERTIONS_DISABLE_ALL Global assert disable	0
PARAM_ASSERTIONS_ENABLED_ADDRESS_ALIAS Enable/disable assertions in memory address aliasing macros	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_ADC Enable/disable assertions in the hardware_adc module	0

Name / Description	Default
PARAM_ASSERTIONS_ENABLED_HARDWARE_BOOT_LOCK Enable/disable assertions in the hardware_boot_lock module	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_CLOCKS Enable/disable assertions in the hardware_clocks module	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_DMA Enable/disable hardware_dma assertions	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_EXCEPTION Enable/disable assertions in the hardware_exception module	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_FLASH Enable/disable assertions in the hardware_flash module	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_GPIO Enable/disable assertions in the hardware_gpio module	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_I2C Enable/disable assertions in the hardware_i2c module	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_INTERP Enable/disable assertions in the hardware_interp module	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_IRQ Enable/disable assertions in the hardware_irq module	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_PIO Enable/disable assertions in the hardware_pio module	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_POWMAN Enable/disable hardware_powman assertions	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_PWM Enable/disable assertions in the hardware_pwm module	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_RESETS Enable/disable assertions in the hardware_resets module	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_SHA256 Enable/disable hardware_sha256 assertions	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_SPI Enable/disable assertions in the hardware_spi module	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_SYNC Enable/disable assertions in the hardware_sync module	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_TICKS Enable/disable assertions in the hardware_ticks module	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_TIMER Enable/disable assertions in the hardware_timer module	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_UART Enable/disable assertions in the hardware_uart module	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_WATCHDOG Enable/disable assertions in the hardware_watchdog module	0
PARAM_ASSERTIONS_ENABLED_HARDWARE_XIP_CACHE Enable/disable assertions in the hardware_xip_cache module	0

Name / Description	Default
PARAM_ASSERTIONS_ENABLED_LOCK_CORE Enable/disable assertions in the lock core	0
PARAM_ASSERTIONS_ENABLED_PHEAP Enable/disable assertions in the pheap module	0
PARAM_ASSERTIONS_ENABLED_PICO_CYW43_ARCH Enable/disable assertions in the pico_cyw43_arch module	0
PARAM_ASSERTIONS_ENABLED_PICO_MULTICORE Enable/disable assertions in the pico_multicore module	0
PARAM_ASSERTIONS_ENABLED_PICO_TIME Enable/disable assertions in the pico_time module	0
PARAM_ASSERTIONS_ENABLED_PIO_INSTRUCTIONS Enable/disable assertions in the PIO instructions	0
PARAM_ASSERTIONS_ENABLE_ALL Global assert enable	0
PICO_ADC_CLKDIV_ROUND_NEAREST True if floating point ADC clock divisors should be rounded to the nearest possible clock divisor rather than rounding down	PICO_CLKDIV_ROUND_NEAREST
PICO_BOOTROM_LOCKING_ENABLED Enable/disable locking for bootrom functions that use shared resources. If this flag is enabled bootrom lock checking is turned on and BOOT locks are taken around the relevant bootrom functions	1
PICO_BOOTSEL_VIA_DOUBLE_RESET_ACTIVITY_LED Optionally define a pin to use as bootloader activity LED when BOOTSEL mode is entered via reset double tap	
PICO_BOOTSEL_VIA_DOUBLE_RESET_ACTIVITY_LED_ACTIVE_LOW Whether pin used as bootloader activity LED when BOOTSEL mode is entered via reset double tap is active low. Not supported on RP2040	0
PICO_BOOTSEL_VIA_DOUBLE_RESET_INTERFACE_DISABLE_MASK Optionally disable either the mass storage interface (bit 0) or the PICOBOOT interface (bit 1) when entering BOOTSEL mode via double reset	0
PICO_BOOTSEL_VIA_DOUBLE_RESET_TIMEOUT_MS Window of opportunity for a second press of a reset button to enter BOOTSEL mode (milliseconds)	200
PICO_BOOT_STAGE2_CHOOSE_AT25SF128A Select boot2_at25sf128a as the boot stage 2 when no boot stage 2 selection is made by the CMake build	0
PICO_BOOT_STAGE2_CHOOSE_GENERIC_03H Select boot2_generic_03h as the boot stage 2 when no boot stage 2 selection is made by the CMake build	1
PICO_BOOT_STAGE2_CHOOSE_IS25LP080 Select boot2_is25lp080 as the boot stage 2 when no boot stage 2 selection is made by the CMake build	0
PICO_BOOT_STAGE2_CHOOSE_W25Q080 Select boot2_w25q080 as the boot stage 2 when no boot stage 2 selection is made by the CMake build	0

Name / Description	Default
PICO_BOOT_STAGE2_CHOOSE_W25X10CL Select boot2_w25x10cl as the boot stage 2 when no boot stage 2 selection is made by the CMake build	0
PICO_BTSTACK_CYW43_MAX_HCI_PROCESS_LOOP_COUNT limit the max number of iterations of the hci processing loop	
PICO_BUILD_BOOT_STAGE2_NAME Name of the boot stage 2 if selected in the build system	
PICO_CLKDIV_ROUND_NEAREST True if floating point clock divisors should be rounded to the nearest possible clock divisor by default rather than rounding down	1
PICO_CLOCK_ADJUST_PERI_CLOCK_WITH_SYS_CLOCK When the SYS clock PLL is changed keep the peripheral clock attached to it	0
PICO_CLOCK_GPIO_CLKDIV_ROUND_NEAREST True if floating point GPIO clock divisors should be rounded to the nearest possible clock divisor rather than rounding down	PICO_CLKDIV_ROUND_NEAREST
PICO_CMSIS_RENAME_EXCEPTIONS Whether to rename SDK exceptions such as isr_nmi to their CMSIS equivalent i.e. NMI_Handler	1
PICO_CONFIG_HEADER Unquoted path to header include in place of the default pico/config.h which may be desirable for build systems which can't easily generate the config_autogen header	
PICO_CONFIG_RTOS_ADAPTER_HEADER Unquoted path to header include in the default pico/config.h for RTOS integration defines that must be included in all sources	
PICO_CORE1_STACK_SIZE Minimum amount of stack space reserved in the linker script for core 1 - if zero then no space is reserved and the user must provide their own stack	PICO_STACK_SIZE (0x800)
PICO_CYW43_ARCH_DEBUG_ENABLED Enable/disable some debugging output in the pico_cyw43_arch module	1 in debug builds
PICO_CYW43_ARCH_DEFAULT_COUNTRY_CODE Default country code for the cyw43 wireless driver	CYW43_COUNTRY_WORLDWIDE
PICO_DEBUG_MALLOC Enable/disable debug printf from malloc	0
PICO_DEBUG_MALLOC_LOW_WATER Define the lower bound for allocation addresses to be printed by PICO_DEBUG_MALLOC	0
PICO_DEBUG_PIN_BASE First pin to use for debug output (if enabled)	19
PICO_DEBUG_PIN_COUNT Number of pins to use for debug output (if enabled)	3
PICO_DEFAULT_I2C Define the default I2C for a board	Usually provided via board header
PICO_DEFAULT_I2C_SCL_PIN Define the default I2C SCL pin	Usually provided via board header

Name / Description	Default
PICO_DEFAULT_I2C_SDA_PIN Define the default I2C SDA pin	Usually provided via board header
PICO_DEFAULT_IRQ_PRIORITY Define the default IRQ priority	0x80
PICO_DEFAULT_LED_PIN Optionally define a pin that drives a regular LED on the board	Usually provided via board header
PICO_DEFAULT_LED_PIN_INVERTED 1 if LED is inverted or 0 if not	0
PICO_DEFAULT_SPI Define the default SPI for a board	Usually provided via board header
PICO_DEFAULT_SPI_CSN_PIN Define the default SPI CSN pin	Usually provided via board header
PICO_DEFAULT_SPI_RX_PIN Define the default SPI RX pin	Usually provided via board header
PICO_DEFAULT_SPI_SCK_PIN Define the default SPI SCK pin	Usually provided via board header
PICO_DEFAULT_SPI_TX_PIN Define the default SPI TX pin	Usually provided via board header
PICO_DEFAULT_TIMER Timer instance number to use for RP2040-period hardware_timer APIs that assumed a single timer instance	0
PICO_DEFAULT_UART Define the default UART used for printf etc	Usually provided via board header
PICO_DEFAULT_UART_BAUD_RATE Define the default UART baudrate	115200
PICO_DEFAULT_UART_RX_PIN Define the default UART RX pin	Usually provided via board header
PICO_DEFAULT_UART_TX_PIN Define the default UART TX pin	Usually provided via board header
PICO_DEFAULT_WS2812_PIN Optionally define a pin that controls data to a WS2812 compatible LED on the board	
PICO_DEFAULT_WS2812_POWER_PIN Optionally define a pin that controls power to a WS2812 compatible LED on the board	
PICO_DISABLE_SHARED_IRQ_HANDLERS Disable shared IRQ handlers	0
PICO_DOUBLE_SUPPORT_ROM_V1 RP2040 Include double support code for RP2040 B0 when that chip revision is supported	1
PICO_EMBED_XIP_SETUP Embed custom XIP setup (boot2) in an RP2350 binary	0

Name / Description	Default
PICO_FLASH_ASSERT_ON_UNSAFE Assert in debug mode rather than returning an error if flash_safe_execute cannot guarantee safety to catch bugs early	1
PICO_FLASH_ASSUME_CORE0_SAFE Assume that core 0 will never be accessing flash and so doesn't need to be considered during flash_safe_execute	0
PICO_FLASH_ASSUME_CORE1_SAFE Assume that core 1 will never be accessing flash and so doesn't need to be considered during flash_safe_execute	0
PICO_FLASH_BANK_STORAGE_OFFSET Offset in flash of the Bluetooth flash storage	PICO_FLASH_SIZE_BYTES - PICO_FLASH_BANK_TOTAL_SIZE
PICO_FLASH_BANK_TOTAL_SIZE Total size of the Bluetooth flash storage. Must be an even multiple of FLASH_SECTOR_SIZE	FLASH_SECTOR_SIZE * 2
PICO_FLASH_SAFE_EXECUTE_PICO_SUPPORT_MULTICORE_LOCKOUT Support using multicore_lockout functions to make the other core safe during flash_safe_execute	1 when using pico_multicore
PICO_FLASH_SAFE_EXECUTE_SUPPORT_FREERTOS_SMP Support using FreeRTOS SMP to make the other core safe during flash_safe_execute	1 when using FreeRTOS SMP
PICO_FLASH_SIZE_BYTES size of primary flash in bytes	Usually provided via board header
PICO_FLOAT_SUPPORT_ROM_V1 RP2040 Include float support code for RP2040 B0 when that chip revision is supported	1
PICO_HEAP_SIZE Minimum amount of heap space reserved by the linker script	0x800
PICO_INCLUDE_RTC_DATETIME Whether to include the datetime_t type used with the RP2040 RTC hardware	1 on RP2040
PICO_MALLOC_PANIC Enable/disable panic when an allocation failure occurs	1
PICO_MAX_SHARED_IRQ_HANDLERS Maximum number of shared IRQ handlers	4
PICO_MBEDTLS_SHA256_ALT_USE_DMA Whether to use DMA for writing to hardware for the mbedtls SHA-256 hardware acceleration	1
PICO_NO_BINARY_INFO Don't include "binary info" in the output binary	0 except for `PICO_PLATFORM host`
PICO_NO_FPGA_CHECK Remove the FPGA platform check for small code size reduction	1
PICO_NO_RAM_VECTOR_TABLE Enable/disable the RAM vector table	0
PICO_NO_SIM_CHECK RP2350 Remove the SIM platform check for small code size reduction	1

Name / Description	Default
PICO_OPAQUE_ABSOLUTE_TIME_T Enable opaque type for absolute_time_t to help catch inadvertent confusing uint64_t delays with absolute times	0
PICO_PANIC_FUNCTION Name of a function to use in place of the stock panic function or empty string to simply breakpoint on panic	
PICO_PHEAP_MAX_ENTRIES Maximum number of entries in the pheap	255
PICO_PIO_CLKDIV_ROUND_NEAREST True if floating point PIO clock divisors should be rounded to the nearest possible clock divisor rather than rounding down	PICO_CLKDIV_ROUND_NEAREST
PICO_PIO_USE_GPIO_BASE Enable code for handling more than 32 PIO pins	true when supported and when the device has more than 32 pins
PICO_PIO_VERSION PIO hardware version	0 on RP2040 and 1 on RP2350
PICO_PRINTF_ALWAYS_INCLUDED Whether to always include printf code even if only called weakly (by panic)	1 in debug build 0 otherwise
PICO_PRINTF_DEFAULT_FLOAT_PRECISION Define default floating point precision	6
PICO_PRINTF_FTOA_BUFFER_SIZE Define printf ftoa buffer size	32
PICO_PRINTF_MAX_FLOAT Define the largest float suitable to print with %f	1e9
PICO_PRINTF_NTOA_BUFFER_SIZE Define printf ntoa buffer size	32
PICO_PRINTF_SUPPORT_EXPONENTIAL Enable exponential floating point printing	1
PICO_PRINTF_SUPPORT_FLOAT Enable floating point printing	1
PICO_PRINTF_SUPPORT_LONG_LONG Enable support for long long types (%llu or %p)	1
PICO_PRINTF_SUPPORT_PTRDIFF_T Enable support for the ptrdiff_t type (%t)	1
PICO_PWM_CLKDIV_ROUND_NEAREST True if floating point PWM clock divisors should be rounded to the nearest possible clock divisor rather than rounding down	PICO_CLKDIV_ROUND_NEAREST
PICO_QUEUE_MAX_LEVEL Maintain a field for the highest level that has been reached by a queue	0
PICO_RAND_BUS_PERF_COUNTER_EVENT Bus performance counter event to use for sourcing entropy	arbiter_sram5_perf_event_access
PICO_RAND_BUS_PERF_COUNTER_INDEX Bus performance counter index to use for sourcing entropy	Undefined meaning pick one that is not counting any valid event already
PICO_RAND_ENTROPY_SRC_BUS_PERF_COUNTER Enable/disable use of a bus performance counter as an entropy source	1 if no hardware TRNG

Name / Description	Default
PICO_RAND_ENTROPY_SRC_ROSC Enable/disable use of ROSC as an entropy source	1 if no hardware TRNG
PICO_RAND_ENTROPY_SRC_TIME Enable/disable use of hardware timestamp as an entropy source	1
PICO_RAND_ENTROPY_SRC_TRNG Enable/disable use of hardware TRNG as an entropy source	1 if hardware TRNG is available
PICO_RAND_MIN_ROSC_BIT_SAMPLE_TIME_US Define a default minimum time between sampling the ROSC random bit	10
PICO_RAND_RAM_HASH_END End of address in RAM (non-inclusive) to hash during pico_rand seed initialization	SRAM_END
PICO_RAND_RAM_HASH_START Start of address in RAM (inclusive) to hash during pico_rand seed initialization	PICO_RAND_RAM_HASH_END - 1024
PICO_RAND_ROSC_BIT_SAMPLE_COUNT Number of samples to take of the ROSC random bit per random number generation	1
PICO_RAND_SEED_ENTROPY_SRC_BOARD_ID Enable/disable use of board id as part of the random seed	not PICO_RAND_SEED_ENTROPY_SRC_BOOT_RANDOM
PICO_RAND_SEED_ENTROPY_SRC_BOOT_RANDOM Enable/disable use of the per boot random number as an entropy source for the random seed	0 on RP2040 which has none
PICO_RAND_SEED_ENTROPY_SRC_BUS_PERF_COUNTER Enable/disable use of a bus performance counter as an entropy source for the random seed	PICO_RAND_ENTROPY_SRC_BUS_PERF_COUNTER
PICO_RAND_SEED_ENTROPY_SRC_RAM_HASH Enable/disable use of a RAM hash as an entropy source for the random seed	1 if no hardware TRNG
PICO_RAND_SEED_ENTROPY_SRC_ROSC Enable/disable use of ROSC as an entropy source for the random seed	PICO_RAND_ENTROPY_SRC_ROSC
PICO_RAND_SEED_ENTROPY_SRC_TIME Enable/disable use of hardware timestamp as an entropy source for the random seed	PICO_RAND_ENTROPY_SRC_TIME
PICO_RAND_SEED_ENTROPY_SRC_TRNG Enable/disable use of hardware TRNG as an entropy source for the random seed	PICO_RAND_ENTROPY_SRC_TRNG
PICO_RP2040_B0_SUPPORTED RP2040 Whether to include any specific software support for RP2040 B0 revision	1
PICO_RP2040_B1_SUPPORTED RP2040 Whether to include any specific software support for RP2040 B1 revision	1
PICO_RP2040_B2_SUPPORTED RP2040 Whether to include any specific software support for RP2040 B2 revision	1
PICO_RP2350A RP2350 Whether the current board has an RP2350 in an A (30 GPIO) package	Usually provided via board header
PICO_RP2350_A2_SUPPORTED RP2350 Whether to include any specific software support for RP2350 A2 revision	1

Name / Description	Default
PICO_RUNTIME_NO_INIT_BOOTROM_RESET Do not include SDK implementation of `runtime_init_bootrom_reset` function	1 on RP2040
PICO_RUNTIME_NO_INIT_CLOCKS Do not include SDK implementation of `runtime_init_clocks` function	0
PICO_RUNTIME_NO_INIT_DEFAULT_ALARM_POOL Do not include SDK implementation of `runtime_init_default_alarm_pool` function	1 if `PICO_TIME_DEFAULT_ALARM_POOL_DISABLED` is
PICO_RUNTIME_NO_INIT_EARLY_RESETS Do not include SDK implementation of `runtime_init_early_resets` function	1 on RP2040
PICO_RUNTIME_NO_INIT_INSTALL_RAM_VECTOR_TABLE Do not include SDK implementation of `runtime_init_install_ram_vector_table` function	0 unless RISC-V or RAM binary
PICO_RUNTIME_NO_INIT_MUTEX Do not include SDK implementation of `runtime_init_mutex` function	0
PICO_RUNTIME_NO_INIT_PER_CORE_BOOTROM_RESET Do not include SDK implementation of `runtime_init_per_core_bootrom_reset` function	1 on RP2040
PICO_RUNTIME_NO_INIT_PER_CORE_ENABLE_COPROCESSORS Do not include SDK implementation of `runtime_init_per_core_enable_coprocessors` function	1 on RP2040 or RISC-V
PICO_RUNTIME_NO_INIT_PER_CORE_INSTALL_STACK_GUARD Do not include SDK implementation of `runtime_init_per_core_install_stack_guard` function	1 unless `PICO_USE_STACK_GUARDS` is 1
PICO_RUNTIME_NO_INIT_RP2040_GPIO_IE_DISABLE RP2040 Do not include SDK implementation of `runtime_init_rp2040_gpio_ie_disable` function	0 on RP2040
PICO_RUNTIME_NO_INIT_SPIN_LOCKS_RESET Do not include SDK implementation of `runtime_init_spin_locks_reset` function	0
PICO_RUNTIME_NO_INIT_USB_POWER_DOWN Do not include SDK implementation of `runtime_init_usb_power_down` function	0
PICO_RUNTIME_SKIP_INIT_BOOTROM_LOCKING_ENABLE Skip calling of `runtime_init_bootrom_locking_enable` function during runtime init	0
PICO_RUNTIME_SKIP_INIT_BOOTROM_RESET Skip calling of `runtime_init_bootrom_reset` function during runtime init	1 on RP2040
PICO_RUNTIME_SKIP_INIT_BOOT_LOCKS_RESET Skip calling of `runtime_init_boot_locks_reset` function during runtime init	0
PICO_RUNTIME_SKIP_INIT_CLOCKS Skip calling of `runtime_init_clocks` function during runtime init	0
PICO_RUNTIME_SKIP_INIT_DEFAULT_ALARM_POOL Skip calling of `runtime_init_default_alarm_pool` function during runtime init	1 if `PICO_TIME_DEFAULT_ALARM_POOL_DISABLED` is 1
PICO_RUNTIME_SKIP_INIT_EARLY_RESETS Skip calling of `runtime_init_early_resets` function during runtime init	1 on RP2040

Name / Description	Default
PICO_RUNTIME_SKIP_INIT_INSTALL_RAM_VECTOR_TABLE Skip calling of `runtime_init_install_ram_vector_table` function during runtime init	0 unless RISC-V or RAM binary
PICO_RUNTIME_SKIP_INIT_MUTEX Skip calling of `runtime_init_mutex` function during runtime init	0
PICO_RUNTIME_SKIP_INIT_PER_CORE_BOOTROM_RESET Skip calling of `runtime_init_per_core_bootrom_reset` function during per-core init	1 on RP2040
PICO_RUNTIME_SKIP_INIT_PER_CORE_ENABLE_COPROCESSORS Skip calling of `runtime_init_per_core_enable_coprocessors` function during per-core init	1 on RP2040 or RISC-V
PICO_RUNTIME_SKIP_INIT_PER_CORE_H3_IRQ_REGISTERS Skip calling of `runtime_init_per_core_h3_irq_registers` function during per-core init	1 on non RISC-V
PICO_RUNTIME_SKIP_INIT_PER_CORE_INSTALL_STACK_GUARD Skip calling of `runtime_init_per_core_install_stack_guard` function during runtime init	1 unless `PICO_USE_STACK_GUARDS` is 1
PICO_RUNTIME_SKIP_INIT_RP2040_GPIO_IE_DISABLE RP2040 Skip calling of `runtime_init_rp2040_gpio_ie_disable` function during runtime init	0 on RP2040
PICO_RUNTIME_SKIP_INIT_SPIN_LOCKS_RESET Skip calling of `runtime_init_spin_locks_reset` function during runtime init	0
PICO_RUNTIME_SKIP_INIT_USB_POWER_DOWN Skip calling of `runtime_init_usb_power_down` function during runtime init	0
PICO_SHARED_IRQ_HANDLER_DEFAULT_ORDER_PRIORITY Set default shared IRQ order priority	0x80
PICO_SPINLOCK_ID_ATOMIC Spinlock ID for atomics	13
PICO_SPINLOCK_ID_CLAIM_FREE_FIRST Lowest Spinlock ID in the 'claim free' range	24
PICO_SPINLOCK_ID_CLAIM_FREE_LAST Highest Spinlock ID in the 'claim free' range	31
PICO_SPINLOCK_ID_HARDWARE_CLAIM Spinlock ID for Hardware claim protection	11
PICO_SPINLOCK_ID_IRQ Spinlock ID for IRQ protection	9
PICO_SPINLOCK_ID_OS1 First Spinlock ID reserved for use by low level OS style software	14
PICO_SPINLOCK_ID_OS2 Second Spinlock ID reserved for use by low level OS style software	15
PICO_SPINLOCK_ID_RAND Spinlock ID for Random Number Generator	12
PICO_SPINLOCK_ID_STRIPED_FIRST Lowest Spinlock ID in the 'striped' range	16
PICO_SPINLOCK_ID_STRIPED_LAST Highest Spinlock ID in the 'striped' range	23

Name / Description	Default
PICO_SPINLOCK_ID_TIMER Spinlock ID for Timer protection	10
PICO_STACK_SIZE Minimum amount of stack space reserved in the linker script for each core. See also PICO_CORE1_STACK_SIZE	0x800
PICO_STDIO_DEADLOCK_TIMEOUT_MS Time after which to assume stdio_usb is deadlocked by use in IRQ and give up	1000
PICO_STDIO_DEFAULT_CRLF Default for CR/LF conversion enabled on all stdio outputs	1
PICO_STDIO_ENABLE_CRLF_SUPPORT Enable/disable CR/LF output conversion support	1
PICO_STDIO_RTT_DEFAULT_CRLF Default state of CR/LF translation for rtt output	PICO_STDIO_DEFAULT_CRLF
PICO_STDIO_SEMIHOSTING_DEFAULT_CRLF Default state of CR/LF translation for semihosting output	PICO_STDIO_DEFAULT_CRLF
PICO_STDIO_SHORT_CIRCUIT_CLIB_FUNCS Directly replace common stdio functions such as putchar from the C-library to avoid pulling in lots of c library code for simple output	1
PICO_STDIO_STACK_BUFFER_SIZE Define printf buffer size (on stack)… this is just a working buffer not a max output size	128
PICO_STDIO_UART_DEFAULT_CRLF Default state of CR/LF translation for UART output	PICO_STDIO_DEFAULT_CRLF
PICO_STDIO_UART_SUPPORT_CHARS_AVAILABLE_CALLBACK Enable UART STDIO support for stdio_set_chars_available_callback. Can be disabled to make use of the uart elsewhere	1
PICO_STDIO_USB_CONNECTION_WITHOUT_DTR Disable use of DTR for connection checking meaning connection is assumed to be valid	0
PICO_STDIO_USB_CONNECT_WAIT_TIMEOUT_MS Maximum number of milliseconds to wait during initialization for a CDC connection from the host (negative means indefinite) during initialization	0
PICO_STDIO_USB_DEFAULT_CRLF Default state of CR/LF translation for USB output	PICO_STDIO_DEFAULT_CRLF
PICO_STDIO_USB_DEINIT_DELAY_MS Number of milliseconds to wait before deinitializing stdio_usb	110
PICO_STDIO_USB_DEVICE_SELF_POWERED Set USB device as self powered device	0
PICO_STDIO_USB_ENABLE_RESET_VIA_BAUD_RATE Enable/disable resetting into BOOTSEL mode if the host sets the baud rate to a magic value (PICO_STDIO_USB_RESET_MAGIC_BAUD_RATE)	1 if application is not using TinyUSB directly
PICO_STDIO_USB_ENABLE_RESET_VIA_VENDOR_INTERFACE Enable/disable resetting into BOOTSEL mode via an additional VENDOR USB interface - enables picotool based reset	1 if application is not using TinyUSB directly

Name / Description	Default
PICO_STDIO_USB_LOW_PRIORITY_IRQ Explicit User IRQ number to claim for tud_task() background execution instead of letting the implementation pick a free one dynamically (deprecated)	
PICO_STDIO_USB_POST_CONNECT_WAIT_DELAY_MS Number of extra milliseconds to wait when using PICO_STDIO_USB_CONNECT_WAIT_TIMEOUT_MS after a host CDC connection is detected (some host terminals seem to sometimes lose transmissions sent right after connection)	50
PICO_STDIO_USB_RESET_BOOTSEL_ACTIVITY_LED Optionally define a pin to use as bootloader activity LED when BOOTSEL mode is entered via USB (either VIA_BAUD_RATE or VIA_VENDOR_INTERFACE)	
PICO_STDIO_USB_RESET_BOOTSEL_ACTIVITY_LED_ACTIVE_LOW Whether pin to use as bootloader activity LED when BOOTSEL mode is entered via USB (either VIA_BAUD_RATE or VIA_VENDOR_INTERFACE) is active low	0
PICO_STDIO_USB_RESET_BOOTSEL_FIXED_ACTIVITY_LED Whether the pin specified by PICO_STDIO_USB_RESET_BOOTSEL_ACTIVITY_LED is fixed or can be modified by picotool over the VENDOR USB interface	0
PICO_STDIO_USB_RESET_BOOTSEL_INTERFACE_DISABLE_MASK Optionally disable either the mass storage interface (bit 0) or the PICOBOOT interface (bit 1) when entering BOOTSEL mode via USB (either VIA_BAUD_RATE or VIA_VENDOR_INTERFACE)	0
PICO_STDIO_USB_RESET_INTERFACE_SUPPORT_MS_OS_20_DESCRIPTOR If vendor reset interface is included add support for Microsoft OS 2.0 Descriptor	1
PICO_STDIO_USB_RESET_INTERFACE_SUPPORT_RESET_TO_BOOTSEL If vendor reset interface is included allow rebooting to BOOTSEL mode	1
PICO_STDIO_USB_RESET_INTERFACE_SUPPORT_RESET_TO_FLASH_BOOT If vendor reset interface is included allow rebooting with regular flash boot	1
PICO_STDIO_USB_RESET_MAGIC_BAUD_RATE Baud rate that if selected causes a reset into BOOTSEL mode (if PICO_STDIO_USB_ENABLE_RESET_VIA_BAUD_RATE is set)	1200
PICO_STDIO_USB_RESET_RESET_TO_FLASH_DELAY_MS Delay in ms before rebooting via regular flash boot	100
PICO_STDIO_USB_STDOUT_TIMEOUT_US Number of microseconds to be blocked trying to write USB output before assuming the host has disappeared and discarding data	500000
PICO_STDIO_USB_SUPPORT_CHARS_AVAILABLE_CALLBACK Enable USB STDIO support for stdio_set_chars_available_callback. Can be disabled to make use of USB CDC RX callback elsewhere	1
PICO_STDIO_USB_TASK_INTERVAL_US Period of microseconds between calling tud_task in the background	1000
PICO_STDOUT_MUTEX Enable/disable mutex around stdout	1
PICO_TIME_DEFAULT_ALARM_POOL_DISABLED Disable the default alarm pool	0

Name / Description	Default
PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM Select which HW alarm is used for the default alarm pool	3
PICO_TIME_DEFAULT_ALARM_POOL_MAX_TIMERS Selects the maximum number of concurrent timers in the default alarm pool	16
PICO_TIME_SLEEP_OVERHEAD_ADJUST_US How many microseconds to wake up early (and then busy_wait) to account for timer overhead when sleeping in low power mode	6
PICO_UART_DEFAULT_CRLF Enable/disable CR/LF translation on UART	0
PICO_UART_ENABLE_CRLF_SUPPORT Enable/disable CR/LF translation support	1
PICO_USE_FASTEST_SUPPORTED_CLOCK Use the fastest officially supported clock by default	0
PICO_USE_GPIO_COPROCESSOR Enable/disable use of the GPIO coprocessor for GPIO access	1
PICO_USE_MALLOC_MUTEX Whether to protect malloc etc with a mutex	1 with pico_multicore, 0 otherwise
PICO_USE_STACK_GUARDS RP2350 Enable/disable stack guards	0
PICO_USE_SW_SPIN_LOCKS Use software implementation for spin locks	1 on RP2350 due to errata
PICO_VTABLE_PER_CORE User is using separate vector tables per core	0
PICO_XOSC_STARTUP_DELAY_MULTIPLIER Multiplier (from 1ms) for xosc startup delay to accommodate slow-starting oscillators	6
PLL_SYS_POSTDIV1 System clock PLL post divider 1 setting	6 on RP2040 or 5 on RP2350
PLL_SYS_POSTDIV2 System clock PLL post divider 2 setting	2
PLL_SYS_REFDIV PLL reference divider setting for PLL_SYS	1
PLL_SYS_VCO_FREQ_HZ System clock PLL frequency	(1500 * MHZ)
PLL_USB_POSTDIV1 USB clock PLL post divider 1 setting	5
PLL_USB_POSTDIV2 USB clock PLL post divider 2 setting	5
PLL_USB_REFDIV PLL reference divider setting for PLL_USB	1
PLL_USB_VCO_FREQ_HZ USB clock PLL frequency	(1200 * MHZ)
SYS_CLK_HZ RP2040 System operating frequency in Hz	125000000

Name / Description	Default
SYS_CLK_HZ RP2350 System operating frequency in Hz	150000000
SYS_CLK_VREG_VOLTAGE_AUTO_ADJUST Should the regulator voltage be adjusted above SYS_CLK_VREG_VOLTAGE_MIN when initializing the clocks	0
SYS_CLK_VREG_VOLTAGE_AUTO_ADJUST_DELAY_US Number of microseconds to wait after updating regulator voltage due to SYS_CLK_VREG_VOLTAGE_MIN to allow voltage to settle	1000
SYS_CLK_VREG_VOLTAGE_MIN minimum voltage (see VREG_VOLTAGE_x_xx) for the voltage regulator to be ensured during clock initialization if SYS_CLK_VREG_VOLTAGE_AUTO_ADJUST is 1	
USB_CLK_HZ USB clock frequency. Must be 48MHz for the USB interface to operate correctly	48000000
USB_DPRAM_MAX Set amount of USB RAM used by USB system	4096
XOSC_HZ Crystal oscillator frequency in Hz	12000000

Chapter 6. CMake build configuration

Use CMake cache variables to customize SDK builds.

6.1. Full List of SDK Configuration Variables

Table 35. SDK CMake Configuration Variables

Name / Description	Default
PICO_BARE_METAL Flag to exclude anything except base headers from the build	0
PICO_BOARD Board name being built for. This may be specified in the user environment (see Section 6.2)	pico or pico2
PICO_BOARD_CMAKE_DIRS List of directories to look for <PICO_BOARD>.cmake in. This may be specified in the user environment	
PICO_BOARD_HEADER_DIRS List of directories to look for <PICO_BOARD>.h in. This may be specified the user environment	
PICO_CMAKE_PRELOAD_PLATFORM_FILE Custom CMake file to use to set up the platform environment	
PICO_COMPILER Specifies the compiler family to use (see Section 6.3)	PICO_DEFAULT_COMPILER which is set based on PICO_PLATFORM
PICO_CONFIG_HEADER_FILES List of extra header files to include from pico/config.h for all platforms	
PICO_COPY_TO_RAM Option to default all binaries to copy code from flash to SRAM before running (see Section 6.4)	0
PICO_CXX_ENABLE_CXA_ATEXIT Enable cxa-atexit	0
PICO_CXX_ENABLE_EXCEPTIONS Enable CXX exception handling	0
PICO_CXX_ENABLE_RTTI Enable CXX rtti	0
PICO_DEFAULT_BOOT_STAGE2 Simpler alternative to specifying PICO_DEFAULT_BOOT_STAGE2_FILE where the latter is set to src/rp2_common/boot_stage2/{PICO_DEFAULT_BOOT_STAGE2}.S	compile_time_choice
PICO_DEFAULT_BOOT_STAGE2_FILE Default boot stage 2 file to use unless overridden by pico_set_boot_stage2 on the TARGET; this setting is useful when explicitly setting the default build from a per board CMake file	

Name / Description	Default
PICO_DEFAULT_PIOASM_OUTPUT_FORMAT Default output format used by pioasm when using pico_generate_pio_header	c-sdk
PICO_DEFAULT_RP2350_PLATFORM RP2350 Default actual platform to build for if rp2350 is specified for PICO_PLATFORM e.g. rp2350-arm-s/rp2350-riscv	rp2350-arm-s
PICO_DEFAULT_UART_BAUD_RATE Define the default UART baudrate	115200
PICO_GCC_TRIPLE List of GCC_TRIPLES — usually only one — to try when searching for a compiler. This may be specified the user environment (see Section 6.3)	PICO_DEFAULT_GCC_TRIPLE which is set based on PICO_COMPILER
PICO_HOST_CONFIG_HEADER_FILES List of extra header files to include from pico/config.h for the host platform only	
PICO_NO_COPRO_DIS Disable disassembly listing postprocessing that disassembles RP2350 coprocessor instructions	0
PICO_NO_FLASH Option to default all binaries to not use flash i.e. run from SRAM (see Section 6.4)	0
PICO_NO_GC_SECTIONS Disable `-ffunction-sections` `-fdata-sections` and `--gc-sections`	0
PICO_NO_HARDWARE Option as to whether the build is not targeting an RP2040 or RP2350 device	1 when PICO_PLATFORM is host, 0 otherwise
PICO_NO_PICOTOOL Disable use/requirement for picotool meaning that UF2 output and signing/hashing and coprocoessor disassembly will all be unavailable	0
PICO_NO_TARGET_NAME Don't define PICO_TARGET_NAME	0
PICO_NO_UF2 Disable UF2 output	0
PICO_ON_DEVICE Option as to whether the build is targeting an RP2040 or RP2350 device	0 when PICO_PLATFORM is host, 1 otherwise
PICO_PLATFORM Platform to build for e.g. rp2040/rp2350/rp2350-arm-s/rp2350-riscv/host. This may be specified in the user environment (see Section 6.2)	based on PICO_BOARD or environment value
PICO_RP2040_CONFIG_HEADER_FILES RP2040 List of extra header files to include from pico/config.h for the rp2040 platform only	
PICO_RP2350_ARM_S_CONFIG_HEADER_FILES RP2350 List of extra header files to include from pico/config.h for the rp2350-arm-s platform only	
PICO_RP2350_RISCV_CONFIG_HEADER_FILES RP2350 List of extra header files to include from pico/config.h for the riscv platform only	

Name / Description	Default
PICO_SDK_VERSION_MAJOR SDK major version number	Current SDK major version
PICO_SDK_VERSION_MINOR SDK minor version number	Current SDK minor version
PICO_SDK_VERSION_PRE_RELEASE_ID Optional SDK pre-release version identifier	Current SDK pre-release identifier
PICO_SDK_VERSION_REVISION SDK version revision	Current SDK revision
PICO_SDK_VERSION_STRING SDK version string	Current SDK version string
PICO_STDIO_RTT Option to globally enable stdio RTT for all targets by default	0
PICO_STDIO_SEMIHOSTING Option to globally enable stdio semi-hosting for all targets by default	0
PICO_STDIO_UART Option to globally enable stdio UART for all targets by default	1
PICO_STDIO_USB Option to globally enable stdio USB for all targets by default	0
PICO_STDIO_USB_CONNECT_WAIT_TIMEOUT_MS Maximum number of milliseconds to wait during initialization for a CDC connection from the host (negative means indefinite) during initialization	0
PICO_TOOLCHAIN_PATH Path to search for compiler (see Section 6.3)	none (i.e. search system paths)
PICO_USE_DEFAULT_MAX_PAGE_SIZE Don't shrink linker max page to 4096	0

6.2. Platform and Board Configuration

Passing `PICO_BOARD=my_board_name` to the CMake build (or specifying it in your environment) will cause the header `my_board_name.h` to be included by all other SDK headers in order to provide `#defines` particular to the board you are using.

You may also 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`.

On previous versions of the SDK there was generally little need for setting `PICO_PLATFORM` as the default value `rp2040` selected RP2040 - the one and only RP-series microcontroller at the time.

SDK version 2.0.0 now supports the following RP-series microcontroller platforms along with the pre-existing value `host` that can be used to build code for testing.

`rp2040`
: Building for RP2040

`rp2350-arm-s`
: Building for RP2350 on Arm processors; the "s" stands for secure, and means the binary runs directly from the bootrom, when the processor is still in secure mode.

`rp2350-riscv`
: Building for RP2350 on RISC-V processors.

Individual manufactured boards will usually support only one of either RP2040 or RP2350. To avoid having to specify PICO_PLATFORM in addition to PICO_BOARD, specifying the latter can now automatically set the former (or vice versa).

The following steps are applied in order, with the results from the previous step being used in the next:

1. If neither PICO_BOARD or PICO_PLATFORM is specified, PICO_PLATFORM defaults to PICO_DEFAULT_PLATFORM which itself defaults to rp2040
2. If PICO_PLATFORM is specified and not PICO_BOARD, then PICO_BOARD is defaulted based on the value of PICO_PLATFORM:
   - pico for PICO_PLATFORM=rp2040
   - pico2 for PICO_PLATFORM=rp2350-arm-s or PICO_PLATFORM=rp2350-riscv
3. If PICO_BOARD is specified but not PICO_PLATFORM, PICO_PLATFORM will be set if a value for it is specified in the board header.

Because most RP2350 boards allow both Arm and RISC-V development, rp2350 is also a valid value for PICO_PLATFORM, and is often specified by a board header in step 3 above, but is always replaced with the value of PICO_DEFAULT_RP2350_PLATFORM to allow the user their own preference. PICO_DEFAULT_RP2350_PLATFORM defaults to rp2350-arm-s if not otherwise specified.

NOTE

Both PICO_PLATFORM and PICO_BOARD are **latched** if they have been specified via the environment, on the first CMake configuration; i.e. the value from the environment will not be used when configuring CMake subsequently in the same existing build directory.

6.3. Compiler and Toolchain Configuration

The SDK supports building for Arm Cortex-M0 plus processors on RP2040 and for both Arm Cortex-M33 processors and RISC-V Hazard3 processors on RP2350.

The SDK also supports building with either GCC or LLVM (clang) on Arm. See Section 2.10 for more details of supported compilers.

6.3.1. Variables

The following variables are used to find and configure the right compiler.

6.3.1.1. PICO_COMPILER

This is usually defaulted for you to a GCC compiler based on PICO_PLATFORM. However, you can select one of the following values

- pico_arm_gcc - Selects pico_arm_cortex_m0plus_gcc on RP2040 and pico_arm_cortex_m33_gcc on RP2350
- pico_arm_cortex_m0plus_gcc - Configures GCC to build for Arm Cortex-M0 plus
- pico_arm_cortex_m33_gcc - Configures GCC to build for Arm Cortex-M33
- pico_arm_clang - Selects pico_arm_cortex_m0plus_clang on RP2040 and pico_arm_cortex_m33_clang on RP2350
- pico_arm_cortex_m0plus_clang - Configures LLVM/clang to build for Arm Cortex-M0 plus
- pico_arm_cortex_m33_clang - Configures LLVM/clang to build for Arm Cortex-M33
- pico_riscv_gcc - Configures GCC to build for RISC-V Hazard3
- pico_riscv_gcc_zcb_zcmp - Configures GCC to build for RISC-V Hazard3 using zcb and zcmp extensions that aren't supported by all compilers

6.3.1.2. PICO_GCC_TRIPLE

This specifies one or more compiler "triples" to try when looking for a GCC compiler.

On Arm this defaults to `arm-none-eabi`.
On RISC-V this defaults to `riscv32-unknown-elf;riscv32-corev-elf` i.e. the two most common options are supported.

6.3.1.3. PICO_TOOLCHAIN_PATH

Armed with `PICO_COMPILER` and `PICO_GCC_TRIPLE` (if using GCC) the SDK will then search for a compiler. By default, it searches the path, but `PICO_TOOLCHAIN_PATH` may be set to specify the root directory of a compiler toolchain install.

6.3.1.4. PICO_CLIB

Most programs for the SDK require a C-library. Generally your installed compiler will include the toolchain. In this case, the SDK will try to detect either of the following runtimes, as which one is used effects how the SDK interacts with it.

- `newlib`
- `picolibc`
- `llvm-libc`

The SDK sets `PICO_CLIB` to one of these values, however you can set it yourself first if you want to force a choice.

6.4. Binary Type configuration

These variables control how executables for RP-series microcontroller are laid 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 copy into RAM at startup.

Variable name	Values	Result
PICO_DEFAULT_BINARY_TYPE	`default`	Stores binaries in flash storage. Runs binaries from flash storage.
	`no_flash`	Stores binaries in memory. Runs binaries from memory. Does not require any flash storage. Note: You must reload this type of binary after every reboot via UF2 file or debugger.
	`copy_to_ram`	Stores binaries in flash, but copies them to memory (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

You can set the binary type for each executable target (as created by `add_executable`) by calling `pico_set_binary_type(target type)` using the same type as `PICO_DEFAULT_BINARY_TYPE`.

Chapter 7. CMake build functions

7.1. Control of picotool post-processing (not available on RP2040)

These functions control what post-processing is performed by picotool on the executables produced by the sdk. They can be used to hash/sign binaries, package no_flash binaries, and embed partition tables into block loops. These functions all set the specified target properties, and therefore if the property should be set for multiple targets then it can be set manually with `set_property` or other CMake functions to set properties for a given scope.

pico_package_uf2_output(TARGET PACKADDR)	Package a UF2 output to be written to the PACKADDR address. This can be used with a no_flash binary to write the UF2 to flash when dragging & dropping, and it will be copied to SRAM by the bootrom before execution. This sets PICOTOOL_UF2_PACKAGE_ADDR to PACKADDR.
pico_set_otp_key_output_file(TARGET OTPFILE)	Output the public key hash and other necessary rows to an otp JSON file. This sets PICOTOOL_OTP_FILE to OTPFILE.
pico_load_map_clear_sram(TARGET)	Adds an entry to the load map to instruct the bootrom to clear all of SRAM before loading the binary. This appends the `--clear` argument to PICOTOOL_EXTRA_PROCESS_ARGS.
pico_set_binary_version(<TARGET> [MAJOR <version>] [MINOR <version>] [ROLLBACK <version>] [ROLLBACK_ROWS <rows…>])	Adds a version item to the metadata block, with the given major, minor and rollback version, along with the rollback rows. These are appended as arguments to PICOTOOL_EXTRA_PROCESS_ARGS if setting the rollback version, or set as compile definitions if only setting the major/minor versions.
pico_set_uf2_family(TARGET FAMILY)	Set the UF2 family to use when creating the UF2. This sets PICOTOOL_UF2_FAMILY to FAMILY.
pico_sign_binary(TARGET [SIGFILE])	Sign the target binary with the given PEM signature. This sets PICOTOOL_SIGN_OUTPUT to true, PICOTOOL_SIGFILE to SIGFILE (if specified), and PICOTOOL_OTP_FILE to ${TARGET}.otp.json (if not already set). To specify a common SIGFILE for multiple targets, the SIGFILE property can be set for a given scope, and then the SIGFILE argument is optional.
pico_hash_binary(TARGET)	Hash the target binary. This sets PICOTOOL_HASH_OUTPUT to true.
pico_embed_pt_in_binary(TARGET PTFILE)	Create the specified partition table from JSON, and embed it in the block loop. This sets PICOTOOL_EMBED_PT to PTFILE.
pico_encrypt_binary(TARGET AESFILE [SIGFILE])	Encrypt the target binary with the given AES key (should be a binary file containing 32 bytes of a random key), and sign the encrypted binary. This sets PICOTOOL_AESFILE to AESFILE, and PICOTOOL_ENC_SIGFILE to SIGFILE if present, else PICOTOOL_SIGFILE.

Chapter 8. Board configuration

Board configuration is the process of customising the SDK to run on a specific board design. The SDK comes with some predefined configurations for boards produced by Raspberry Pi and other manufacturers, 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 through where these configurations files are, how to make changes and set parameters, and how to build your SDK using CMake with your customisations.

8.1. The Configuration files

Board specific 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.

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/boards/include/boards/pico.h

```
/*
 * 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"

// pico_cmake_set PICO_PLATFORM=rp2040

#ifndef _BOARDS_PICO_H
#define _BOARDS_PICO_H

// For board detection
#define RASPBERRYPI_PICO

// --- UART ---
#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

// --- LED ---
#ifndef PICO_DEFAULT_LED_PIN
```

```
#define PICO_DEFAULT_LED_PIN 25
#endif
// no PICO_DEFAULT_WS2812_PIN

// --- I2C ---
#ifndef PICO_DEFAULT_I2C
#define PICO_DEFAULT_I2C 0
#endif
#ifndef PICO_DEFAULT_I2C_SDA_PIN
#define PICO_DEFAULT_I2C_SDA_PIN 4
#endif
#ifndef PICO_DEFAULT_I2C_SCL_PIN
#define PICO_DEFAULT_I2C_SCL_PIN 5
#endif

// --- SPI ---
#ifndef PICO_DEFAULT_SPI
#define PICO_DEFAULT_SPI 0
#endif
#ifndef PICO_DEFAULT_SPI_SCK_PIN
#define PICO_DEFAULT_SPI_SCK_PIN 18
#endif
#ifndef PICO_DEFAULT_SPI_TX_PIN
#define PICO_DEFAULT_SPI_TX_PIN 19
#endif
#ifndef PICO_DEFAULT_SPI_RX_PIN
#define PICO_DEFAULT_SPI_RX_PIN 16
#endif
#ifndef PICO_DEFAULT_SPI_CSN_PIN
#define PICO_DEFAULT_SPI_CSN_PIN 17
#endif

// --- FLASH ---

#define PICO_BOOT_STAGE2_CHOOSE_W25Q080 1

#ifndef PICO_FLASH_SPI_CLKDIV
#define PICO_FLASH_SPI_CLKDIV 2
#endif

// pico_cmake_set_default PICO_FLASH_SIZE_BYTES = (2 * 1024 * 1024)
#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_RP2040_B0_SUPPORTED
#define PICO_RP2040_B0_SUPPORTED 1
#endif

// The GPIO Pin used to read VBUS to determine if the device is battery powered.
#ifndef PICO_VBUS_PIN
#define PICO_VBUS_PIN 24
#endif

// The GPIO Pin used to monitor VSYS. Typically you would use this with ADC.
// There is an example in adc/read_vsys in pico-examples.
#ifndef PICO_VSYS_PIN
#define PICO_VSYS_PIN 29
#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 on-board LED, and the flash size.

To create your own configuration file, create a file in the board `../source/folder` with the name of your board, for example, `myboard.h`. Enter your board specific parameters in this file.

8.2. 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.

8.3. Available configuration parameters

Table 34 lists all the available configuration parameters available within the SDK. You can set any configuration variable in a board configuration header file, however the convention is to limit that to configuration items directly affected by the board design (e.g. pins, clock frequencies etc.) Other configuration items should generally be overridden in the CMake configuration (or another configuration header) for the application being built.

Chapter 9. Embedded Binary Information

Binary information is machine-readable information embedded in a binary by the SDK (or other development tools) such that it can be read by `picotool` or other tooling.

9.1. Basic information

This information is really handy when you pick up a Pico-series device and don't know what is on it!

Basic information includes

- program name
- program description
- program version string
- program build date
- program url
- program end address
- program features, this is a list built from individual strings in the binary, that can be displayed (e.g. we will have one for UART stdio and one for USB stdio) in the SDK
- build attributes, this is a similar list of strings, for things pertaining to the binary itself (e.g. Debug Build)

9.2. Pins

This is certainly handy when you have an executable called `hello_serial.elf` but you forgot what Raspberry Pi microcontroller-based board it was built for, as different boards may have different pins broken out.

Static (fixed) pin assignments can be recorded in the binary in very compact form:

```
$ picotool info --pins sprite_demo.elf
File sprite_demo.elf:

Fixed Pin Information
0-4:    Red 0-4
6-10:   Green 0-4
11-15:  Blue 0-4
16:     HSync
17:     VSync
18:     Display Enable
19:     Pixel Clock
20:     UART1 TX
21:     UART1 RX
```

9.3. Full Information

Full information is available with the `-a` option:

```
$ picotool info -a i2c_bus_scan.elf
File i2c_bus_scan.elf:

Program Information
 name:          i2c_bus_scan
 web site:      https://github.com/raspberrypi/pico-examples/tree/HEAD/i2c/bus_scan
 features:      UART stdin / stdout
 binary start:  0x10000000
 binary end:    0x10004c74

Fixed Pin Information
 0:  UART0 TX
 1:  UART0 RX
 4:  I2C0 SDA
 5:  I2C0 SCL

Build Information
 sdk version:       2.0.0-develop
 pico_board:        pico
 build date:        Aug  1 2024
 build attributes:  Debug
```

9.4. Including Binary Information

Binary information is declared in the program by macros; for the following example:

```
$ picotool info --pins sprite_demo.elf
File sprite_demo.elf:

Fixed Pin Information
0-4:    Red 0-4
6-10:   Green 0-4
11-15:  Blue 0-4
16:     HSync
17:     VSync
18:     Display Enable
19:     Pixel Clock
20:     UART1 TX
21:     UART1 RX
```

There is one line in the `setup_default_uart` function:

```
bi_decl_if_func_used(bi_2pins_with_func(PICO_DEFAULT_UART_RX_PIN, PICO_DEFAULT_UART_TX_PIN,
GPIO_FUNC_UART));
```

The two pin numbers, and the function UART are stored, then decoded to their actual function names (UART1 TX etc) by picotool. The `bi_decl_if_func_used` makes sure the binary information is only included if the containing function is called.

Equally, the video code contains a few lines like this:

```
bi_decl_if_func_used(bi_pin_mask_with_name(0x1f << (PICO_SCANVIDEO_COLOR_PIN_BASE +
PICO_SCANVIDEO_DPI_PIXEL_RSHIFT), "Red 0-4"));
```

The macros are designed to waste as little space as possible, but you can turn everything off with preprocessor var `PICO_NO_BINARY_INFO=1`. Additionally, any SDK code that inserts binary info can be separately excluded by its own preprocessor var.

To ad your own binary info, you need:

```
#include "pico/binary_info.h"
```

There are a bunch of `bi_` macros in the headers

```
#define bi_binary_end(end)
#define bi_program_name(name)
#define bi_program_description(description)
#define bi_program_version_string(version_string)
#define bi_program_build_date_string(date_string)
#define bi_program_url(url)
#define bi_program_feature(feature)
#define bi_program_build_attribute(attr)
#define bi_1pin_with_func(p0, func)
#define bi_2pins_with_func(p0, p1, func)
#define bi_3pins_with_func(p0, p1, p2, func)
#define bi_4pins_with_func(p0, p1, p2, p3, func)
#define bi_5pins_with_func(p0, p1, p2, p3, p4, func)
#define bi_pin_range_with_func(plo, phi, func)
#define bi_pin_mask_with_name(pmask, label)
#define bi_pin_mask_with_names(pmask, label)
#define bi_1pin_with_name(p0, name)
#define bi_2pins_with_names(p0, name0, p1, name1)
#define bi_3pins_with_names(p0, name0, p1, name1, p2, name2)
#define bi_4pins_with_names(p0, name0, p1, name1, p2, name2, p3, name3)
```

which make use of underlying macros, e.g.

```
#define bi_program_url(url) bi_string(BINARY_INFO_TAG_RASPBERRY_PI, BINARY_INFO_ID_RP_PROGRAM_URL,
url)
```

You then either use `bi_decl(bi_blah(…))` for unconditional inclusion of the binary info blah, or `bi_decl_if_func_used(bi_blah(…))` for binary information that may be stripped if the enclosing function is not included in the binary by the linker (think `--gc-sections`).

For example,

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

const uint LED_PIN = 25;

int main() {
```

```

    bi_decl(bi_program_description("This is a test binary."));
    bi_decl(bi_1pin_with_name(LED_PIN, "On-board LED"));

    setup_default_uart();
    gpio_set_function(LED_PIN, GPIO_FUNC_PROC);
    gpio_set_dir(LED_PIN, GPIO_OUT);
    while (1) {
        gpio_put(LED_PIN, 0);
        sleep_ms(250);
        gpio_put(LED_PIN, 1);
        puts("Hello World\n");
        sleep_ms(1000);
    }
}
```

when queried with `picotool`,

```
$ sudo picotool info -a test.uf2
File test.uf2:

Program Information
 name:          test
 description:   This is a test binary.
 features:      stdout to UART
 binary start:  0x10000000
 binary end:    0x100031f8

Fixed Pin Information
 0:    UART0 TX
 1:    UART0 RX
 25:   On-board LED

Build Information
 build date:   Jan  4 2021
```

shows our information strings in the output.

9.5. Setting Common Information from CMake

You can also set fields directly from your project's CMake file, e.g.,

```
pico_set_program_name(foo "not foo") ①
pico_set_program_description(foo "this is a foo")
pico_set_program_version_string(foo "0.00001a")
pico_set_program_url(foo "www.plinth.com/foo")
```

1. The name "foo" would be the default.

NOTE

All of these are passed as command line arguments to the compilation, so if you plan to use quotes, newlines etc. you may have better luck defining it using `bi_decl` in the code.

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 segment 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 GPIOs need to pull low (to ground) to complete the circuit. The pull direction of the GPIOs 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.

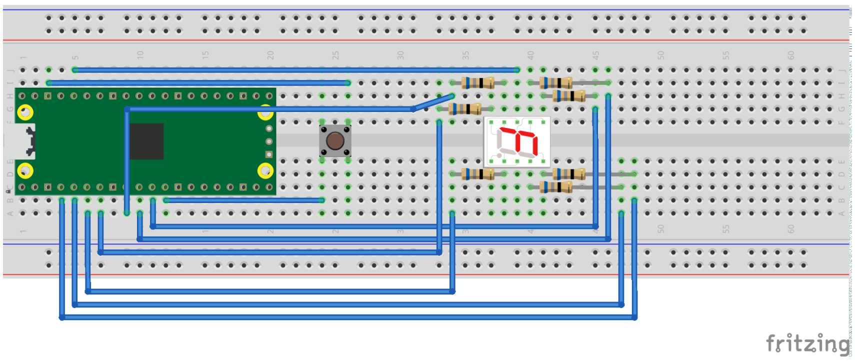


Figure 9. Wiring Diagram for 7 segment LED.

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/blob/master/gpio/hello_7segment/CMakeLists.txt

```
add_executable(hello_7segment
        hello_7segment.c
        )

# pull in common dependencies
target_link_libraries(hello_7segment pico_stdlib)

```

```
# create map/bin/hex file etc.
pico_add_extra_outputs(hello_7segment)

# add url via pico_set_program_url
example_auto_set_url(hello_7segment)
```

hello_7segment.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/gpio/hello_7segment/hello_7segment.c

```
/**
 * 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 segment A, 3 to B etc.
  So, connect GPIO 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 GPIOs
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 GPIOs 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);
    }
}
/// \end::hello_gpio[]
```

Bill of Materials

Table 36. A list of materials required for the example

Item	**Quantity**	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	https://www.raspberrypi.com/products/raspberry-pi-pico/
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 10 for wiring instructions.

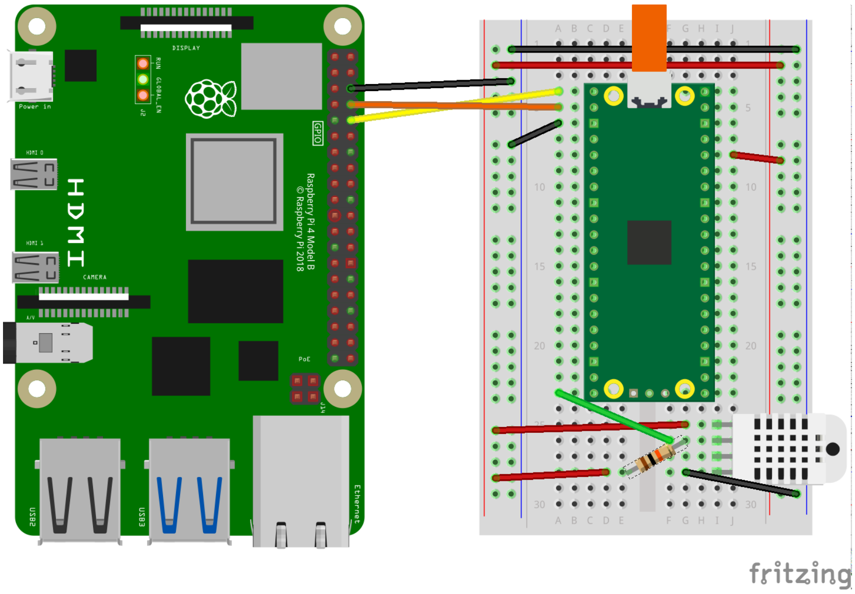


Figure 10. Wiring the DHT-22 temperature sensor to Raspberry Pi Pico, and connecting Pico's UART0 to the Raspberry Pi 4.

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 10 and you should see something similar to Figure 11 in `minicom` when connected to `/dev/serial0` on the Raspberry Pi.

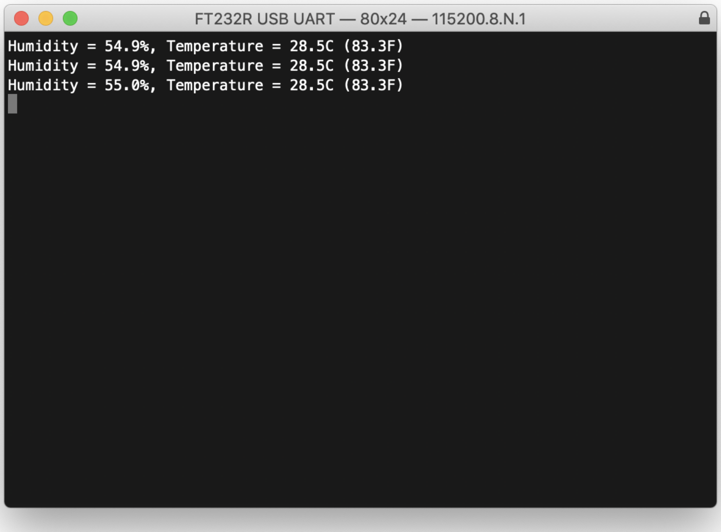


Figure 11. Serial output over Pico's UART0 in a terminal window.

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/blob/master/gpio/dht_sensor/CMakeLists.txt

```
add_executable(dht
        dht.c
        )

target_link_libraries(dht pico_stdlib)

pico_add_extra_outputs(dht)

# add url via pico_set_program_url
example_auto_set_url(dht)
```

dht.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/gpio/dht_sensor/dht.c

```
/**
 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
 *
```

```
 * SPDX-License-Identifier: BSD-3-Clause
 **/

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

#ifdef PICO_DEFAULT_LED_PIN
#define LED_PIN PICO_DEFAULT_LED_PIN
#endif

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(DHT_PIN);
#ifdef LED_PIN
    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);
#endif
    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);

#ifdef LED_PIN
    gpio_put(LED_PIN, 1);
#endif
    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++;
        }
    }
#ifdef LED_PIN
    gpio_put(LED_PIN, 0);
#endif

    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 37. A list of materials required for the example

Item	**Quantity**	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	https://www.raspberrypi.com/products/raspberry-pi-pico/
10 kΩ resistor	1	generic part
M/M Jumper wires	4	generic part
DHT-22 sensor	1	generic part

Attaching a 16x2 LCD via TTL

This example code shows how to interface the Raspberry Pi Pico to one of the very common 16x2 LCD character displays. Due to the large number of pins these displays use, they are commonly used with extra drivers or backpacks. In this example, we will use an Adafruit LCD display backpack, which supports communication over USB or TTL. A monochrome display with an RGB backlight is also used, but the backpack is compatible with monochrome backlight displays too. There is another example that uses I2C to control a 16x2 display.

The backpack processes a set of commands that are documented here and preceded by the "special" byte 0xFE. The backpack does the ASCII character conversion and even supports custom character creation. In this example, we use the Pico's primary UART (uart0) to read characters from our computer and send them via the other UART (uart1) to print them onto the LCD. We also define a special startup sequence and vary the display's backlight color.

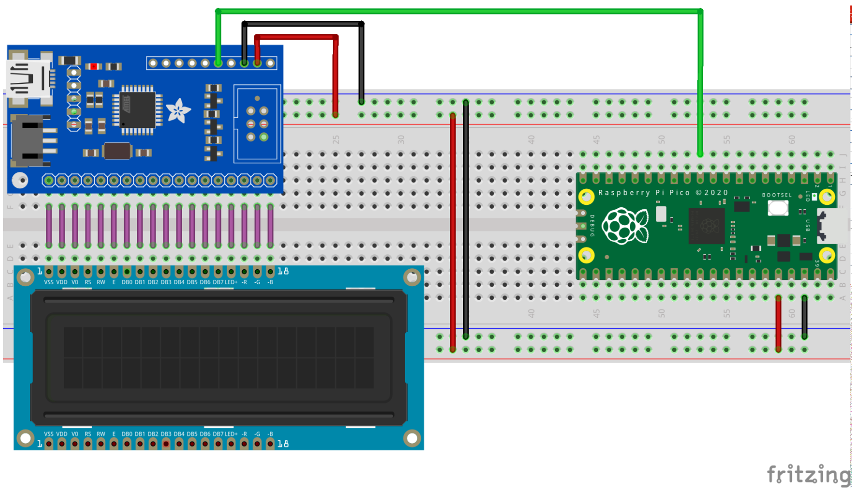
> **NOTE**
>
> You can change where stdio output goes (Pico's USB, uart0 or both) with CMake directives. The CMakeLists.txt file shows how to enable both.

Wiring information

Wiring up the backpack to the Pico requires 3 jumpers, to connect VCC (3.3v), GND, TX. The example here uses both of the Pico's UARTs, one (uart0) for stdio and the other (uart1) for communication with the backpack. Pin 8 is used as the TX pin. Power is supplied from the 3.3V pin. To connect the backpack to the display, it is common practice to solder it onto the back of the display, or during the prototyping stage to use the same parallel lanes on a breadboard.

> **NOTE**
>
> While this display will work at 3.3V, it will be quite dim. Using a 5V source will make it brighter.

Figure 12. Wiring Diagram for LCD with TTL backpack.



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/blob/master/uart/lcd_uart/CMakeLists.txt

```
add_executable(lcd_uart
        lcd_uart.c
        )

# pull in common dependencies and additional uart hardware support
target_link_libraries(lcd_uart pico_stdlib hardware_uart)

# enable usb output and uart output
# modify here as required
pico_enable_stdio_usb(lcd_uart 1)
pico_enable_stdio_uart(lcd_uart 1)

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

# add url via pico_set_program_url
```

```
example_auto_set_url(lcd_uart)
```

lcd_uart.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/uart/lcd_uart/lcd_uart.c

```
/**
 * Copyright (c) 2021 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */

/* Example code to drive a 16x2 LCD panel via an Adafruit TTL LCD "backpack"

   Optionally, the backpack can be connected the VBUS (pin 40) at 5V if
   the Pico in question is powered by USB for greater brightness.

   If this is done, then no other connections should be made to the backpack apart
   from those listed below as the backpack's logic levels will change.

   Connections on Raspberry Pi Pico board, other boards may vary.

   GPIO 8 (pin 11)-> RX on backpack
   3.3v (pin 36) -> 3.3v on backpack
   GND (pin 38)  -> GND on backpack
*/

#include <stdio.h>
#include <math.h>
#include "pico/stdlib.h"
#include "pico/binary_info.h"
#include "hardware/uart.h"

 // leave uart0 free for stdio
#define UART_ID uart1
#define BAUD_RATE 9600
#define UART_TX_PIN 8
#define LCD_WIDTH 16
#define LCD_HEIGHT 2

// basic commands
#define LCD_DISPLAY_ON 0x42
#define LCD_DISPLAY_OFF 0x46
#define LCD_SET_BRIGHTNESS 0x99
#define LCD_SET_CONTRAST 0x50
#define LCD_AUTOSCROLL_ON 0x51
#define LCD_AUTOSCROLL_OFF 0x52
#define LCD_CLEAR_SCREEN 0x58
#define LCD_SET_SPLASH 0x40

// cursor commands
#define LCD_SET_CURSOR_POS 0x47
#define LCD_CURSOR_HOME 0x48
#define LCD_CURSOR_BACK 0x4C
#define LCD_CURSOR_FORWARD 0x4D
#define LCD_UNDERLINE_CURSOR_ON 0x4A
#define LCD_UNDERLINE_CURSOR_OFF 0x4B
#define LCD_BLOCK_CURSOR_ON 0x53
#define LCD_BLOCK_CURSOR_OFF 0x54

// rgb commands
```

```c
#define LCD_SET_BACKLIGHT_COLOR 0xD0
#define LCD_SET_DISPLAY_SIZE 0xD1

// change to 0 if display is not RGB capable
#define LCD_IS_RGB 1

void lcd_write(uint8_t cmd, uint8_t* buf, uint8_t buflen) {
    // all commands are prefixed with 0xFE
    const uint8_t pre = 0xFE;
    uart_write_blocking(UART_ID, &pre, 1);
    uart_write_blocking(UART_ID, &cmd, 1);
    uart_write_blocking(UART_ID, buf, buflen);
    sleep_ms(10); // give the display some time
}

void lcd_set_size(uint8_t w, uint8_t h) {
    // sets the dimensions of the display
    uint8_t buf[] = { w, h };
    lcd_write(LCD_SET_DISPLAY_SIZE, buf, 2);
}

void lcd_set_contrast(uint8_t contrast) {
    // sets the display contrast
    lcd_write(LCD_SET_CONTRAST, &contrast, 1);
}

void lcd_set_brightness(uint8_t brightness) {
    // sets the backlight brightness
    lcd_write(LCD_SET_BRIGHTNESS, &brightness, 1);
}

void lcd_set_cursor(bool is_on) {
    // set is_on to true if we want the blinking block and underline cursor to show
    if (is_on) {
        lcd_write(LCD_BLOCK_CURSOR_ON, NULL, 0);
        lcd_write(LCD_UNDERLINE_CURSOR_ON, NULL, 0);
    } else {
        lcd_write(LCD_BLOCK_CURSOR_OFF, NULL, 0);
        lcd_write(LCD_UNDERLINE_CURSOR_OFF, NULL, 0);
    }
}

void lcd_set_backlight(bool is_on) {
    // turn the backlight on (true) or off (false)
    if (is_on) {
        lcd_write(LCD_DISPLAY_ON, (uint8_t *) 0, 1);
    } else {
        lcd_write(LCD_DISPLAY_OFF, NULL, 0);
    }
}

void lcd_clear() {
    // clear the contents of the display
    lcd_write(LCD_CLEAR_SCREEN, NULL, 0);
}

void lcd_cursor_reset() {
    // reset the cursor to (1, 1)
    lcd_write(LCD_CURSOR_HOME, NULL, 0);
}

#if LCD_IS_RGB
void lcd_set_backlight_color(uint8_t r, uint8_t g, uint8_t b) {
    // only supported on RGB displays!
```

```
    uint8_t buf[] = { r, g, b };
    lcd_write(LCD_SET_BACKLIGHT_COLOR, buf, 3);
}
#endif

void lcd_init() {
    lcd_set_backlight(true);
    lcd_set_size(LCD_WIDTH, LCD_HEIGHT);
    lcd_set_contrast(155);
    lcd_set_brightness(255);
    lcd_set_cursor(false);
}

int main() {
    stdio_init_all();
    uart_init(UART_ID, BAUD_RATE);
    uart_set_translate_crlf(UART_ID, false);
    gpio_set_function(UART_TX_PIN, UART_FUNCSEL_NUM(UART_ID, UART_TX_PIN));

    bi_decl(bi_1pin_with_func(UART_TX_PIN, UART_FUNCSEL_NUM(UART_ID, UART_TX_PIN)));

    lcd_init();

    // define startup sequence and save to EEPROM
    // no more or less than 32 chars, if not enough, fill remaining ones with spaces
    uint8_t splash_buf[] = "Hello LCD, from Pi Towers!      ";
    lcd_write(LCD_SET_SPLASH, splash_buf, LCD_WIDTH * LCD_HEIGHT);

    lcd_cursor_reset();
    lcd_clear();

#if LCD_IS_RGB
    uint8_t i = 0; // it's ok if this overflows and wraps, we're using sin
    const float frequency = 0.1f;
    uint8_t red, green, blue;
#endif

    while (1) {
        // send any chars from stdio straight to the backpack
        char c = getchar();
        // any bytes not followed by 0xFE (the special command) are interpreted
        // as text to be displayed on the backpack, so we just send the char
        // down the UART byte pipe!
        if (c < 128) uart_putc_raw(UART_ID, c); // skip extra non-ASCII chars
#if LCD_IS_RGB
        // change the display color on keypress, rainbow style!
        red = (uint8_t)(sin(frequency * i + 0) * 127 + 128);
        green = (uint8_t)(sin(frequency * i + 2) * 127 + 128);
        blue = (uint8_t)(sin(frequency * i + 4) * 127 + 128);
        lcd_set_backlight_color(red, green, blue);
        i++;
#endif
    }
}
```

Bill of Materials

Table 38. A list of materials required for the example

Item	Quantity	Details
Breadboard	1	generic part

Raspberry Pi Pico	1	https://www.raspberrypi.com/products/raspberry-pi-pico/
16x2 RGB LCD panel 3.3v	1	generic part, available on Adafruit
16x2 LCD backpack	1	from Adafruit
M/M Jumper wires	3	generic part

Attaching a microphone using the ADC

This example code shows how to interface the Raspberry Pi Pico with a standard analog microphone via the onboard analog to digital converter (ADC). In this example, we use an ICS-40180 breakout board by SparkFun but any analog microphone should be compatible with this tutorial. SparkFun have written a guide for this board that goes into more detail about the board and how it works.

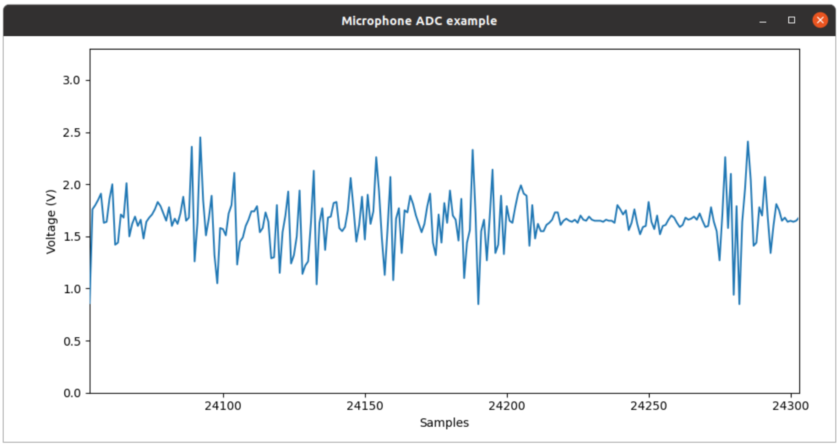
TIP

An analog to digital converter (ADC) is responsible for reading continually varying input signals that may range from 0 to a specified reference voltage (in the Pico's case this reference voltage is set by the supply voltage and can be measured on pin 35, ADC_VREF) and converting them into binary, i.e. a number that can be digitally stored.

The Pico has a 12-bit ADC (ENOB of 8.7-bit, see RP2040 datasheet section 4.9.3 for more details), meaning that a read operation will return a number ranging from 0 to 4095 (2^12 - 1) for a total of 4096 possible values. Therefore, the resolution of the ADC is 3.3/4096, so roughly steps of 0.8 millivolts. The SparkFun breakout uses an OPA344 operational amplifier to boost the signal coming from the microphone to voltage levels that can be easily read by the ADC. An important side effect is that a bias of 0.5*Vcc is added to the signal, even when the microphone is not picking up any sound.

The ADC provides us with a raw voltage value but when dealing with sound, we're more interested in the amplitude of the audio signal. This is defined as one half the peak-to-peak amplitude. Included with this example is a very simple Python script that will plot the voltage values it receives via the serial port. By tweaking the sampling rates, and various other parameters, the data from the microphone can be analysed in various ways, such as in a Fast Fourier Transform to see what frequencies make up the signal.

Figure 13. Example output from included Python script



Wiring information

Wiring up the device requires 3 jumpers, to connect VCC (3.3v), GND, and AOUT. The example here uses ADC0, which is GP26. Power is supplied from the 3.3V pin.

WARNING

Most boards will take a range of VCC voltages from the Pico's default 3.3V to the 5 volts commonly seen on other microcontrollers. Ensure your board doesn't output an analogue signal greater than 3.3V as this may result in permanent damage to the Pico's ADC.

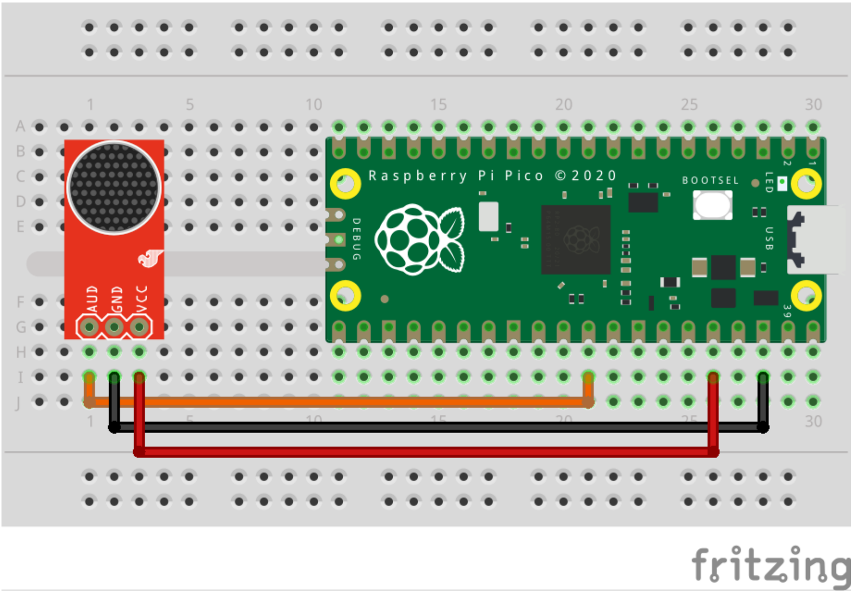


Figure 14. Wiring Diagram for ICS-40180 microphone breakout board.

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/blob/master/adc/microphone_adc/CMakeLists.txt

```
add_executable(microphone_adc
        microphone_adc.c
        )

# pull in common dependencies and adc hardware support
target_link_libraries(microphone_adc pico_stdlib hardware_adc)

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

# add url via pico_set_program_url
example_auto_set_url(microphone_adc)
```

microphone_adc.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/adc/microphone_adc/microphone_adc.c

```
/**
 * Copyright (c) 2021 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */

```

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

/* Example code to extract analog values from a microphone using the ADC
   with accompanying Python file to plot these values

   Connections on Raspberry Pi Pico board, other boards may vary.

   GPIO 26/ADC0 (pin 31)-> AOUT or AUD on microphone board
   3.3v (pin 36) -> VCC on microphone board
   GND (pin 38)  -> GND on microphone board
*/

#define ADC_NUM 0
#define ADC_PIN (26 + ADC_NUM)
#define ADC_VREF 3.3
#define ADC_RANGE (1 << 12)
#define ADC_CONVERT (ADC_VREF / (ADC_RANGE - 1))

int main() {
    stdio_init_all();
    printf("Beep boop, listening...\n");

    bi_decl(bi_program_description("Analog microphone example for Raspberry Pi Pico")); // for picotool
    bi_decl(bi_1pin_with_name(ADC_PIN, "ADC input pin"));

    adc_init();
    adc_gpio_init( ADC_PIN);
    adc_select_input( ADC_NUM);

    uint adc_raw;
    while (1) {
        adc_raw = adc_read(); // raw voltage from ADC
        printf("%.2f\n", adc_raw * ADC_CONVERT);
        sleep_ms(10);
    }
}
```

Bill of Materials

Table 39. A list of materials required for the example

Item	Quantity	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	https://www.raspberrypi.com/products/raspberry-pi-pico/
ICS-40180 microphone breakout board or similar	1	From SparkFun
M/M Jumper wires	3	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 appropriate compensation 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.

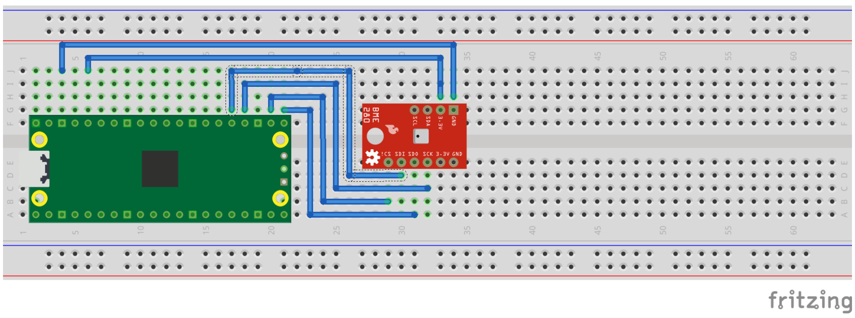


Figure 15. Wiring Diagram for bme280.

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/blob/master/spi/bme280_spi/CMakeLists.txt

```
add_executable(bme280_spi
        bme280_spi.c
        )

# pull in common dependencies and additional spi hardware support
target_link_libraries(bme280_spi pico_stdlib hardware_spi)
```

```

# create map/bin/hex file etc.
pico_add_extra_outputs(bme280_spi)

# add url via pico_set_program_url
example_auto_set_url(bme280_spi)
```

bme280_spi.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/spi/bme280_spi/bme280_spi.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 "pico/binary_info.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 therefore 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 36) -> 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 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;
```

```c
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 in these 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);
}

```

```c
#ifdef PICO_DEFAULT_SPI_CSN_PIN
static inline void cs_select() {
    asm volatile("nop \n nop \n nop");
    gpio_put(PICO_DEFAULT_SPI_CSN_PIN, 0);  // Active low
    asm volatile("nop \n nop \n nop");
}

static inline void cs_deselect() {
    asm volatile("nop \n nop \n nop");
    gpio_put(PICO_DEFAULT_SPI_CSN_PIN, 1);
    asm volatile("nop \n nop \n nop");
}
#endif

#if defined(spi_default) && defined(PICO_DEFAULT_SPI_CSN_PIN)
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_default, 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_default, &reg, 1);
    sleep_ms(10);
    spi_read_blocking(spi_default, 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, 26);

    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]; // 0xA1

    read_registers(0xE1, buffer, 8);

    dig_H2 = buffer[0] | (buffer[1] << 8); // 0xE1 | 0xE2
    dig_H3 = (int8_t) buffer[2]; // 0xE3
```

```
    dig_H4 = buffer[3] << 4 | (buffer[4] & 0xf); // 0xE4 | 0xE5[3:0]
    dig_H5 = (buffer[4] >> 4) | (buffer[5] << 4); // 0xE5[7:4] | 0xE6
    dig_H6 = (int8_t) buffer[6]; // 0xE7
}

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];
}
#endif

int main() {
    stdio_init_all();
#if !defined(spi_default) || !defined(PICO_DEFAULT_SPI_SCK_PIN) ||
!defined(PICO_DEFAULT_SPI_TX_PIN) || !defined(PICO_DEFAULT_SPI_RX_PIN) ||
!defined(PICO_DEFAULT_SPI_CSN_PIN)
#warning spi/bme280_spi example requires a board with SPI pins
    puts("Default SPI pins were not defined");
#else

    printf("Hello, bme280! Reading raw data from registers via SPI...\n");

    // This example will use SPI0 at 0.5MHz.
    spi_init(spi_default, 500 * 1000);
    gpio_set_function(PICO_DEFAULT_SPI_RX_PIN, GPIO_FUNC_SPI);
    gpio_set_function(PICO_DEFAULT_SPI_SCK_PIN, GPIO_FUNC_SPI);
    gpio_set_function(PICO_DEFAULT_SPI_TX_PIN, GPIO_FUNC_SPI);
    // Make the SPI pins available to picotool
    bi_decl(bi_3pins_with_func(PICO_DEFAULT_SPI_RX_PIN, PICO_DEFAULT_SPI_TX_PIN,
PICO_DEFAULT_SPI_SCK_PIN, GPIO_FUNC_SPI));

    // Chip select is active-low, so we'll initialise it to a driven-high state
    gpio_init(PICO_DEFAULT_SPI_CSN_PIN);
    gpio_set_dir(PICO_DEFAULT_SPI_CSN_PIN, GPIO_OUT);
    gpio_put(PICO_DEFAULT_SPI_CSN_PIN, 1);
    // Make the CS pin available to picotool
    bi_decl(bi_1pin_with_name(PICO_DEFAULT_SPI_CSN_PIN, "SPI CS"));

    // 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
        temperature = compensate_temp(temperature);
        pressure = compensate_pressure(pressure);
        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);
    }
#endif
}
```

Bill of Materials

Table 40. A list of materials required for the example

Item	Quantity	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	https://www.raspberrypi.com/products/raspberry-pi-pico/
BME280 board	1	generic part
M/M Jumper wires	6	generic part

Attaching a MPU9250 accelerometer/gyroscope 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.

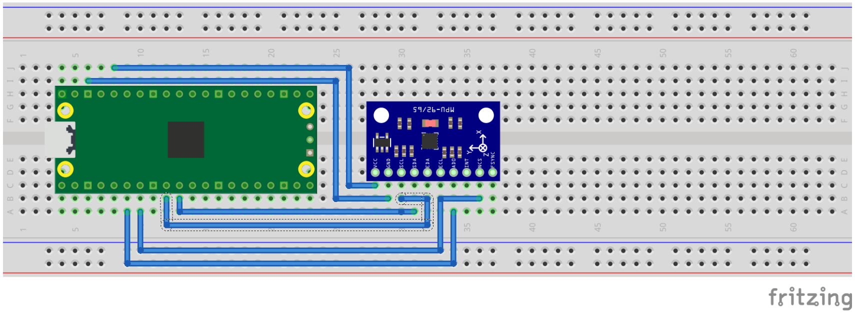


Figure 16. Wiring Diagram for MPU9250.

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/blob/master/spi/mpu9250_spi/CMakeLists.txt

```
add_executable(mpu9250_spi
        mpu9250_spi.c
        )

# pull in common dependencies and additional spi hardware support
target_link_libraries(mpu9250_spi pico_stdlib hardware_spi)

# create map/bin/hex file etc.
pico_add_extra_outputs(mpu9250_spi)

# add url via pico_set_program_url
example_auto_set_url(mpu9250_spi)
```

mpu9250_spi.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/spi/mpu9250_spi/mpu9250_spi.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 "pico/binary_info.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 therefore 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);
    // Make the SPI pins available to picotool
    bi_decl(bi_3pins_with_func(PIN_MISO, PIN_MOSI, PIN_SCK, 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);
    // Make the CS pin available to picotool
    bi_decl(bi_1pin_with_name(PIN_CS, "SPI CS"));

    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);
    }
}
```

Bill of Materials

Table 41. A list of materials required for the example

Item	Quantity	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	https://www.raspberrypi.com/products/raspberry-pi-pico/
MPU9250 board	1	generic part
M/M Jumper wires	6	generic part

Attaching a MPU6050 accelerometer/gyroscope 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.

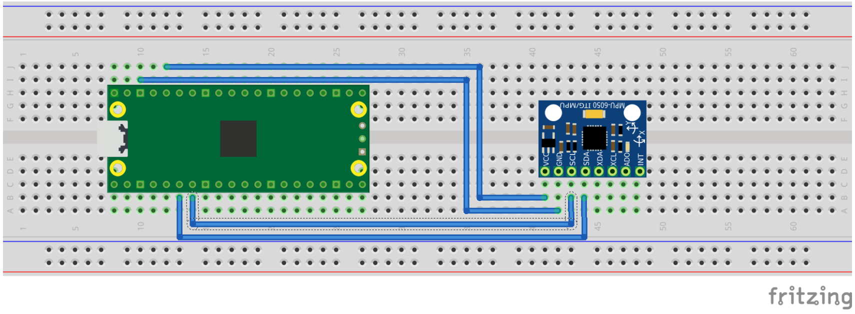


Figure 17. Wiring Diagram for MPU6050.

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/blob/master/i2c/mpu6050_i2c/CMakeLists.txt

```
add_executable(mpu6050_i2c
        mpu6050_i2c.c
        )

# pull in common dependencies and additional i2c hardware support
target_link_libraries(mpu6050_i2c pico_stdlib hardware_i2c)

# create map/bin/hex file etc.
pico_add_extra_outputs(mpu6050_i2c)

# add url via pico_set_program_url
example_auto_set_url(mpu6050_i2c)
```

mpu6050_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/mpu6050_i2c/mpu6050_i2c.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 "pico/binary_info.h"
#include "hardware/i2c.h"

/* Example code to talk to a MPU6050 MEMS accelerometer 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 therefore 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 PICO_DEFAULT_I2C_SDA_PIN (On Pico this is GP4 (pin 6)) -> SDA on MPU6050 board
   GPIO PICO_DEFAULT_I2C_SCL_PIN (On Pico this is GP5 (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;

#ifdef i2c_default
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, 0x80};
    i2c_write_blocking(i2c_default, addr, buf, 2, false);
    sleep_ms(100); // Allow device to reset and stabilize

    // Clear sleep mode (0x6B register, 0x00 value)
    buf[1] = 0x00;  // Clear sleep mode by writing 0x00 to the 0x6B register
    i2c_write_blocking(i2c_default, addr, buf, 2, false);
    sleep_ms(10); // Allow stabilization after waking up
}

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 don't 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_default, addr, &val, 1, true); // true to keep master control of bus
    i2c_read_blocking(i2c_default, 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_default, addr, &val, 1, true);
    i2c_read_blocking(i2c_default, 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_default, addr, &val, 1, true);
    i2c_read_blocking(i2c_default, addr, buffer, 2, false);  // False - finished with bus

    *temp = buffer[0] << 8 | buffer[1];
}
#endif
```

```
int main() {
    stdio_init_all();
#if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) ||
!defined(PICO_DEFAULT_I2C_SCL_PIN)
    #warning i2c/mpu6050_i2c example requires a board with I2C pins
    puts("Default I2C pins were not defined");
    return 0;
#else
    printf("Hello, MPU6050! Reading raw data from registers...\n");

    // This example will use I2C0 on the default SDA and SCL pins (4, 5 on a Pico)
    i2c_init(i2c_default, 400 * 1000);
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
    gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);
    // Make the I2C pins available to picotool
    bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN,
GPIO_FUNC_I2C));

    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);
    }
#endif
}
```

Bill of Materials

Table 42. A list of materials required for the example

Item	Quantity	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	https://www.raspberrypi.com/products/raspberry-pi-pico/
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.

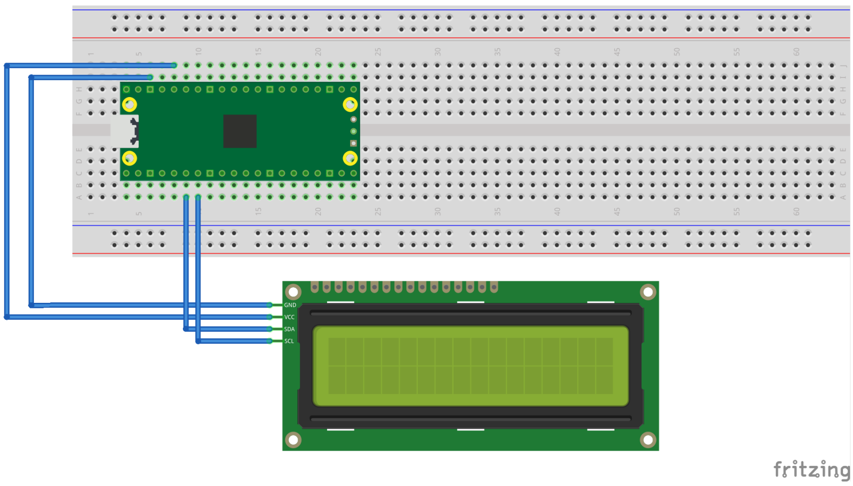


Figure 18. Wiring Diagram for LCD1602A LCD with I2C bridge.

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/blob/master/i2c/lcd_1602_i2c/CMakeLists.txt

```
add_executable(lcd_1602_i2c
        lcd_1602_i2c.c
        )

# pull in common dependencies and additional i2c hardware support
target_link_libraries(lcd_1602_i2c pico_stdlib hardware_i2c)

# create map/bin/hex file etc.
pico_add_extra_outputs(lcd_1602_i2c)

# add url via pico_set_program_url
example_auto_set_url(lcd_1602_i2c)
```

lcd_1602_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/lcd_1602_i2c/lcd_1602_i2c.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/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 therefore 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;

// 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 function for single byte transfers */
void i2c_write_byte(uint8_t val) {
#ifdef i2c_default
    i2c_write_blocking(i2c_default, addr, &val, 1, false);
#endif
}

void lcd_toggle_enable(uint8_t val) {
    // Toggle enable pin on LCD display
    // We cannot do this too quickly or things don't 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);
}

static inline void 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() {
#if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) || !defined(PICO_DEFAULT_I2C_SCL_PIN)
    #warning i2c/lcd_1602_i2c example requires a board with I2C pins
#else
    // This example will use I2C0 on the default SDA and SCL pins (4, 5 on a Pico)
    i2c_init(i2c_default, 100 * 1000);
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
    gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);
    // Make the I2C pins available to picotool
    bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C));

    lcd_init();

    static char *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 (uint 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();
        }
    }
#endif
}
```

Bill of Materials

Table 43. A list of materials required for the example

Item	Quantity	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	https://www.raspberrypi.com/products/raspberry-pi-pico/
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

Attaching a BMP280 temp/pressure sensor via I2C

This example code shows how to interface the Raspberry Pi Pico with the popular BMP280 temperature and air pressure sensor manufactured by Bosch. A similar variant, the BME280, exists that can also measure humidity. There is another example that uses the BME280 device but talks to it via SPI as opposed to I2C.

The code reads data from the sensor's registers every 500 milliseconds and prints it via the onboard UART. This example operates the BMP280 in *normal* mode, meaning that the device continuously cycles between a measurement period and a standby period at a regular interval we can set. This has the advantage that subsequent reads do not require configuration register writes and is the recommended mode of operation to filter out short-term disturbances.

TIP

> The BMP280 is highly configurable with 3 modes of operation, various oversampling levels, and 5 filter settings. Find the datasheet online (https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bmp280-ds001.pdf) to explore all of its capabilities beyond the simple example given here.

Wiring information

Wiring up the device requires 4 jumpers, to connect VCC (3.3v), GND, SDA and SCL. The example here uses the default I2C port 0, which is assigned to GPIO 4 (SDA) and 5 (SCL) in software. Power is supplied from the 3.3V pin from the Pico.

WARNING

> The BMP280 has a maximum supply voltage rating of 3.6V. Most breakout boards have voltage regulators that will allow a range of input voltages of 2-6V, but make sure to check beforehand.

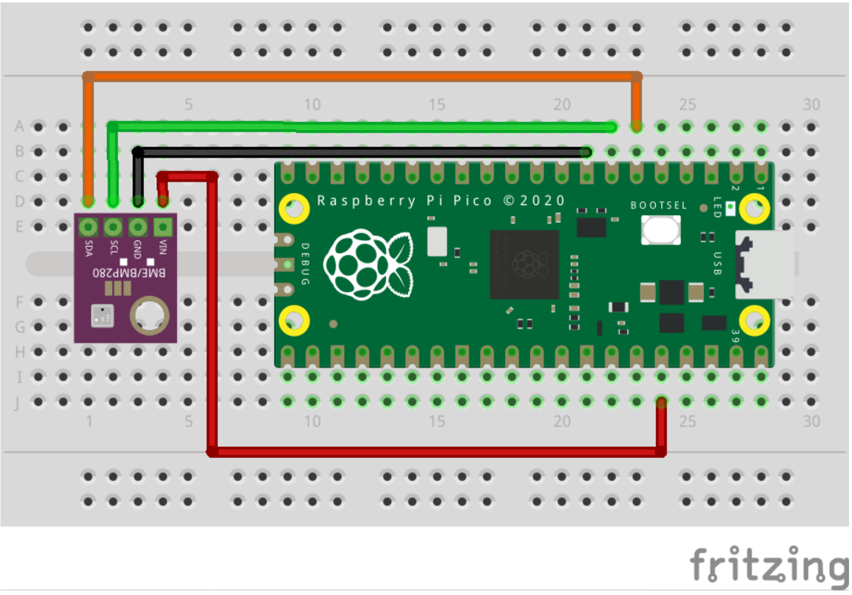


Figure 19. Wiring Diagram for BMP280 sensor via I2C.

List of Files

CMakeLists.txt

CMake file to incorporate the example into the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/bmp280_i2c/CMakeLists.txt

```
add_executable(bmp280_i2c
        bmp280_i2c.c
        )

# pull in common dependencies and additional i2c hardware support
target_link_libraries(bmp280_i2c pico_stdlib hardware_i2c)

# create map/bin/hex file etc.
pico_add_extra_outputs(bmp280_i2c)

# add url via pico_set_program_url
example_auto_set_url(bmp280_i2c)
```

bmp280_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/bmp280_i2c/bmp280_i2c.c

```
/**
 * Copyright (c) 2021 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 **/

#include <stdio.h>

#include "hardware/i2c.h"
#include "pico/binary_info.h"
#include "pico/stdlib.h"

 /* Example code to talk to a BMP280 temperature and pressure sensor

    NOTE: Ensure the device is capable of being driven at 3.3v NOT 5v. The Pico
    GPIO (and therefore 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 PICO_DEFAULT_I2C_SDA_PIN (on Pico this is GP4 (pin 6)) -> SDA on BMP280
    board
    GPIO PICO_DEFAULT_I2C_SCK_PIN (on Pico this is GP5 (pin 7)) -> SCL on
    BMP280 board
    3.3v (pin 36) -> VCC on BMP280 board
    GND (pin 38)  -> GND on BMP280 board
 */

 // device has default bus address of 0x76
#define ADDR _u(0x76)

// hardware registers
#define REG_CONFIG _u(0xF5)
#define REG_CTRL_MEAS _u(0xF4)
#define REG_RESET _u(0xE0)

```

```
#define REG_TEMP_XLSB _u(0xFC)
#define REG_TEMP_LSB _u(0xFB)
#define REG_TEMP_MSB _u(0xFA)

#define REG_PRESSURE_XLSB _u(0xF9)
#define REG_PRESSURE_LSB _u(0xF8)
#define REG_PRESSURE_MSB _u(0xF7)

// calibration registers
#define REG_DIG_T1_LSB _u(0x88)
#define REG_DIG_T1_MSB _u(0x89)
#define REG_DIG_T2_LSB _u(0x8A)
#define REG_DIG_T2_MSB _u(0x8B)
#define REG_DIG_T3_LSB _u(0x8C)
#define REG_DIG_T3_MSB _u(0x8D)
#define REG_DIG_P1_LSB _u(0x8E)
#define REG_DIG_P1_MSB _u(0x8F)
#define REG_DIG_P2_LSB _u(0x90)
#define REG_DIG_P2_MSB _u(0x91)
#define REG_DIG_P3_LSB _u(0x92)
#define REG_DIG_P3_MSB _u(0x93)
#define REG_DIG_P4_LSB _u(0x94)
#define REG_DIG_P4_MSB _u(0x95)
#define REG_DIG_P5_LSB _u(0x96)
#define REG_DIG_P5_MSB _u(0x97)
#define REG_DIG_P6_LSB _u(0x98)
#define REG_DIG_P6_MSB _u(0x99)
#define REG_DIG_P7_LSB _u(0x9A)
#define REG_DIG_P7_MSB _u(0x9B)
#define REG_DIG_P8_LSB _u(0x9C)
#define REG_DIG_P8_MSB _u(0x9D)
#define REG_DIG_P9_LSB _u(0x9E)
#define REG_DIG_P9_MSB _u(0x9F)

// number of calibration registers to be read
#define NUM_CALIB_PARAMS 24

struct bmp280_calib_param {
    // temperature params
    uint16_t dig_t1;
    int16_t dig_t2;
    int16_t dig_t3;

    // pressure params
    uint16_t dig_p1;
    int16_t dig_p2;
    int16_t dig_p3;
    int16_t dig_p4;
    int16_t dig_p5;
    int16_t dig_p6;
    int16_t dig_p7;
    int16_t dig_p8;
    int16_t dig_p9;
};

#ifdef i2c_default
void bmp280_init() {
    // use the "handheld device dynamic" optimal setting (see datasheet)
    uint8_t buf[2];

    // 500ms sampling time, x16 filter
    const uint8_t reg_config_val = ((0x04 << 5) | (0x05 << 2)) & 0xFC;

    // send register number followed by its corresponding value
```

```
    buf[0] = REG_CONFIG;
    buf[1] = reg_config_val;
    i2c_write_blocking(i2c_default, ADDR, buf, 2, false);

    // osrs_t x1, osrs_p x4, normal mode operation
    const uint8_t reg_ctrl_meas_val = (0x01 << 5) | (0x03 << 2) | (0x03);
    buf[0] = REG_CTRL_MEAS;
    buf[1] = reg_ctrl_meas_val;
    i2c_write_blocking(i2c_default, ADDR, buf, 2, false);
}

void bmp280_read_raw(int32_t* temp, int32_t* pressure) {
    // BMP280 data registers are auto-incrementing and we have 3 temperature and
    // pressure registers each, so we start at 0xF7 and read 6 bytes to 0xFC
    // note: normal mode does not require further ctrl_meas and config register writes

    uint8_t buf[6];
    uint8_t reg = REG_PRESSURE_MSB;
    i2c_write_blocking(i2c_default, ADDR, &reg, 1, true);  // true to keep master control of bus
    i2c_read_blocking(i2c_default, ADDR, buf, 6, false);  // false - finished with bus

    // store the 20 bit read in a 32 bit signed integer for conversion
    *pressure = (buf[0] << 12) | (buf[1] << 4) | (buf[2] >> 4);
    *temp = (buf[3] << 12) | (buf[4] << 4) | (buf[5] >> 4);
}

void bmp280_reset() {
    // reset the device with the power-on-reset procedure
    uint8_t buf[2] = { REG_RESET, 0xB6 };
    i2c_write_blocking(i2c_default, ADDR, buf, 2, false);
}

// intermediate function that calculates the fine resolution temperature
// used for both pressure and temperature conversions
int32_t bmp280_convert(int32_t temp, struct bmp280_calib_param* params) {
    // use the 32-bit fixed point compensation implementation given in the
    // datasheet

    int32_t var1, var2;
    var1 = ((((temp >> 3) - ((int32_t)params->dig_t1 << 1))) * ((int32_t)params->dig_t2)) >> 11;
    var2 = (((((temp >> 4) - ((int32_t)params->dig_t1)) * ((temp >> 4) - ((int32_t)params->dig_t1))) >> 12) * ((int32_t)params->dig_t3)) >> 14;
    return var1 + var2;
}

int32_t bmp280_convert_temp(int32_t temp, struct bmp280_calib_param* params) {
    // uses the BMP280 calibration parameters to compensate the temperature value read from its registers
    int32_t t_fine = bmp280_convert(temp, params);
    return (t_fine * 5 + 128) >> 8;
}

int32_t bmp280_convert_pressure(int32_t pressure, int32_t temp, struct bmp280_calib_param* params) {
    // uses the BMP280 calibration parameters to compensate the pressure value read from its registers

    int32_t t_fine = bmp280_convert(temp, params);

    int32_t var1, var2;
    uint32_t converted = 0.0;
    var1 = (((int32_t)t_fine) >> 1) - (int32_t)64000;
```

```
    var2 = (((var1 >> 2) * (var1 >> 2)) >> 11) * ((int32_t)params->dig_p6);
    var2 += ((var1 * ((int32_t)params->dig_p5)) << 1);
    var2 = (var2 >> 2) + (((int32_t)params->dig_p4) << 16);
    var1 = (((params->dig_p3 * (((var1 >> 2) * (var1 >> 2)) >> 13)) >> 3) + ((((int32_t)params->dig_p2) * var1) >> 1)) >> 18;
    var1 = ((((32768 + var1)) * ((int32_t)params->dig_p1)) >> 15);
    if (var1 == 0) {
        return 0;  // avoid exception caused by division by zero
    }
    converted = (((uint32_t)(((int32_t)1048576) - pressure) - (var2 >> 12))) * 3125;
    if (converted < 0x80000000) {
        converted = (converted << 1) / ((uint32_t)var1);
    } else {
        converted = (converted / (uint32_t)var1) * 2;
    }
    var1 = (((int32_t)params->dig_p9) * ((int32_t)(((converted >> 3) * (converted >> 3)) >> 13))) >> 12;
    var2 = (((int32_t)(converted >> 2)) * ((int32_t)params->dig_p8)) >> 13;
    converted = (uint32_t)((int32_t)converted + ((var1 + var2 + params->dig_p7) >> 4));
    return converted;
}

void bmp280_get_calib_params(struct bmp280_calib_param* params) {
    // raw temp and pressure values need to be calibrated according to
    // parameters generated during the manufacturing of the sensor
    // there are 3 temperature params, and 9 pressure params, each with a LSB
    // and MSB register, so we read from 24 registers

    uint8_t buf[NUM_CALIB_PARAMS] = { 0 };
    uint8_t reg = REG_DIG_T1_LSB;
    i2c_write_blocking(i2c_default, ADDR, &reg, 1, true);  // true to keep master control of bus
    // read in one go as register addresses auto-increment
    i2c_read_blocking(i2c_default, ADDR, buf, NUM_CALIB_PARAMS, false);  // false, we're done reading

    // store these in a struct for later use
    params->dig_t1 = (uint16_t)(buf[1] << 8) | buf[0];
    params->dig_t2 = (int16_t)(buf[3] << 8) | buf[2];
    params->dig_t3 = (int16_t)(buf[5] << 8) | buf[4];

    params->dig_p1 = (uint16_t)(buf[7] << 8) | buf[6];
    params->dig_p2 = (int16_t)(buf[9] << 8) | buf[8];
    params->dig_p3 = (int16_t)(buf[11] << 8) | buf[10];
    params->dig_p4 = (int16_t)(buf[13] << 8) | buf[12];
    params->dig_p5 = (int16_t)(buf[15] << 8) | buf[14];
    params->dig_p6 = (int16_t)(buf[17] << 8) | buf[16];
    params->dig_p7 = (int16_t)(buf[19] << 8) | buf[18];
    params->dig_p8 = (int16_t)(buf[21] << 8) | buf[20];
    params->dig_p9 = (int16_t)(buf[23] << 8) | buf[22];
}

#endif

int main() {
    stdio_init_all();

#if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) || !defined(PICO_DEFAULT_I2C_SCL_PIN)
    #warning i2c / bmp280_i2c example requires a board with I2C pins
        puts("Default I2C pins were not defined");
    return 0;
#else
    // useful information for picotool
```

```
    bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C));
    bi_decl(bi_program_description("BMP280 I2C example for the Raspberry Pi Pico"));

    printf("Hello, BMP280! Reading temperaure and pressure values from sensor...\n");

    // I2C is "open drain", pull ups to keep signal high when no data is being sent
    i2c_init(i2c_default, 100 * 1000);
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
    gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);

    // configure BMP280
    bmp280_init();

    // retrieve fixed compensation params
    struct bmp280_calib_param params;
    bmp280_get_calib_params(&params);

    int32_t raw_temperature;
    int32_t raw_pressure;

    sleep_ms(250); // sleep so that data polling and register update don't collide
    while (1) {
        bmp280_read_raw(&raw_temperature, &raw_pressure);
        int32_t temperature = bmp280_convert_temp(raw_temperature, &params);
        int32_t pressure = bmp280_convert_pressure(raw_pressure, raw_temperature, &params);
        printf("Pressure = %.3f kPa\n", pressure / 1000.f);
        printf("Temp. = %.2f C\n", temperature / 100.f);
        // poll every 500ms
        sleep_ms(500);
    }
#endif
}
```

Bill of Materials

Table 44. A list of materials required for the example

Item	**Quantity**	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	https://www.raspberrypi.com/products/raspberry-pi-pico/
BMP280-based breakout board	1	from Pimoroni
M/M Jumper wires	4	generic part

Attaching a LIS3DH Nano Accelerometer via i2c.

This example shows you how to interface the Raspberry Pi Pico to the LIS3DH accelerometer and temperature sensor.

The code reads and displays the acceleration values of the board in the 3 axes and the ambient temperature value. The datasheet for the sensor can be found at https://www.st.com/resource/en/datasheet/cd00274221.pdf. The device is being operated on 'normal mode' and at a frequency of 1.344 kHz (this can be changed by editing the ODR bits of CTRL_REG4). The range of the data is controlled by the FS bit in CTRL_REG4 and is equal to ±2g in this example. The sensitivity depends on the operating mode and data range; exact values can be found on page 10 of the datasheet. In this case, the sensitivity value is 4mg (where g is the value of gravitational acceleration on the surface of Earth). In order to use the auxiliary ADC to read temperature, the we must set the BDU bit to 1 in CTRL_REG4 and the ADC_EN bit to 1 in TEMP_CFG_REG. Temperature is communicated through ADC 3.

NOTE

The sensor doesn't have features to eliminate offsets in the data and these will need to be taken into account in the code.

Wiring information

Wiring up the device requires 4 jumpers, to connect VIN, 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 3V pin.

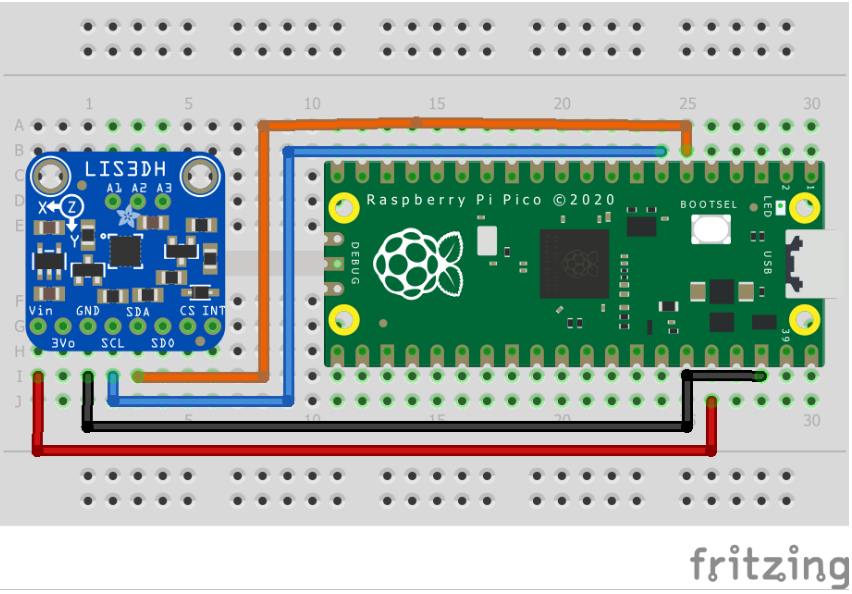


Figure 20. Wiring Diagram for LIS3DH.

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/blob/master/i2c/lis3dh_i2c/CMakeLists.txt

```
add_executable(lis3dh_i2c
        lis3dh_i2c.c
        )

# pull in common dependencies and additional i2c hardware support
target_link_libraries(lis3dh_i2c pico_stdlib hardware_i2c)

# create map/bin/hex file etc.
pico_add_extra_outputs(lis3dh_i2c)

# add url via pico_set_program_url
example_auto_set_url(lis3dh_i2c)
```

lis3dh_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/lis3dh_i2c/lis3dh_i2c.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 "pico/binary_info.h"
#include "hardware/i2c.h"

/* Example code to talk to a LIS3DH Mini GPS module.

   This example reads data from all 3 axes of the accelerometer and uses an auxiliary ADC to
output temperature values.

   Connections on Raspberry Pi Pico board, other boards may vary.

   GPIO PICO_DEFAULT_I2C_SDA_PIN (On Pico this is 4 (physical pin 6)) -> SDA on LIS3DH board
   GPIO PICO_DEFAULT_I2C_SCK_PIN (On Pico this is 5 (physical pin 7)) -> SCL on LIS3DH board
   3.3v (physical pin 36) -> VIN on LIS3DH board
   GND (physical pin 38)  -> GND on LIS3DH board
*/

// By default this device is on bus address 0x18

const int ADDRESS = 0x18;
const uint8_t CTRL_REG_1 = 0x20;
const uint8_t CTRL_REG_4 = 0x23;
const uint8_t TEMP_CFG_REG = 0x1F;

#ifdef i2c_default

void lis3dh_init() {
    uint8_t buf[2];

    // Turn normal mode and 1.344kHz data rate on
    buf[0] = CTRL_REG_1;
    buf[1] = 0x97;
    i2c_write_blocking(i2c_default, ADDRESS, buf, 2, false);

    // Turn block data update on (for temperature sensing)
    buf[0] = CTRL_REG_4;
    buf[1] = 0x80;
    i2c_write_blocking(i2c_default, ADDRESS, buf, 2, false);

    // Turn auxiliary ADC on
    buf[0] = TEMP_CFG_REG;
    buf[1] = 0xC0;
    i2c_write_blocking(i2c_default, ADDRESS, buf, 2, false);
}

void lis3dh_calc_value(uint16_t raw_value, float *final_value, bool isAccel) {
    // Convert with respect to the value being temperature or acceleration reading
    float scaling;
    float senstivity = 0.004f; // g per unit

```

```
    if (isAccel == true) {
        scaling = 64 / senstivity;
    } else {
        scaling = 64;
    }

    // raw_value is signed
    *final_value = (float) ((int16_t) raw_value) / scaling;
}

void lis3dh_read_data(uint8_t reg, float *final_value, bool IsAccel) {
    // Read two bytes of data and store in a 16 bit data structure
    uint8_t lsb;
    uint8_t msb;
    uint16_t raw_accel;
    i2c_write_blocking(i2c_default, ADDRESS, &reg, 1, true);
    i2c_read_blocking(i2c_default, ADDRESS, &lsb, 1, false);

    reg |= 0x01;
    i2c_write_blocking(i2c_default, ADDRESS, &reg, 1, true);
    i2c_read_blocking(i2c_default, ADDRESS, &msb, 1, false);

    raw_accel = (msb << 8) | lsb;

    lis3dh_calc_value(raw_accel, final_value, IsAccel);
}

#endif

int main() {
    stdio_init_all();
#if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) || !defined(PICO_DEFAULT_I2C_SCL_PIN)
#warning i2c/lis3dh_i2c example requires a board with I2C pins
    puts("Default I2C pins were not defined");
#else
    printf("Hello, LIS3DH! Reading raw data from registers...\n");

    // This example will use I2C0 on the default SDA and SCL pins (4, 5 on a Pico)
    i2c_init(i2c_default, 400 * 1000);
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
    gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);
    // Make the I2C pins available to picotool
    bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C));

    float x_accel, y_accel, z_accel, temp;

    lis3dh_init();

    while (1) {
        lis3dh_read_data(0x28, &x_accel, true);
        lis3dh_read_data(0x2A, &y_accel, true);
        lis3dh_read_data(0x2C, &z_accel, true);
        lis3dh_read_data(0x0C, &temp, false);

        // Display data
        printf("TEMPERATURE: %.3f%cC\n", temp, 176);
        // Acceleration is read as a multiple of g (gravitational acceleration on the Earth's surface)
        printf("ACCELERATION VALUES: \n");
        printf("X acceleration: %.3fg\n", x_accel);
```

```
        printf("Y acceleration: %.3fg\n", y_accel);
        printf("Z acceleration: %.3fg\n", z_accel);

        sleep_ms(500);

        // Clear terminal
        printf("\033[1;1H\033[2J");
    }
#endif
}
```

Bill of Materials

Table 45. A list of materials required for the example

Item	Quantity	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	https://www.raspberrypi.com/products/raspberry-pi-pico/
LIS3DH board	1	https://www.adafruit.com/product/2809
M/M Jumper wires	4	generic part

Attaching a MCP9808 digital temperature sensor via I2C

This example code shows how to interface the Raspberry Pi Pico to the MCP9808 digital temperature sensor board.

This example reads the ambient temperature value each second from the sensor and sets upper, lower and critical limits for the temperature and checks if alerts need to be raised. The CONFIG register can also be used to check for an alert if the critical temperature is surpassed.

Wiring information

Wiring up the device requires 4 jumpers, to connect VDD, 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 VSYS pin.

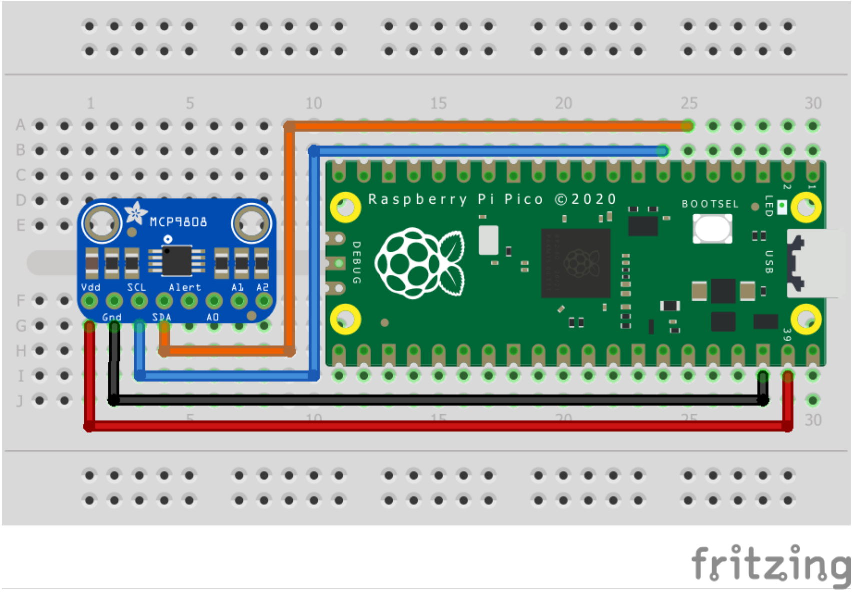


Figure 21. Wiring Diagram for MCP9808.

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/blob/master/i2c/mcp9808_i2c/CMakeLists.txt

```
add_executable(mcp9808_i2c
        mcp9808_i2c.c
        )

# pull in common dependencies and additional i2c hardware support
target_link_libraries(mcp9808_i2c pico_stdlib hardware_i2c)

# create map/bin/hex file etc.
pico_add_extra_outputs(mcp9808_i2c)

# add url via pico_set_program_url
example_auto_set_url(mcp9808_i2c)
```

mcp9808_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/mcp9808_i2c/mcp9808_i2c.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 "pico/binary_info.h"
#include "hardware/i2c.h"

/* Example code to talk to a MCP9808 ±0.5°C Digital temperature Sensor
```

```
   This reads and writes to registers on the board.

   Connections on Raspberry Pi Pico board, other boards may vary.

   GPIO PICO_DEFAULT_I2C_SDA_PIN (On Pico this is GP4 (physical pin 6)) -> SDA on MCP9808
board
   GPIO PICO_DEFAULT_I2C_SCK_PIN (On Pico this is GP5 (physical pin 7)) -> SCL on MCP9808
board
   Vsys (physical pin 39) -> VDD on MCP9808 board
   GND (physical pin 38)  -> GND on MCP9808 board

*/
//The bus address is determined by the state of pins A0, A1 and A2 on the MCP9808 board
static uint8_t ADDRESS = 0x18;

//hardware registers

const uint8_t REG_POINTER = 0x00;
const uint8_t REG_CONFIG = 0x01;
const uint8_t REG_TEMP_UPPER = 0x02;
const uint8_t REG_TEMP_LOWER = 0x03;
const uint8_t REG_TEMP_CRIT = 0x04;
const uint8_t REG_TEMP_AMB = 0x05;
const uint8_t REG_RESOLUTION = 0x08;


void mcp9808_check_limits(uint8_t upper_byte) {

    // Check flags and raise alerts accordingly
    if ((upper_byte & 0x40) == 0x40) { //TA > TUPPER
        printf("Temperature is above the upper temperature limit.\n");
    }
    if ((upper_byte & 0x20) == 0x20) { //TA < TLOWER
        printf("Temperature is below the lower temperature limit.\n");
    }
    if ((upper_byte & 0x80) == 0x80) { //TA > TCRIT
        printf("Temperature is above the critical temperature limit.\n");
    }
}

float mcp9808_convert_temp(uint8_t upper_byte, uint8_t lower_byte) {

    float temperature;


    //Check if TA <= 0°C and convert to denary accordingly
    if ((upper_byte & 0x10) == 0x10) {
        upper_byte = upper_byte & 0x0F;
        temperature = 256 - (((float) upper_byte * 16) + ((float) lower_byte / 16));
    } else {
        temperature = (((float) upper_byte * 16) + ((float) lower_byte / 16));

    }
    return temperature;
}

#ifdef i2c_default
void mcp9808_set_limits() {

    //Set an upper limit of 30°C for the temperature
    uint8_t upper_temp_msb = 0x01;
    uint8_t upper_temp_lsb = 0xE0;

```

```
    //Set a lower limit of 20°C for the temperature
    uint8_t lower_temp_msb = 0x01;
    uint8_t lower_temp_lsb = 0x40;

    //Set a critical limit of 40°C for the temperature
    uint8_t crit_temp_msb = 0x02;
    uint8_t crit_temp_lsb = 0x80;

    uint8_t buf[3];
    buf[0] = REG_TEMP_UPPER;
    buf[1] = upper_temp_msb;
    buf[2] = upper_temp_lsb;
    i2c_write_blocking(i2c_default, ADDRESS, buf, 3, false);

    buf[0] = REG_TEMP_LOWER;
    buf[1] = lower_temp_msb;
    buf[2] = lower_temp_lsb;
    i2c_write_blocking(i2c_default, ADDRESS, buf, 3, false);

    buf[0] = REG_TEMP_CRIT;
    buf[1] = crit_temp_msb;
    buf[2] = crit_temp_lsb;;
    i2c_write_blocking(i2c_default, ADDRESS, buf, 3, false);
}
#endif

int main() {

    stdio_init_all();

#if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) || !defined(PICO_DEFAULT_I2C_SCL_PIN)
#warning i2c/mcp9808_i2c example requires a board with I2C pins
    puts("Default I2C pins were not defined");
#else
    printf("Hello, MCP9808! Reading raw data from registers...\n");

    // This example will use I2C0 on the default SDA and SCL pins (4, 5 on a Pico)
    i2c_init(i2c_default, 400 * 1000);
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
    gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);
    // Make the I2C pins available to picotool
    bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C));

    mcp9808_set_limits();

    uint8_t buf[2];
    uint16_t upper_byte;
    uint16_t lower_byte;

    float temperature;

    while (1) {
        // Start reading ambient temperature register for 2 bytes
        i2c_write_blocking(i2c_default, ADDRESS, &REG_TEMP_AMB, 1, true);
        i2c_read_blocking(i2c_default, ADDRESS, buf, 2, false);

        upper_byte = buf[0];
        lower_byte = buf[1];

        //isolates limit flags in upper byte
```

```
        mcp9808_check_limits(upper_byte & 0xE0);

        //clears flag bits in upper byte
        temperature = mcp9808_convert_temp(upper_byte & 0x1F, lower_byte);
        printf("Ambient temperature: %.4f°C\n", temperature);

        sleep_ms(1000);
    }
#endif
}
```

Bill of Materials

Table 46. A list of materials required for the example

Item	**Quantity**	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	https://www.raspberrypi.com/products/raspberry-pi-pico/
MCP9808 board	1	https://www.adafruit.com/product/1782
M/M Jumper wires	4	generic part

Attaching a MMA8451 3-axis digital accelerometer via I2C

This example code shows how to interface the Raspberry Pi Pico to the MMA8451 digital accelerometer sensor board.

This example reads and displays the acceleration values of the board in the 3 axis. It also allows the user to set the trade-off between the range and precision based on the values they require. Values often have an offset which can be accounted for by writing to the offset correction registers. The datasheet for the sensor can be found at https://cdn-shop.adafruit.com/datasheets/MMA8451Q-1.pdf for additional information.

Wiring information

Wiring up the device requires 4 jumpers, to connect VIN, 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 VSYS pin.

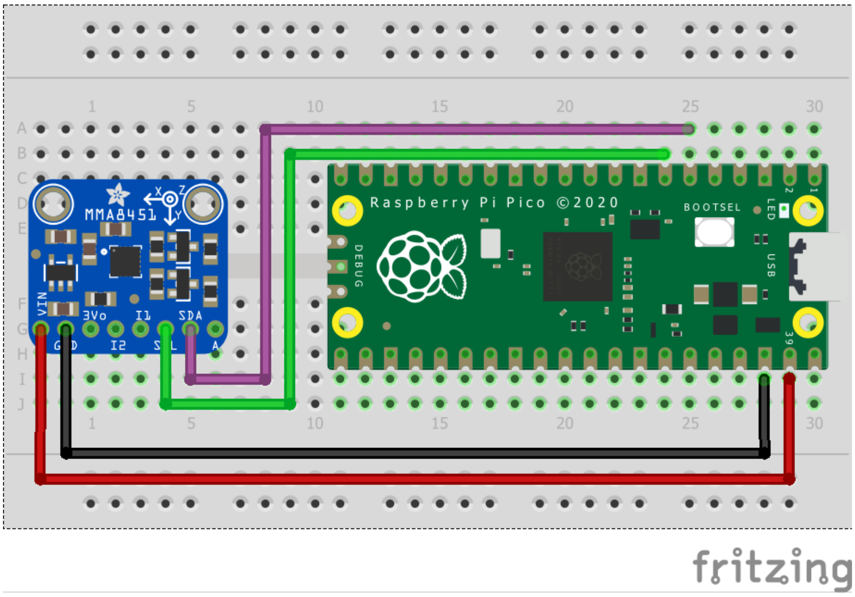


Figure 22. Wiring Diagram for MMA8451.

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/blob/master/i2c/mma8451_i2c/CMakeLists.txt

```
add_executable(mma8451_i2c
        mma8451_i2c.c
        )
# pull in common dependencies and additional i2c hardware support
target_link_libraries(mma8451_i2c pico_stdlib hardware_i2c)

# create map/bin/hex file etc.
pico_add_extra_outputs(mma8451_i2c)

# add url via pico_set_program_url
example_auto_set_url(mma8451_i2c)
```

mma8451_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/mma8451_i2c/mma8451_i2c.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 "pico/binary_info.h"
#include "hardware/i2c.h"

/* Example code to talk to a MMA8451 triple-axis accelerometer.

```

```
   This reads and writes to registers on the board.

   Connections on Raspberry Pi Pico board, other boards may vary.

   GPIO PICO_DEFAULT_I2C_SDA_PIN (On Pico this is GP4 (physical pin 6)) -> SDA on MMA8451
board
   GPIO PICO_DEFAULT_I2C_SCK_PIN (On Pico this is GP5 (physical pin 7)) -> SCL on MMA8451
board
   VSYS (physical pin 39) -> VDD on MMA8451 board
   GND (physical pin 38)  -> GND on MMA8451 board

*/

const uint8_t ADDRESS = 0x1D;

//hardware registers

const uint8_t REG_X_MSB = 0x01;
const uint8_t REG_X_LSB = 0x02;
const uint8_t REG_Y_MSB = 0x03;
const uint8_t REG_Y_LSB = 0x04;
const uint8_t REG_Z_MSB = 0x05;
const uint8_t REG_Z_LSB = 0x06;
const uint8_t REG_DATA_CFG = 0x0E;
const uint8_t REG_CTRL_REG1 = 0x2A;

// Set the range and precision for the data
const uint8_t range_config = 0x01; // 0x00 for ±2g, 0x01 for ±4g, 0x02 for ±8g
const float count = 2048; // 4096 for ±2g, 2048 for ±4g, 1024 for ±8g

uint8_t buf[2];

float mma8451_convert_accel(uint16_t raw_accel) {
    float acceleration;
    // Acceleration is read as a multiple of g (gravitational acceleration on the Earth's
surface)
    // Check if acceleration < 0 and convert to decimal accordingly
    if ((raw_accel & 0x2000) == 0x2000) {
        raw_accel &= 0x1FFF;
        acceleration = (-8192 + (float) raw_accel) / count;
    } else {
        acceleration = (float) raw_accel / count;
    }
    acceleration *= 9.81f;
    return acceleration;
}

#ifdef i2c_default
void mma8451_set_state(uint8_t state) {
    buf[0] = REG_CTRL_REG1;
    buf[1] = state; // Set RST bit to 1
    i2c_write_blocking(i2c_default, ADDRESS, buf, 2, false);
}
#endif

int main() {
    stdio_init_all();

#if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) ||
!defined(PICO_DEFAULT_I2C_SCL_PIN)
#warning i2c/mma8451_i2c example requires a board with I2C pins
    puts("Default I2C pins were not defined");
#else
    printf("Hello, MMA8451! Reading raw data from registers...\n");
```

```

    // This example will use I2C0 on the default SDA and SCL pins (4, 5 on a Pico)
    i2c_init(i2c_default, 400 * 1000);
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
    gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);
    // Make the I2C pins available to picotool
    bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN,
GPIO_FUNC_I2C));

    float x_acceleration;
    float y_acceleration;
    float z_acceleration;

    // Enable standby mode
    mma8451_set_state(0x00);

    // Edit configuration while in standby mode
    buf[0] = REG_DATA_CFG;
    buf[1] = range_config;
    i2c_write_blocking(i2c_default, ADDRESS, buf, 2, false);

    // Enable active mode
    mma8451_set_state(0x01);

    while (1) {

        // Start reading acceleration registers for 2 bytes
        i2c_write_blocking(i2c_default, ADDRESS, &REG_X_MSB, 1, true);
        i2c_read_blocking(i2c_default, ADDRESS, buf, 2, false);
        x_acceleration = mma8451_convert_accel(buf[0] << 6 | buf[1] >> 2);

        i2c_write_blocking(i2c_default, ADDRESS, &REG_Y_MSB, 1, true);
        i2c_read_blocking(i2c_default, ADDRESS, buf, 2, false);
        y_acceleration = mma8451_convert_accel(buf[0] << 6 | buf[1] >> 2);

        i2c_write_blocking(i2c_default, ADDRESS, &REG_Z_MSB, 1, true);
        i2c_read_blocking(i2c_default, ADDRESS, buf, 2, false);
        z_acceleration = mma8451_convert_accel(buf[0] << 6 | buf[1] >> 2);

        // Display acceleration values
        printf("ACCELERATION VALUES: \n");
        printf("X acceleration: %.6fms^-2\n", x_acceleration);
        printf("Y acceleration: %.6fms^-2\n", y_acceleration);
        printf("Z acceleration: %.6fms^-2\n", z_acceleration);

        sleep_ms(500);

        // Clear terminal
        printf("\033[1;1H\033[2J");
    }

#endif
}
```

Bill of Materials

Table 47. A list of materials required for the example

Item	Quantity	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	https://www.raspberrypi.com/products/raspberry-pi-pico/
MMA8451 board	1	https://www.adafruit.com/product/2019
M/M Jumper wires	4	generic part

Attaching an MPL3115A2 altimeter via I2C

This example code shows how to interface the Raspberry Pi Pico to an MPL3115A2 altimeter via I2C. The MPL3115A2 has onboard pressure and temperature sensors which are used to estimate the altitude. In comparison to the BMP-family of pressure and temperature sensors, the MPL3115A2 has two interrupt pins for ultra low power operation and takes care of the sensor reading compensation on the board! It also has multiple modes of operation and impressive operating conditions.

The board used in this example comes from Adafruit, but any MPL3115A2 breakouts should work similarly.

The MPL3115A2 makes available two ways of reading its temperature and pressure data. The first is known as polling, where the Pico will continuously read data out of a set of auto-incrementing registers which are refreshed with new data every so often. The second, which this example will demonstrate, uses a 160-byte first-in-first-out (FIFO) queue and configurable interrupts to tell the Pico when to read data. More information regarding when the interrupts can be triggered available in the datasheet. This example waits for the 32 sample FIFO to overflow, detects this via an interrupt pin, and then averages the 32 samples taken. The sensor is configured to take a sample every second.

Bit math is used to convert the temperature and altitude data from the raw bits collected in the registers. Take the temperature calculation as an example: it is a 12-bit signed number with 8 integer bits and 4 fractional bits. First, we read the 2 8-bit registers and store them in a buffer. Then, we concatenate them into one unsigned 16-bit integer starting with the OUT_T_MSB register, thus making sure that the last bit of this register is aligned with the MSB in our 16 bit unsigned integer so it is correctly interpreted as the signed bit when we later cast this to a signed 16-bit integer. Finally, the entire number is converted to a float implicitly when we multiply it by 1/2^8 to shift it 8 bits to the right of the decimal point. Though only the last 4 bits of the OUT_T_LSB register hold data, this does not matter as the remaining 4 are held at zero and "disappear" when we shift the decimal point left by 8. Similar logic is applied to the altitude calculation.

TIP

Choosing the right sensor for your project among so many choices can be hard! There are multiple factors you may have to consider in addition to any constraints imposed on you. Cost, operating temperature, sensor resolution, power consumption, ease of use, communication protocols and supply voltage are all but a few factors that can play a role in sensor choice. For most hobbyist purposes though, the majority of sensors out there will do just fine!

Wiring information

Wiring up the device requires 5 jumpers, to connect VCC (3.3v), GND, INT1, SDA and SCL. The example here uses I2C port 0, which is assigned to GPIO 4 (SDA) and GPIO 5 (SCL) by default. Power is supplied from the 3.3V pin.

> **NOTE**
>
> The MPL3115A2 has a 1.6-3.6V voltage supply range. This means it can work with the Pico's 3.3v pins out of the box but our Adafruit breakout has an onboard voltage regulator for good measure. This may not always be true of other sensors, though.

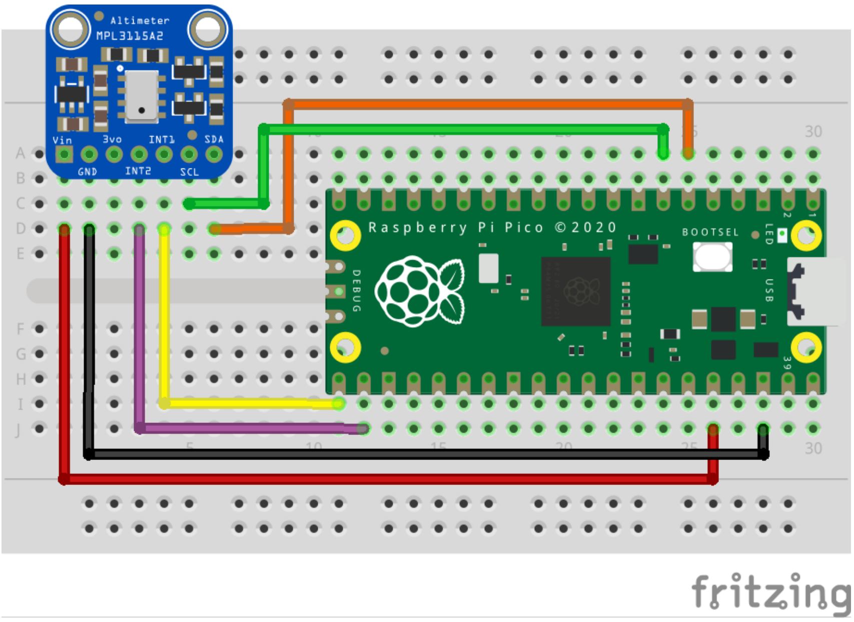


Figure 23. Wiring Diagram for MPL3115A2 altimeter.

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/blob/master/i2c/mpl3115a2_i2c/CMakeLists.txt

```
add_executable(mpl3115a2_i2c
        mpl3115a2_i2c.c
        )

# pull in common dependencies and additional i2c hardware support
target_link_libraries(mpl3115a2_i2c pico_stdlib hardware_i2c)

# create map/bin/hex file etc.
pico_add_extra_outputs(mpl3115a2_i2c)

# add url via pico_set_program_url
example_auto_set_url(mpl3115a2_i2c)
```

mpl3115a2_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/mpl3115a2_i2c/mpl3115a2_i2c.c

```
/**
 * Copyright (c) 2021 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */
```

```

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

/* Example code to talk to an MPL3115A2 altimeter sensor via I2C

   See accompanying documentation in README.adoc or the C++ SDK booklet.

   Connections on Raspberry Pi Pico board, other boards may vary.

   GPIO PICO_DEFAULT_I2C_SDA_PIN (On Pico this is 4 (pin 6)) -> SDA on MPL3115A2 board
   GPIO PICO_DEFAULT_I2C_SCK_PIN (On Pico this is 5 (pin 7)) -> SCL on MPL3115A2 board
   GPIO 16 -> INT1 on MPL3115A2 board
   3.3v (pin 36) -> VCC on MPL3115A2 board
   GND (pin 38)  -> GND on MPL3115A2 board
*/

// 7-bit address
#define ADDR 0x60
#define INT1_PIN _u(16)

// following definitions only valid for F_MODE > 0 (ie. if FIFO enabled)
#define MPL3115A2_F_DATA _u(0x01)
#define MPL3115A2_F_STATUS _u(0x00)
#define MPL3115A2_F_SETUP _u(0x0F)
#define MPL3115A2_INT_SOURCE _u(0x12)
#define MPL3115A2_CTRLREG1 _u(0x26)
#define MPL3115A2_CTRLREG2 _u(0x27)
#define MPL3115A2_CTRLREG3 _u(0x28)
#define MPL3115A2_CTRLREG4 _u(0x29)
#define MPL3115A2_CTRLREG5 _u(0x2A)
#define MPL3115A2_PT_DATA_CFG _u(0x13)
#define MPL3115A2_OFF_P _u(0x2B)
#define MPL3115A2_OFF_T _u(0x2C)
#define MPL3115A2_OFF_H _u(0x2D)

#define MPL3115A2_FIFO_DISABLED _u(0x00)
#define MPL3115A2_FIFO_STOP_ON_OVERFLOW _u(0x80)
#define MPL3115A2_FIFO_SIZE 32
#define MPL3115A2_DATA_BATCH_SIZE 5
#define MPL3115A2_ALTITUDE_NUM_REGS 3
#define MPL3115A2_ALTITUDE_INT_SIZE 20
#define MPL3115A2_TEMPERATURE_INT_SIZE 12
#define MPL3115A2_NUM_FRAC_BITS 4

#define PARAM_ASSERTIONS_ENABLE_I2C 1

volatile uint8_t fifo_data[MPL3115A2_FIFO_SIZE * MPL3115A2_DATA_BATCH_SIZE];
volatile bool has_new_data = false;

struct mpl3115a2_data_t {
    // Q8.4 fixed point
    float temperature;
    // Q16.4 fixed-point
    float altitude;
};

void copy_to_vbuf(uint8_t buf1[], volatile uint8_t buf2[], uint buflen) {
    for (size_t i = 0; i < buflen; i++) {
        buf2[i] = buf1[i];
    }
```

```
}

#ifdef i2c_default

void mpl3115a2_read_fifo(volatile uint8_t fifo_buf[]) {
    // drains the 160 byte FIFO
    uint8_t reg = MPL3115A2_F_DATA;
    uint8_t buf[MPL3115A2_FIFO_SIZE * MPL3115A2_DATA_BATCH_SIZE];
    i2c_write_blocking(i2c_default, ADDR, &reg, 1, true);
    // burst read 160 bytes from fifo
    i2c_read_blocking(i2c_default, ADDR, buf, MPL3115A2_FIFO_SIZE *
MPL3115A2_DATA_BATCH_SIZE, false);
    copy_to_vbuf(buf, fifo_buf, MPL3115A2_FIFO_SIZE * MPL3115A2_DATA_BATCH_SIZE);
}

uint8_t mpl3115a2_read_reg(uint8_t reg) {
    uint8_t read;
    i2c_write_blocking(i2c_default, ADDR, &reg, 1, true); // keep control of bus
    i2c_read_blocking(i2c_default, ADDR, &read, 1, false);
    return read;
}

void mpl3115a2_init() {
    // set as altimeter with oversampling ratio of 128
    uint8_t buf[] = {MPL3115A2_CTRLREG1, 0xB8};
    i2c_write_blocking(i2c_default, ADDR, buf, 2, false);

    // set data refresh every 2 seconds, 0 next bits as we're not using those interrupts
    buf[0] = MPL3115A2_CTRLREG2, buf[1] = 0x00;
    i2c_write_blocking(i2c_default, ADDR, buf, 2, false);

    // set both interrupts pins to active low and enable internal pullups
    buf[0] = MPL3115A2_CTRLREG3, buf[1] = 0x01;
    i2c_write_blocking(i2c_default, ADDR, buf, 2, false);

    // enable FIFO interrupt
    buf[0] = MPL3115A2_CTRLREG4, buf[1] = 0x40;
    i2c_write_blocking(i2c_default, ADDR, buf, 2, false);

    // tie FIFO interrupt to pin INT1
    buf[0] = MPL3115A2_CTRLREG5, buf[1] = 0x40;
    i2c_write_blocking(i2c_default, ADDR, buf, 2, false);

    // set p, t and h offsets here if needed
    // eg. 2's complement number: 0xFF subtracts 1 meter
    //buf[0] = MPL3115A2_OFF_H, buf[1] = 0xFF;
    //i2c_write_blocking(i2c_default, ADDR, buf, 2, false);

    // do not accept more data on FIFO overflow
    buf[0] = MPL3115A2_F_SETUP, buf[1] = MPL3115A2_FIFO_STOP_ON_OVERFLOW;
    i2c_write_blocking(i2c_default, ADDR, buf, 2, false);

    // set device active
    buf[0] = MPL3115A2_CTRLREG1, buf[1] = 0xB9;
    i2c_write_blocking(i2c_default, ADDR, buf, 2, false);
}

void gpio_callback(uint gpio, __unused uint32_t events) {
    // if we had enabled more than 2 interrupts on same pin, then we should read
    // INT_SOURCE reg to find out which interrupt triggered

    // we can filter by which GPIO was triggered
    if (gpio == INT1_PIN) {
        // FIFO overflow interrupt
```

```
        // watermark bits set to 0 in F_SETUP reg, so only possible event is an overflow
        // otherwise, we would read F_STATUS to confirm it was an overflow
        printf("FIFO overflow!\n");
        // drain the fifo
        mpl3115a2_read_fifo(fifo_data);
        // read status register to clear interrupt bit
        mpl3115a2_read_reg(MPL3115A2_F_STATUS);
        has_new_data = true;
    }
}

#endif

void mpl3115a2_convert_fifo_batch(uint8_t start, volatile uint8_t buf[], struct mpl3115a2_data_t *data) {
    // convert a batch of fifo data into temperature and altitude data

    // 3 altitude registers: MSB (8 bits), CSB (8 bits) and LSB (4 bits, starting from MSB)
    // first two are integer bits (2's complement) and LSB is fractional bits -> makes 20 bit signed integer
    int32_t h = (int32_t) buf[start] << 24;
    h |= (int32_t) buf[start + 1] << 16;
    h |= (int32_t) buf[start + 2] << 8;
    data->altitude = ((float)h) / 65536.f;

    // 2 temperature registers: MSB (8 bits) and LSB (4 bits, starting from MSB)
    // first 8 are integer bits with sign and LSB is fractional bits -> 12 bit signed integer
    int16_t t = (int16_t) buf[start + 3] << 8;
    t |= (int16_t) buf[start + 4];
    data->temperature = ((float)t) / 256.f;
}

int main() {
    stdio_init_all();
#if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) || !defined(PICO_DEFAULT_I2C_SCL_PIN)
#warning i2c / mpl3115a2_i2c example requires a board with I2C pins
    puts("Default I2C pins were not defined");
    return 0;
#else
    printf("Hello, MPL3115A2. Waiting for something to interrupt me!...\n");

    // use default I2C0 at 400kHz, I2C is active low
    i2c_init(i2c_default, 400 * 1000);
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
    gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);

    gpio_init(INT1_PIN);
    gpio_pull_up(INT1_PIN); // pull it up even more!

    // add program information for picotool
    bi_decl(bi_program_name("Example in the pico-examples library for the MPL3115A2 altimeter"));
    bi_decl(bi_1pin_with_name(16, "Interrupt pin 1"));
    bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C));

    mpl3115a2_init();

    gpio_set_irq_enabled_with_callback(INT1_PIN, GPIO_IRQ_LEVEL_LOW, true, &gpio_callback);

    while (1) {
```

```
        // as interrupt data comes in, let's print the 32 sample average
        if (has_new_data) {
            float tsum = 0, hsum = 0;
            struct mpl3115a2_data_t data;
            for (int i = 0; i < MPL3115A2_FIFO_SIZE; i++) {
                mpl3115a2_convert_fifo_batch(i * MPL3115A2_DATA_BATCH_SIZE, fifo_data, &
data);
                tsum += data.temperature;
                hsum += data.altitude;
            }
            printf("%d sample average -> t: %.4f C, h: %.4f m\n", MPL3115A2_FIFO_SIZE, tsum
/ MPL3115A2_FIFO_SIZE,
                   hsum / MPL3115A2_FIFO_SIZE);
            has_new_data = false;
        }
        sleep_ms(10);
    };

#endif
}
```

Bill of Materials

Table 48. A list of materials required for the example

Item	Quantity	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	https://www.raspberrypi.com/products/raspberry-pi-pico/
MPL3115A2 altimeter	1	Adafruit
M/M Jumper wires	5	generic part

Attaching an OLED display via I2C

This example code shows how to interface the Raspberry Pi Pico with an 128x32 OLED display board based on the SSD1306 display driver, datasheet here.

The code displays a series of small demo graphics; tiny raspberries that scroll horizontally, some text, and some line drawing, in the process showing you how to initialize the display, write to the entire display, write to only a portion of the display, configure scrolling, invert the display etc.

The SSD1306 is operated via a list of versatile commands (see datasheet) that allows the user to access all the capabilities of the driver. After sending a slave address, the data that follows can be either a command, flags to follow up a command or data to be written directly into the display's RAM. A control byte is required for each write after the slave address so that the driver knows what type of data is being sent.

The example code supports displays of 32 pixel or 64 pixels high by 128 pixels wide by changing a define at the top of the code.

In the 32 vertical pixels case, the display is partitioned into 4 pages, each 8 pixels in height. In RAM, this looks roughly like:

```
           | COL0 | COL1 | COL2 | COL3 |  ...  | COL126 | COL127 |
    PAGE 0 |      |      |      |      |       |        |        |
    PAGE 1 |      |      |      |      |       |        |        |
```

```
PAGE 2 |       |       |       |       |        |         |         |
PAGE 3 |       |       |       |       |        |         |         |
---------------------------------------------------------------
```

Within each page, we have:

```
      | COL0  | COL1  | COL2  | COL3  |  ...   | COL126  | COL127  |
COM 0 |       |       |       |       |        |         |         |
COM 1 |       |       |       |       |        |         |         |
   :  |       |       |       |       |        |         |         |
COM 7 |       |       |       |       |        |         |         |
--------------------------------------------------------------
```

NOTE

There is a difference between columns in RAM and the actual segment pads that connect the driver to the display. The RAM addresses COL0 - COL127 are mapped to these segment pins SEG0 - SEG127 by default. The distinction between these two is important as we can for example, easily mirror contents of RAM without rewriting a buffer.

The driver has 3 modes of transferring the pixels in RAM to the display (provided that the driver is set to use its RAM content to drive the display, ie. command 0xA4 is sent). We choose horizontal addressing mode which, after setting the column address and page address registers to our desired start positions, will increment the column address register until the OLED display width is reached (127 in our case) after which the column address register will reset to its starting value and the page address is incremented. Once the page register reaches the end, it will wrap around as well. Effectively, this scans across the display from top to bottom, left to right in blocks that are 8 pixels high. When a byte is sent to be written into RAM, it sets all the rows for the current position of the column address register. So, if we send 10101010, and we are on PAGE 0 and COL1, COM0 is set to 1, COM1 is set to 0, COM2 is set to 1, and so on. Effectively, the byte is "transposed" to fill a single page’s column. The datasheet has further information on this and the two other modes.

Horizontal addressing mode has the key advantage that we can keep one single 512 byte buffer (128 columns x 4 pages and each byte fills a page’s rows) and write this in one go to the RAM (column address auto increments on writes as well as reads) instead of working with 2D matrices of pixels and adding more overhead.

Wiring information

Wiring up the device requires 4 jumpers, to connect VCC (3.3v), GND, SDA and SCL and optionally a 5th jumper for the driver RESET pin. The example here uses the default I2C port 0, which is assigned to GPIO 4 (SDA) and 5 (SCL) in software. Power is supplied from the 3.3V pin from the Pico.

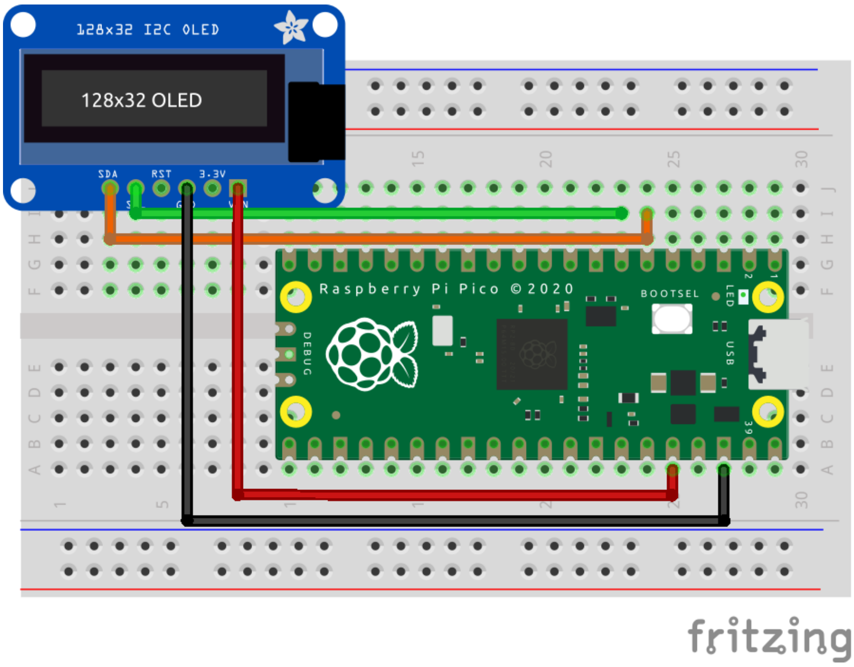


Figure 24. Wiring Diagram for oled display via I2C.

List of Files

CMakeLists.txt

CMake file to incorporate the example into the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/ssd1306_i2c/CMakeLists.txt

```
add_executable(ssd1306_i2c
        ssd1306_i2c.c
        )

# pull in common dependencies and additional i2c hardware support
target_link_libraries(ssd1306_i2c pico_stdlib hardware_i2c)

# create map/bin/hex file etc.
pico_add_extra_outputs(ssd1306_i2c)

# add url via pico_set_program_url
example_auto_set_url(ssd1306_i2c)
```

ssd1306_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/ssd1306_i2c/ssd1306_i2c.c

```
/**
 * Copyright (c) 2021 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */

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

```c
#include "pico/stdlib.h"
#include "pico/binary_info.h"
#include "hardware/i2c.h"
#include "raspberry26x32.h"
#include "ssd1306_font.h"

/* Example code to talk to an SSD1306-based OLED display

   The SSD1306 is an OLED/PLED driver chip, capable of driving displays up to
   128x64 pixels.

   NOTE: Ensure the device is capable of being driven at 3.3v NOT 5v. The Pico
   GPIO (and therefore 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 PICO_DEFAULT_I2C_SDA_PIN (on Pico this is GP4 (pin 6)) -> SDA on display
   board
   GPIO PICO_DEFAULT_I2C_SCL_PIN (on Pico this is GP5 (pin 7)) -> SCL on
   display board
   3.3v (pin 36) -> VCC on display board
   GND (pin 38)  -> GND on display board
*/

// Define the size of the display we have attached. This can vary, make sure you
// have the right size defined or the output will look rather odd!
// Code has been tested on 128x32 and 128x64 OLED displays
#define SSD1306_HEIGHT              32
#define SSD1306_WIDTH               128

#define SSD1306_I2C_ADDR            _u(0x3C)

// 400 is usual, but often these can be overclocked to improve display response.
// Tested at 1000 on both 32 and 84 pixel height devices and it worked.
#define SSD1306_I2C_CLK             400
//#define SSD1306_I2C_CLK             1000


// commands (see datasheet)
#define SSD1306_SET_MEM_MODE        _u(0x20)
#define SSD1306_SET_COL_ADDR        _u(0x21)
#define SSD1306_SET_PAGE_ADDR       _u(0x22)
#define SSD1306_SET_HORIZ_SCROLL    _u(0x26)
#define SSD1306_SET_SCROLL          _u(0x2E)

#define SSD1306_SET_DISP_START_LINE _u(0x40)

#define SSD1306_SET_CONTRAST        _u(0x81)
#define SSD1306_SET_CHARGE_PUMP     _u(0x8D)

#define SSD1306_SET_SEG_REMAP       _u(0xA0)
#define SSD1306_SET_ENTIRE_ON       _u(0xA4)
#define SSD1306_SET_ALL_ON          _u(0xA5)
#define SSD1306_SET_NORM_DISP       _u(0xA6)
#define SSD1306_SET_INV_DISP        _u(0xA7)
#define SSD1306_SET_MUX_RATIO       _u(0xA8)
#define SSD1306_SET_DISP            _u(0xAE)
#define SSD1306_SET_COM_OUT_DIR     _u(0xC0)
#define SSD1306_SET_COM_OUT_DIR_FLIP _u(0xC0)

#define SSD1306_SET_DISP_OFFSET     _u(0xD3)
```

```
#define SSD1306_SET_DISP_CLK_DIV    _u(0xD5)
#define SSD1306_SET_PRECHARGE       _u(0xD9)
#define SSD1306_SET_COM_PIN_CFG     _u(0xDA)
#define SSD1306_SET_VCOM_DESEL      _u(0xDB)

#define SSD1306_PAGE_HEIGHT         _u(8)
#define SSD1306_NUM_PAGES           (SSD1306_HEIGHT / SSD1306_PAGE_HEIGHT)
#define SSD1306_BUF_LEN             (SSD1306_NUM_PAGES * SSD1306_WIDTH)

#define SSD1306_WRITE_MODE         _u(0xFE)
#define SSD1306_READ_MODE          _u(0xFF)


struct render_area {
    uint8_t start_col;
    uint8_t end_col;
    uint8_t start_page;
    uint8_t end_page;

    int buflen;
};

void calc_render_area_buflen(struct render_area *area) {
    // calculate how long the flattened buffer will be for a render area
    area->buflen = (area->end_col - area->start_col + 1) * (area->end_page - area->start_page + 1);
}

#ifdef i2c_default

void SSD1306_send_cmd(uint8_t cmd) {
    // I2C write process expects a control byte followed by data
    // this "data" can be a command or data to follow up a command
    // Co = 1, D/C = 0 => the driver expects a command
    uint8_t buf[2] = {0x80, cmd};
    i2c_write_blocking(i2c_default, SSD1306_I2C_ADDR, buf, 2, false);
}

void SSD1306_send_cmd_list(uint8_t *buf, int num) {
    for (int i=0;i<num;i++)
        SSD1306_send_cmd(buf[i]);
}

void SSD1306_send_buf(uint8_t buf[], int buflen) {
    // in horizontal addressing mode, the column address pointer auto-increments
    // and then wraps around to the next page, so we can send the entire frame
    // buffer in one gooooooo!

    // copy our frame buffer into a new buffer because we need to add the control byte
    // to the beginning

    uint8_t *temp_buf = malloc(buflen + 1);

    temp_buf[0] = 0x40;
    memcpy(temp_buf+1, buf, buflen);

    i2c_write_blocking(i2c_default, SSD1306_I2C_ADDR, temp_buf, buflen + 1, false);

    free(temp_buf);
}

void SSD1306_init() {
    // Some of these commands are not strictly necessary as the reset
    // process defaults to some of these but they are shown here
```

```c
    // to demonstrate what the initialization sequence looks like
    // Some configuration values are recommended by the board manufacturer

    uint8_t cmds[] = {
        SSD1306_SET_DISP,               // set display off
        /* memory mapping */
        SSD1306_SET_MEM_MODE,           // set memory address mode 0 = horizontal, 1 = vertical, 2 = page
        0x00,                           // horizontal addressing mode
        /* resolution and layout */
        SSD1306_SET_DISP_START_LINE,    // set display start line to 0
        SSD1306_SET_SEG_REMAP | 0x01,   // set segment re-map, column address 127 is mapped to SEG0
        SSD1306_SET_MUX_RATIO,          // set multiplex ratio
        SSD1306_HEIGHT - 1,             // Display height - 1
        SSD1306_SET_COM_OUT_DIR | 0x08, // set COM (common) output scan direction. Scan from bottom up, COM[N-1] to COM0
        SSD1306_SET_DISP_OFFSET,        // set display offset
        0x00,                           // no offset
        SSD1306_SET_COM_PIN_CFG,        // set COM (common) pins hardware configuration. Board specific magic number.
                                        // 0x02 Works for 128x32, 0x12 Possibly works for 128x64. Other options 0x22, 0x32
#if ((SSD1306_WIDTH == 128) && (SSD1306_HEIGHT == 32))
        0x02,
#elif ((SSD1306_WIDTH == 128) && (SSD1306_HEIGHT == 64))
        0x12,
#else
        0x02,
#endif
        /* timing and driving scheme */
        SSD1306_SET_DISP_CLK_DIV,       // set display clock divide ratio
        0x80,                           // div ratio of 1, standard freq
        SSD1306_SET_PRECHARGE,          // set pre-charge period
        0xF1,                           // Vcc internally generated on our board
        SSD1306_SET_VCOM_DESEL,         // set VCOMH deselect level
        0x30,                           // 0.83xVcc
        /* display */
        SSD1306_SET_CONTRAST,           // set contrast control
        0xFF,
        SSD1306_SET_ENTIRE_ON,          // set entire display on to follow RAM content
        SSD1306_SET_NORM_DISP,           // set normal (not inverted) display
        SSD1306_SET_CHARGE_PUMP,        // set charge pump
        0x14,                           // Vcc internally generated on our board
        SSD1306_SET_SCROLL | 0x00,      // deactivate horizontal scrolling if set. This is necessary as memory writes will corrupt if scrolling was enabled
        SSD1306_SET_DISP | 0x01, // turn display on
    };

    SSD1306_send_cmd_list(cmds, count_of(cmds));
}

void SSD1306_scroll(bool on) {
    // configure horizontal scrolling
    uint8_t cmds[] = {
        SSD1306_SET_HORIZ_SCROLL | 0x00,
        0x00, // dummy byte
        0x00, // start page 0
        0x00, // time interval
        0x03, // end page 3 SSD1306_NUM_PAGES ??
        0x00, // dummy byte
        0xFF, // dummy byte
        SSD1306_SET_SCROLL | (on ? 0x01 : 0) // Start/stop scrolling
    };
```

```

    SSD1306_send_cmd_list(cmds, count_of(cmds));
}

void render(uint8_t *buf, struct render_area *area) {
    // update a portion of the display with a render area
    uint8_t cmds[] = {
        SSD1306_SET_COL_ADDR,
        area->start_col,
        area->end_col,
        SSD1306_SET_PAGE_ADDR,
        area->start_page,
        area->end_page
    };

    SSD1306_send_cmd_list(cmds, count_of(cmds));
    SSD1306_send_buf(buf, area->buflen);
}

static void SetPixel(uint8_t *buf, int x,int y, bool on) {
    assert(x >= 0 && x < SSD1306_WIDTH && y >=0 && y < SSD1306_HEIGHT);

    // The calculation to determine the correct bit to set depends on which address
    // mode we are in. This code assumes horizontal

    // The video ram on the SSD1306 is split up in to 8 rows, one bit per pixel.
    // Each row is 128 long by 8 pixels high, each byte vertically arranged, so byte 0 is x=0, y=0->7,
    // byte 1 is x = 1, y=0->7 etc

    // This code could be optimised, but is like this for clarity. The compiler
    // should do a half decent job optimising it anyway.

    const int BytesPerRow = SSD1306_WIDTH ; // x pixels, 1bpp, but each row is 8 pixel high, so (x / 8) * 8

    int byte_idx = (y / 8) * BytesPerRow + x;
    uint8_t byte = buf[byte_idx];

    if (on)
        byte |=  1 << (y % 8);
    else
        byte &= ~(1 << (y % 8));

    buf[byte_idx] = byte;
}
// Basic Bresenhams.
static void DrawLine(uint8_t *buf, int x0, int y0, int x1, int y1, bool on) {

    int dx =  abs(x1-x0);
    int sx = x0<x1 ? 1 : -1;
    int dy = -abs(y1-y0);
    int sy = y0<y1 ? 1 : -1;
    int err = dx+dy;
    int e2;

    while (true) {
        SetPixel(buf, x0, y0, on);
        if (x0 == x1 && y0 == y1)
            break;
        e2 = 2*err;

        if (e2 >= dy) {
            err += dy;
```

```
            x0 += sx;
        }
        if (e2 <= dx) {
            err += dx;
            y0 += sy;
        }
    }
}

static inline int GetFontIndex(uint8_t ch) {
    if (ch >= 'A' && ch <='Z') {
        return  ch - 'A' + 1;
    }
    else if (ch >= '0' && ch <='9') {
        return  ch - '0' + 27;
    }
    else return  0; // Not got that char so space.
}

static void WriteChar(uint8_t *buf, int16_t x, int16_t y, uint8_t ch) {
    if (x > SSD1306_WIDTH - 8 || y > SSD1306_HEIGHT - 8)
        return;

    // For the moment, only write on Y row boundaries (every 8 vertical pixels)
    y = y/8;

    ch = toupper(ch);
    int idx = GetFontIndex(ch);
    int fb_idx = y * 128 + x;

    for (int i=0;i<8;i++) {
        buf[fb_idx++] = font[idx * 8 + i];
    }
}

static void WriteString(uint8_t *buf, int16_t x, int16_t y, char *str) {
    // Cull out any string off the screen
    if (x > SSD1306_WIDTH - 8 || y > SSD1306_HEIGHT - 8)
        return;

    while (*str) {
        WriteChar(buf, x, y, *str++);
        x+=8;
    }
}



#endif

int main() {
    stdio_init_all();

#if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) ||
!defined(PICO_DEFAULT_I2C_SCL_PIN)
#warning i2c / SSD1306_i2d example requires a board with I2C pins
    puts("Default I2C pins were not defined");
#else
    // useful information for picotool
    bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN,
GPIO_FUNC_I2C));
    bi_decl(bi_program_description("SSD1306 OLED driver I2C example for the Raspberry Pi
Pico"));

```

```
    printf("Hello, SSD1306 OLED display! Look at my raspberries..\n");

    // I2C is "open drain", pull ups to keep signal high when no data is being
    // sent
    i2c_init(i2c_default, SSD1306_I2C_CLK * 1000);
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
    gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);

    // run through the complete initialization process
    SSD1306_init();

    // Initialize render area for entire frame (SSD1306_WIDTH pixels by SSD1306_NUM_PAGES pages)
    struct render_area frame_area = {
        start_col: 0,
        end_col : SSD1306_WIDTH - 1,
        start_page : 0,
        end_page : SSD1306_NUM_PAGES - 1
        };

    calc_render_area_buflen(&frame_area);

    // zero the entire display
    uint8_t buf[SSD1306_BUF_LEN];
    memset(buf, 0, SSD1306_BUF_LEN);
    render(buf, &frame_area);

    // intro sequence: flash the screen 3 times
    for (int i = 0; i < 3; i++) {
        SSD1306_send_cmd(SSD1306_SET_ALL_ON);    // Set all pixels on
        sleep_ms(500);
        SSD1306_send_cmd(SSD1306_SET_ENTIRE_ON); // go back to following RAM for pixel state
        sleep_ms(500);
    }

    // render 3 cute little raspberries
    struct render_area area = {
        start_page : 0,
        end_page : (IMG_HEIGHT / SSD1306_PAGE_HEIGHT)  - 1
    };

restart:

    area.start_col = 0;
    area.end_col = IMG_WIDTH - 1;

    calc_render_area_buflen(&area);

    uint8_t offset = 5 + IMG_WIDTH; // 5px padding

    for (int i = 0; i < 3; i++) {
        render(raspberry26x32, &area);
        area.start_col += offset;
        area.end_col += offset;
    }

    SSD1306_scroll(true);
    sleep_ms(5000);
    SSD1306_scroll(false);

    char *text[] = {
        "A long time ago",
```

```
        "  on an OLED ",
        "   display",
        " far far away",
        "Lived a small",
        "red raspberry",
        "by the name of",
        "    PICO"
    };

    int y = 0;
    for (uint i = 0 ;i < count_of(text); i++) {
        WriteString(buf, 5, y, text[i]);
        y+=8;
    }
    render(buf, &frame_area);

    // Test the display invert function
    sleep_ms(3000);
    SSD1306_send_cmd(SSD1306_SET_INV_DISP);
    sleep_ms(3000);
    SSD1306_send_cmd(SSD1306_SET_NORM_DISP);

    bool pix = true;
    for (int i = 0; i < 2;i++) {
        for (int x = 0;x < SSD1306_WIDTH;x++) {
            DrawLine(buf, x, 0,  SSD1306_WIDTH - 1 - x, SSD1306_HEIGHT - 1, pix);
            render(buf, &frame_area);
        }

        for (int y = SSD1306_HEIGHT-1; y >= 0 ;y--) {
            DrawLine(buf, 0, y, SSD1306_WIDTH - 1, SSD1306_HEIGHT - 1 - y, pix);
            render(buf, &frame_area);
        }
        pix = false;
    }

    goto restart;

#endif
    return 0;
}
```

ssd1306_font.h

A simple font used in the example.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/ssd1306_i2c/ssd1306_font.h

```
/**
 * Copyright (c) 2022 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */

// Vertical bitmaps, A-Z, 0-9. Each is 8 pixels high and wide
// These are defined vertically to make them quick to copy to FB

static uint8_t font[] = {
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // Nothing
0x78, 0x14, 0x12, 0x11, 0x12, 0x14, 0x78, 0x00, //A
0x7f, 0x49, 0x49, 0x49, 0x49, 0x49, 0x7f, 0x00, //B
0x7e, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x00, //C
0x7f, 0x41, 0x41, 0x41, 0x41, 0x41, 0x7e, 0x00, //D
```

```
0x7f, 0x49, 0x49, 0x49, 0x49, 0x49, 0x49, 0x00, //E
0x7f, 0x09, 0x09, 0x09, 0x09, 0x01, 0x01, 0x00, //F
0x7f, 0x41, 0x41, 0x41, 0x51, 0x51, 0x73, 0x00, //G
0x7f, 0x08, 0x08, 0x08, 0x08, 0x08, 0x7f, 0x00, //H
0x00, 0x00, 0x00, 0x7f, 0x00, 0x00, 0x00, 0x00, //I
0x21, 0x41, 0x41, 0x3f, 0x01, 0x01, 0x01, 0x00, //J
0x00, 0x7f, 0x08, 0x08, 0x14, 0x22, 0x41, 0x00, //K
0x7f, 0x40, 0x40, 0x40, 0x40, 0x40, 0x40, 0x00, //L
0x7f, 0x02, 0x04, 0x08, 0x04, 0x02, 0x7f, 0x00, //M
0x7f, 0x02, 0x04, 0x08, 0x10, 0x20, 0x7f, 0x00, //N
0x3e, 0x41, 0x41, 0x41, 0x41, 0x41, 0x3e, 0x00, //O
0x7f, 0x11, 0x11, 0x11, 0x11, 0x11, 0x0e, 0x00, //P
0x3e, 0x41, 0x41, 0x49, 0x51, 0x61, 0x7e, 0x00, //Q
0x7f, 0x11, 0x11, 0x11, 0x31, 0x51, 0x0e, 0x00, //R
0x46, 0x49, 0x49, 0x49, 0x49, 0x30, 0x00, 0x00, //S
0x01, 0x01, 0x01, 0x7f, 0x01, 0x01, 0x01, 0x00, //T
0x3f, 0x40, 0x40, 0x40, 0x40, 0x40, 0x3f, 0x00, //U
0x0f, 0x10, 0x20, 0x40, 0x20, 0x10, 0x0f, 0x00, //V
0x7f, 0x20, 0x10, 0x08, 0x10, 0x20, 0x7f, 0x00, //W
0x00, 0x41, 0x22, 0x14, 0x14, 0x22, 0x41, 0x00, //X
0x01, 0x02, 0x04, 0x78, 0x04, 0x02, 0x01, 0x00, //Y
0x41, 0x61, 0x59, 0x45, 0x43, 0x41, 0x00, 0x00, //Z
0x3e, 0x41, 0x41, 0x49, 0x41, 0x41, 0x3e, 0x00, //0
0x00, 0x00, 0x42, 0x7f, 0x40, 0x00, 0x00, 0x00, //1
0x30, 0x49, 0x49, 0x49, 0x49, 0x46, 0x00, 0x00, //2
0x49, 0x49, 0x49, 0x49, 0x49, 0x49, 0x36, 0x00, //3
0x3f, 0x20, 0x20, 0x78, 0x20, 0x20, 0x00, 0x00, //4
0x4f, 0x49, 0x49, 0x49, 0x49, 0x30, 0x00, 0x00, //5
0x3f, 0x48, 0x48, 0x48, 0x48, 0x48, 0x30, 0x00, //6
0x01, 0x01, 0x01, 0x61, 0x31, 0x0d, 0x03, 0x00, //7
0x36, 0x49, 0x49, 0x49, 0x49, 0x49, 0x36, 0x00, //8
0x06, 0x09, 0x09, 0x09, 0x09, 0x09, 0x7f, 0x00, //9
};
```

img_to_array.py

A helper to convert an image file to an array that can be used in the example.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/ssd1306_i2c/img_to_array.py

```python
#!/usr/bin/env python3

# Converts a grayscale image into a format able to be
# displayed by the SSD1306 driver in horizontal addressing mode

# usage: python3 img_to_array.py <logo.bmp>

# depends on the Pillow library
# `python3 -m pip install --upgrade Pillow`

from PIL import Image
import sys
from pathlib import Path

OLED_HEIGHT = 32
OLED_WIDTH = 128
OLED_PAGE_HEIGHT = 8

if len(sys.argv) < 2:
    print("No image path provided.")
    sys.exit()

img_path = sys.argv[1]
```

```

try:
    im = Image.open(img_path)
except OSError:
    raise Exception("Oops! The image could not be opened.")

img_width = im.size[0]
img_height = im.size[1]

if img_width > OLED_WIDTH or img_height > OLED_HEIGHT:
    print(f'Your image is f{img_width} pixels wide and {img_height} pixels high, but...')
    raise Exception(f"OLED display only {OLED_WIDTH} pixels wide and {OLED_HEIGHT} pixels
high!")

if not (im.mode == "1" or im.mode == "L"):
    raise Exception("Image must be grayscale only")

# black or white
out = im.convert("1")

img_name = Path(im.filename).stem

# `pixels` is a flattened array with the top left pixel at index 0
# and bottom right pixel at the width*height-1
pixels = list(out.getdata())

# swap white for black and swap (255, 0) for (1, 0)
pixels = [0 if x == 255 else 1 for x in pixels]

# our goal is to divide the image into 8-pixel high pages
# and turn a pixel column into one byte, eg for one page:
# 0 1 0 ....
# 1 0 0
# 1 1 1
# 0 0 1
# 1 1 0
# 0 1 0
# 1 1 1
# 0 0 1 ....

# we get 0x6A, 0xAE, 0x33 ... and so on
# as `pixels` is flattened, each bit in a column is IMG_WIDTH apart from the next

buffer = []
for i in range(img_height // OLED_PAGE_HEIGHT):
    start_index = i*img_width*OLED_PAGE_HEIGHT
    for j in range(img_width):
        out_byte = 0
        for k in range(OLED_PAGE_HEIGHT):
            out_byte |= pixels[k*img_width + start_index + j] << k
        buffer.append(f'{out_byte:#04x}')

buffer = ", ".join(buffer)
buffer_hex = f'static uint8_t {img_name}[] = {{{buffer}}};\n'

with open(f'{img_name}.h', 'wt') as file:
    file.write(f'#define IMG_WIDTH {img_width}\n')
    file.write(f'#define IMG_HEIGHT {img_height}\n\n')
    file.write(buffer_hex)
```

raspberry26x32.bmp

Example image file of a Raspberry.

raspberry26x32.h

The example image file converted to an C array.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/ssd1306_i2c/raspberry26x32.h

```
#define IMG_WIDTH 26
#define IMG_HEIGHT 32

static uint8_t raspberry26x32[] = { 0x0, 0x0, 0xe, 0x7e, 0xfe, 0xff, 0xff, 0xff, 0xff, 0xff,
0xfe, 0xfe, 0xfc, 0xf8, 0xfc, 0xfe, 0xfe, 0xff, 0xff, 0xff, 0xff, 0xff, 0xfe, 0x7e, 0x1e, 0x0,
0x0, 0x0, 0x80, 0xe0, 0xf8, 0xfd, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xfd, 0xf8, 0xe0, 0x80, 0x0, 0x0, 0x1e, 0x7f, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0x7f, 0x1e, 0x0, 0x0, 0x0, 0x3, 0x7, 0xf, 0x1f, 0x1f, 0x3f, 0x3f, 0x7f,
0xff, 0xff, 0xff, 0xff, 0x7f, 0x7f, 0x3f, 0x3f, 0x1f, 0x1f, 0xf, 0x7, 0x3, 0x0, 0x0};
```

Bill of Materials

Table 49. A list of materials required for the example

Item	Quantity	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	https://www.raspberrypi.com/products/raspberry-pi-pico/
SSD1306-based OLED display	1	Adafruit part
M/M Jumper wires	4	generic part

Attaching a PA1010D Mini GPS module via I2C

This example code shows how to interface the Raspberry Pi Pico to the PA1010D Mini GPS module

This allows you to read basic location and time data from the Recommended Minimum Specific GNSS Sentence (GNRMC protocol) and displays it in a user-friendly format. The datasheet for the module can be found on https://cdn-learn.adafruit.com/assets/assets/000/084/295/original/CD_PA1010D_Datasheet_v.03.pdf?1573833002. The output sentence is read and parsed to split the data fields into a 2D character array, which are then individually printed out. The commands to use different protocols and change settings are found on https://www.sparkfun.com/datasheets/GPS/Modules/PMTK_Protocol.pdf. Additional protocols can be used by editing the `init_command` array.

NOTE

Each command requires a checksum after the asterisk. The checksum can be calculated for your command using the following website: https://nmeachecksum.eqth.net/.

The GPS needs to be used outdoors in open skies and requires about 15 seconds to acquire a satellite signal in order to display valid data. When the signal is detected, the device will blink a green LED at 1 Hz.

Wiring information

Wiring up the device requires 4 jumpers, to connect VDD, 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 3V pin.

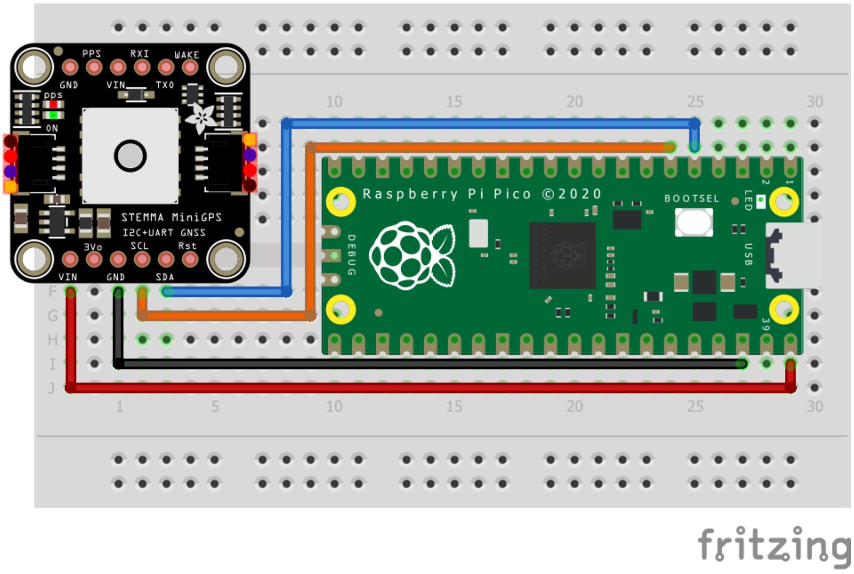


Figure 25. Wiring Diagram for PA1010D.

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/blob/master/i2c/pa1010d_i2c/CMakeLists.txt

```
add_executable(pa1010d_i2c
        pa1010d_i2c.c
        )

# pull in common dependencies and additional i2c hardware support
target_link_libraries(pa1010d_i2c pico_stdlib hardware_i2c)

# create map/bin/hex file etc.
pico_add_extra_outputs(pa1010d_i2c)

# add url via pico_set_program_url
example_auto_set_url(pa1010d_i2c)
```

pa1010d_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/pa1010d_i2c/pa1010d_i2c.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 "pico/binary_info.h"
#include "hardware/i2c.h"
#include "string.h"

```

```
/* Example code to talk to a PA1010D Mini GPS module.

   This example reads the Recommended Minimum Specific GNSS Sentence, which includes basic
location and time data, each second, formats and displays it.

   Connections on Raspberry Pi Pico board, other boards may vary.

   GPIO PICO_DEFAULT_I2C_SDA_PIN (On Pico this is 4 (physical pin 6)) -> SDA on PA1010D board
   GPIO PICO_DEFAULT_I2C_SCK_PIN (On Pico this is 5 (physical pin 7)) -> SCL on PA1010D board
   3.3v (physical pin 36) -> VCC on PA1010D board
   GND (physical pin 38)  -> GND on PA1010D board
*/

const int addr = 0x10;
#define MAX_READ 250

#ifdef i2c_default

void pa1010d_write_command(const char command[], int com_length) {
    // Convert character array to bytes for writing
    uint8_t int_command[com_length];

    for (int i = 0; i < com_length; ++i) {
        int_command[i] = command[i];
        i2c_write_blocking(i2c_default, addr, &int_command[i], 1, true);
    }
}

void pa1010d_parse_string(char output[], char protocol[]) {
    // Finds location of protocol message in output
    char *com_index = strstr(output, protocol);
    int p = com_index - output;

    // Splits components of output sentence into array
#define NO_OF_FIELDS 14
#define MAX_LEN 15

    int n = 0;
    int m = 0;

    char gps_data[NO_OF_FIELDS][MAX_LEN];
    memset(gps_data, 0, sizeof(gps_data));

    bool complete = false;
    while (output[p] != '$' && n < MAX_LEN && complete == false) {
        if (output[p] == ',' || output[p] == '*') {
            n += 1;
            m = 0;
        } else {
            gps_data[n][m] = output[p];
            // Checks if sentence is complete
            if (m < NO_OF_FIELDS) {
                m++;
            } else {
                complete = true;
            }
        }
        p++;
    }

    // Displays GNRMC data
    // Similarly, additional if statements can be used to add more protocols
    if (strcmp(protocol, "GNRMC") == 0) {
        printf("Protocol:%s\n", gps_data[0]);
```

```
        printf("UTC Time: %s\n", gps_data[1]);
        printf("Status: %s\n", gps_data[2][0] == 'V' ? "Data invalid. GPS fix not found." : "Data Valid");
        printf("Latitude: %s\n", gps_data[3]);
        printf("N/S indicator: %s\n", gps_data[4]);
        printf("Longitude: %s\n", gps_data[5]);
        printf("E/W indicator: %s\n", gps_data[6]);
        printf("Speed over ground: %s\n", gps_data[7]);
        printf("Course over ground: %s\n", gps_data[8]);
        printf("Date: %c%c/%c%c/%c%c\n", gps_data[9][0], gps_data[9][1], gps_data[9][2], gps_data[9][3], gps_data[9][4],
               gps_data[9][5]);
        printf("Magnetic Variation: %s\n", gps_data[10]);
        printf("E/W degree indicator: %s\n", gps_data[11]);
        printf("Mode: %s\n", gps_data[12]);
        printf("Checksum: %c%c\n", gps_data[13][0], gps_data[13][1]);
    }
}

void pa1010d_read_raw(char numcommand[]) {
    uint8_t buffer[MAX_READ];

    int i = 0;
    bool complete = false;

    i2c_read_blocking(i2c_default, addr, buffer, MAX_READ, false);

    // Convert bytes to characters
    while (i < MAX_READ && complete == false) {
        numcommand[i] = buffer[i];
        // Stop converting at end of message
        if (buffer[i] == 10 && buffer[i + 1] == 10) {
            complete = true;
        }
        i++;
    }
}

#endif

int main() {
    stdio_init_all();
#if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) || !defined(PICO_DEFAULT_I2C_SCL_PIN)
#warning i2c/mpu6050_i2c example requires a board with I2C pins
    puts("Default I2C pins were not defined");
#else

    char numcommand[MAX_READ];

    // Decide which protocols you would like to retrieve data from
    char init_command[] = "$PMTK314,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0*29\r\n";

    // This example will use I2C0 on the default SDA and SCL pins (4, 5 on a Pico)
    i2c_init(i2c_default, 400 * 1000);
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
    gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);

    // Make the I2C pins available to picotool
    bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C));

```

```
    printf("Hello, PA1010D! Reading raw data from module...\n");

    pa1010d_write_command(init_command, sizeof(init_command));

    while (1) {
        // Clear array
        memset(numcommand, 0, MAX_READ);
        // Read and re-format
        pa1010d_read_raw(numcommand);
        pa1010d_parse_string(numcommand, "GNRMC");

        // Wait for data to refresh
        sleep_ms(1000);

        // Clear terminal
        printf("\033[1;1H\033[2J");
    }
#endif
}
```

Bill of Materials

Table 50. A list of materials required for the example

Item	**Quantity**	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	https://www.raspberrypi.com/products/raspberry-pi-pico/
PA1010D board	1	https://shop.pimoroni.com/products/pa1010d-gps-breakout
M/M Jumper wires	4	generic part

Attaching a PCF8523 Real Time Clock via I2C

This example code shows how to interface the Raspberry Pi Pico to the PCF8523 Real Time Clock

This example allows you to initialise the current time and date and then displays it every half-second. Additionally it lets you set an alarm for a particular time and date and raises an alert accordingly. More information about the module is available at https://learn.adafruit.com/adafruit-pcf8523-real-time-clock.

Wiring information

Wiring up the device requires 4 jumpers, to connect VDD, 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 5V pin.

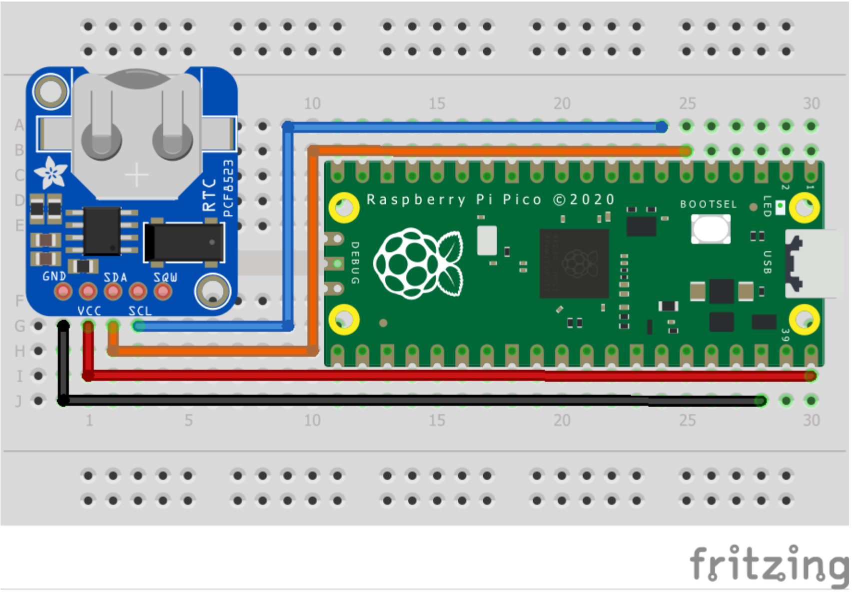


Figure 26. Wiring Diagram for PCF8523.

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/blob/master/i2c/pcf8523_i2c/CMakeLists.txt

```
add_executable(pcf8523_i2c
        pcf8523_i2c.c
        )

# pull in common dependencies and additional i2c hardware support
target_link_libraries(pcf8523_i2c pico_stdlib hardware_i2c)

# create map/bin/hex file etc.
pico_add_extra_outputs(pcf8523_i2c)

# add url via pico_set_program_url
example_auto_set_url(pcf8523_i2c)
```

pcf8523_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/pcf8523_i2c/pcf8523_i2c.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 "pico/binary_info.h"
#include "hardware/i2c.h"

/* Example code to talk to a PCF8520 Real Time Clock module
```

```

   Connections on Raspberry Pi Pico board, other boards may vary.

   GPIO PICO_DEFAULT_I2C_SDA_PIN (On Pico this is 4 (physical pin 6)) -> SDA on PCF8520 board
   GPIO PICO_DEFAULT_I2C_SCK_PIN (On Pico this is 5 (physical pin 7)) -> SCL on PCF8520 board
   5V (physical pin 40) -> VCC on PCF8520 board
   GND (physical pin 38)  -> GND on PCF8520 board
*/

#ifdef i2c_default

// By default these devices  are on bus address 0x68
static int addr = 0x68;

static void pcf8520_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[] = {0x00, 0x58};
    i2c_write_blocking(i2c_default, addr, buf, 2, false);
}

static void pcf820_write_current_time() {
    // buf[0] is the register to write to
    // buf[1] is the value that will be written to the register
    uint8_t buf[2];

    //Write values for the current time in the array
    //index 0 -> second: bits 4-6 are responsible for the ten's digit and bits 0-3 for the unit's digit
    //index 1 -> minute: bits 4-6 are responsible for the ten's digit and bits 0-3 for the unit's digit
    //index 2 -> hour: bits 4-5 are responsible for the ten's digit and bits 0-3 for the unit's digit
    //index 3 -> day of the month: bits 4-5 are responsible for the ten's digit and bits 0-3 for the unit's digit
    //index 4 -> day of the week: where Sunday = 0x00, Monday = 0x01, Tuesday... ...Saturday = 0x06
    //index 5 -> month: bit 4 is responsible for the ten's digit and bits 0-3 for the unit's digit
    //index 6 -> year: bits 4-7 are responsible for the ten's digit and bits 0-3 for the unit's digit

    //NOTE: if the value in the year register is a multiple for 4, it will be considered a leap year and hence will include the 29th of February

    uint8_t current_val[7] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

    for (int i = 3; i < 10; ++i) {
        buf[0] = i;
        buf[1] = current_val[i - 3];
        i2c_write_blocking(i2c_default, addr, buf, 2, false);
    }
}

static void pcf8520_read_raw(uint8_t *buffer) {
    // 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.

    // Start reading acceleration registers from register 0x3B for 6 bytes
    uint8_t val = 0x03;
    i2c_write_blocking(i2c_default, addr, &val, 1, true); // true to keep master control of bus
```

```c
    i2c_read_blocking(i2c_default, addr, buffer, 7, false);
}


void pcf8520_set_alarm() {
    // buf[0] is the register to write to
    // buf[1] is the value that will be written to the register
    uint8_t buf[2];

    // Default value of alarm register is 0x80
    // Set bit 8 of values to 0 to activate that particular alarm
    // Index 0 -> minute: bits 4-5 are responsible for the ten's digit and bits 0-3 for the unit's digit
    // Index 1 -> hour: bits 4-6 are responsible for the ten's digit and bits 0-3 for the unit's digit
    // Index 2 -> day of the month: bits 4-5 are responsible for the ten's digit and bits 0-3 for the unit's digit
    // Index 3 -> day of the week: where Sunday = 0x00, Monday = 0x01, Tuesday... ...Saturday = 0x06

    uint8_t alarm_val[4] = {0x01, 0x80, 0x80, 0x80};
    // Write alarm values to registers
    for (int i = 10; i < 14; ++i) {
        buf[0] = (uint8_t) i;
        buf[1] = alarm_val[i - 10];
        i2c_write_blocking(i2c_default, addr, buf, 2, false);
    }
}

void pcf8520_check_alarm() {
    // Check bit 3 of control register 2 for alarm flags
    uint8_t status[1];
    uint8_t val = 0x01;
    i2c_write_blocking(i2c_default, addr, &val, 1, true); // true to keep master control of bus
    i2c_read_blocking(i2c_default, addr, status, 1, false);

    if ((status[0] & 0x08) == 0x08) {
        printf("ALARM RINGING");
    } else {
        printf("Alarm not triggered yet");
    }
}


void pcf8520_convert_time(int conv_time[7], const uint8_t raw_time[7]) {
    // Convert raw data into time
    conv_time[0] = (10 * (int) ((raw_time[0] & 0x70) >> 4)) + ((int) (raw_time[0] & 0x0F));
    conv_time[1] = (10 * (int) ((raw_time[1] & 0x70) >> 4)) + ((int) (raw_time[1] & 0x0F));
    conv_time[2] = (10 * (int) ((raw_time[2] & 0x30) >> 4)) + ((int) (raw_time[2] & 0x0F));
    conv_time[3] = (10 * (int) ((raw_time[3] & 0x30) >> 4)) + ((int) (raw_time[3] & 0x0F));
    conv_time[4] = (int) (raw_time[4] & 0x07);
    conv_time[5] = (10 * (int) ((raw_time[5] & 0x10) >> 4)) + ((int) (raw_time[5] & 0x0F));
    conv_time[6] = (10 * (int) ((raw_time[6] & 0xF0) >> 4)) + ((int) (raw_time[6] & 0x0F));
}
#endif

int main() {
    stdio_init_all();
#if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) || !defined(PICO_DEFAULT_I2C_SCL_PIN)
#warning i2c/pcf8520_i2c example requires a board with I2C pins
    puts("Default I2C pins were not defined");
#else
```

```
    printf("Hello, PCF8520! Reading raw data from registers...\n");

    // This example will use I2C0 on the default SDA and SCL pins (4, 5 on a Pico)
    i2c_init(i2c_default, 400 * 1000);
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
    gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);
    // Make the I2C pins available to picotool
    bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN,
GPIO_FUNC_I2C));

    pcf8520_reset();

    pcf820_write_current_time();
    pcf8520_set_alarm();
    pcf8520_check_alarm();

    uint8_t raw_time[7];
    int real_time[7];
    char days_of_week[7][12] = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday"};

    while (1) {

        pcf8520_read_raw(raw_time);
        pcf8520_convert_time(real_time, raw_time);

        printf("Time: %02d : %02d : %02d\n", real_time[2], real_time[1], real_time[0]);
        printf("Date: %s %02d / %02d / %02d\n", days_of_week[real_time[4]], real_time[3],
real_time[5], real_time[6]);
        pcf8520_check_alarm();

        sleep_ms(500);

        // Clear terminal
        printf("\033[1;1H\033[2J");
    }
#endif
}
```

Bill of Materials

Table 51. A list of materials required for the example

Item	**Quantity**	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	https://www.raspberrypi.com/products/raspberry-pi-pico/
PCF8523 board	1	https://www.adafruit.com/product/3295
M/M Jumper wires	4	generic part

Interfacing 1-Wire devices to the Pico

This example demonstrates how to use 1-Wire devices with the Raspberry Pi Pico (RP2040). 1-Wire is an interface that enables a master to control several slave devices over a simple shared serial bus.

The example provides a 1-Wire library that is used to take readings from a set of connected DS18B20 1-Wire temperature sensors. The results are sent to the default serial terminal connected via USB or UART as configured in the SDK.

The library uses a driver based on the RP2040 PIO state machine to generate accurate bus timings and control the 1-Wire bus via a GPIO pin.

1-Wire® is a registered trademark of Maxim Integrated Products, Inc.

Wiring information

Connect one or more DS18B20 sensors to the Pico as shown in the diagram and table below.

Connect GPIO 15 to 3V3(OUT) with a pull-up resistor of about 4k ohms.

Figure 27. Wiring diagram

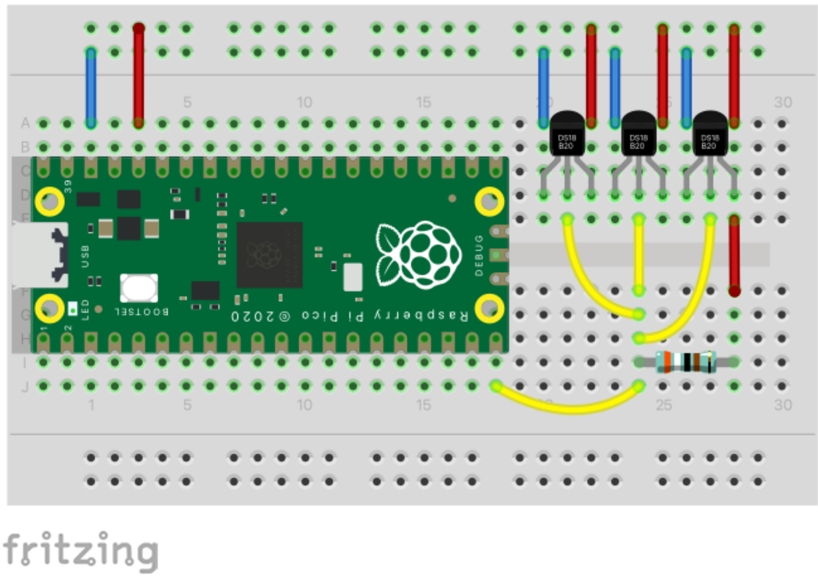


Table 52. Connections table

Pico	pin	DS18B20	pin / sensor wire
GND	38 or equivalent	GND	1 / Black
GPIO 15	20	DQ	2 / Yellow
3V3(OUT)	36	VDD	3 / Red

Bill of materials

Table 53. A list of materials for the example circuit

Item	Quantity	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	https://www.raspberrypi.com/products/raspberry-pi-pico/
DS18B20	3	chip or wired sensor
3900 ohm resistor	1	generic part
M/M jumper wire	13	generic part

List of files

CMakeLists.txt

CMake file to incorporate the example in the build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/onewire/CMakeLists.txt

```
add_executable(pio_onewire)

target_sources(pio_onewire PRIVATE onewire.c)

add_subdirectory(onewire_library)

target_link_libraries(pio_onewire PRIVATE
    pico_stdlib
    hardware_pio
    onewire_library)

pico_add_extra_outputs(pio_onewire)

# add url via pico_set_program_url
example_auto_set_url(pio_onewire)
```

onewire.c

Source code for the example program.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/onewire/onewire.c

```
/**
 * Copyright (c) 2023 mjcross
 *
 * SPDX-License-Identifier: BSD-3-Clause
**/

#include <stdio.h>
#include "pico/stdlib.h"
#include "pico/binary_info.h"

#include "onewire_library.h"    // onewire library functions
#include "ow_rom.h"             // onewire ROM command codes
#include "ds18b20.h"            // ds18b20 function codes

// Demonstrates the PIO onewire driver by taking readings from a set of
// ds18b20 1-wire temperature sensors.

int main() {
    stdio_init_all();

    PIO pio = pio0;
    uint gpio = 15;

    OW ow;
    uint offset;
    // add the program to the PIO shared address space
    if (pio_can_add_program (pio, &onewire_program)) {
        offset = pio_add_program (pio, &onewire_program);

        // claim a state machine and initialise a driver instance
        if (ow_init (&ow, pio, offset, gpio)) {

```

```
            // find and display 64-bit device addresses
            int maxdevs = 10;
            uint64_t romcode[maxdevs];
            int num_devs = ow_romsearch (&ow, romcode, maxdevs, OW_SEARCH_ROM);

            printf("Found %d devices\n", num_devs);
            for (int i = 0; i < num_devs; i += 1) {
                printf("\t%d: 0x%llx\n", i, romcode[i]);
            }
            putchar ('\n');

            while (num_devs > 0) {
                // start temperature conversion in parallel on all devices
                // (see ds18b20 datasheet)
                ow_reset (&ow);
                ow_send (&ow, OW_SKIP_ROM);
                ow_send (&ow, DS18B20_CONVERT_T);

                // wait for the conversions to finish
                while (ow_read(&ow) == 0);

                // read the result from each device
                for (int i = 0; i < num_devs; i += 1) {
                    ow_reset (&ow);
                    ow_send (&ow, OW_MATCH_ROM);
                    for (int b = 0; b < 64; b += 8) {
                        ow_send (&ow, romcode[i] >> b);
                    }
                    ow_send (&ow, DS18B20_READ_SCRATCHPAD);
                    int16_t temp = 0;
                    temp = ow_read (&ow) | (ow_read (&ow) << 8);
                    printf ("\t%d: %f", i, temp / 16.0);
                }
                putchar ('\n');
            }

        } else {
            puts ("could not initialise the driver");
        }
    } else {
        puts ("could not add the program");
    }

    while(true);
}
```

ow_rom.h

Header file with generic ROM command codes for 1-Wire devices.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/onewire/ow_rom.h

```
// Generic ROM commands for 1-Wire devices
// https://www.analog.com/en/technical-articles/guide-to-1wire-communication.html
//
#define OW_READ_ROM         0x33
#define OW_MATCH_ROM        0x55
#define OW_SKIP_ROM         0xCC
#define OW_ALARM_SEARCH     0xEC
#define OW_SEARCH_ROM       0xF0
```

ds18b20.h

Header file with function command codes for the DS18B20 device.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/onewire/ds18b20.h

```
// Function commands for d218b20 1-Wire temperature sensor
// https://www.analog.com/en/products/ds18b20.html
//
#define DS18B20_CONVERT_T           0x44
#define DS18B20_WRITE_SCRATCHPAD    0x4e
#define DS18B20_READ_SCRATCHPAD     0xbe
#define DS18B20_COPY_SCRATCHPAD     0x48
#define DS18B20_RECALL_EE           0xb8
#define DS18B20_READ_POWER_SUPPLY   0xb4
```

onewire_library/

Subdirectory containing the 1-Wire library and driver.

onewire_library/CMakeLists.txt

CMake file to build the 1-Wire library and driver.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/onewire/onewire_library/CMakeLists.txt

```
add_library(onewire_library INTERFACE)
target_sources(onewire_library INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}/onewire_library.c)

# invoke pio_asm to assemble the state machine programs
#
pico_generate_pio_header(onewire_library ${CMAKE_CURRENT_LIST_DIR}/onewire_library.pio)

target_link_libraries(onewire_library INTERFACE
        pico_stdlib
        hardware_pio
        )

# add the `binary` directory so that the generated headers are included in the project
#
target_include_directories(onewire_library INTERFACE
    ${CMAKE_CURRENT_SOURCE_DIR}
    ${CMAKE_CURRENT_BINARY_DIR}
    )
```

onewire_library/onewire_library.c

Source code for the 1-Wire user functions.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/onewire/onewire_library/onewire_library.c

```
/**
 * Copyright (c) 2023 mjcross
 *
 * SPDX-License-Identifier: BSD-3-Clause
**/

#include "pico/stdlib.h"
#include "hardware/gpio.h"
#include "hardware/pio.h"

#include "onewire_library.h"

```

```

// Create a driver instance and populate the provided OW structure.
// Returns: True on success.
// ow: A pointer to a blank OW structure to hold the driver parameters.
// pio: The PIO hardware instance to use.
// offset: The location of the onewire program in the PIO shared address space.
// gpio: The pin to use for the bus (NB: see the README).
bool ow_init (OW *ow, PIO pio, uint offset, uint gpio) {
    int sm = pio_claim_unused_sm (pio, false);
    if (sm == -1) {
        return false;
    }
    gpio_init (gpio);                // enable the gpio and clear any output value
    pio_gpio_init (pio, gpio);       // set the function to PIO output
    ow->gpio = gpio;
    ow->pio = pio;
    ow->offset = offset;
    ow->sm = (uint)sm;
    ow->jmp_reset = onewire_reset_instr (ow->offset);   // assemble the bus reset instruction
    onewire_sm_init (ow->pio, ow->sm, ow->offset, ow->gpio, 8); // set 8 bits per word
    return true;
}


// Send a binary word on the bus (LSB first).
// ow: A pointer to an OW driver struct.
// data: The word to be sent.
void ow_send (OW *ow, uint data) {
    pio_sm_put_blocking (ow->pio, ow->sm, (uint32_t)data);
    pio_sm_get_blocking (ow->pio, ow->sm);  // discard the response
}


// Read a binary word from the bus.
// Returns: the word read (LSB first).
// ow: pointer to an OW driver struct
uint8_t ow_read (OW *ow) {
    pio_sm_put_blocking (ow->pio, ow->sm, 0xff);    // generate read slots
    return (uint8_t)(pio_sm_get_blocking (ow->pio, ow->sm) >> 24);  // shift response into bits 0..7
}


// Reset the bus and detect any connected slaves.
// Returns: true if any slaves responded.
// ow: pointer to an OW driver struct
bool ow_reset (OW *ow) {
    pio_sm_exec_wait_blocking (ow->pio, ow->sm, ow->jmp_reset);
    if ((pio_sm_get_blocking (ow->pio, ow->sm) & 1) == 0) {     // apply pin mask (see pio program)
        return true;    // a slave pulled the bus low
    }
    return false;
}


// Find ROM codes (64-bit hardware addresses) of all connected devices.
// See https://www.analog.com/en/app-notes/1wire-search-algorithm.html
// Returns: the number of devices found (up to maxdevs) or -1 if an error occurrred.
// ow: pointer to an OW driver struct
// romcodes: location at which store the addresses (NULL means don't store)
// maxdevs: maximum number of devices to find (0 means no limit)
// command: 1-Wire search command (e.g. OW_SEARCHROM or OW_ALARM_SEARCH)
int ow_romsearch (OW *ow, uint64_t *romcodes, int maxdevs, uint command) {
```

```
    int index;
    uint64_t romcode = 0ull;
    int branch_point;
    int next_branch_point = -1;
    int num_found = 0;
    bool finished = false;

    onewire_sm_init (ow->pio, ow->sm, ow->offset, ow->gpio, 1); // set driver to 1-bit mode

    while (finished == false && (maxdevs == 0 || num_found < maxdevs )) {
        finished = true;
        branch_point = next_branch_point;
        if (ow_reset (ow) == false) {
            num_found = 0;     // no slaves present
            finished = true;
            break;
        }
        for (int i = 0; i < 8; i += 1) {    // send search command as single bits
            ow_send (ow, command >> i);
        }
        for (index = 0; index < 64; index += 1) {   // determine romcode bits 0..63 (see ref)
            uint a = ow_read (ow);
            uint b = ow_read (ow);
            if (a == 0 && b == 0) {         // (a, b) = (0, 0)
                if (index == branch_point) {
                    ow_send (ow, 1);
                    romcode |= (1ull << index);
                } else {
                    if (index > branch_point || (romcode & (1ull << index)) == 0) {
                        ow_send(ow, 0);
                        finished = false;
                        romcode &= ~(1ull << index);
                        next_branch_point = index;
                    } else {                // index < branch_point or romcode[index] = 1
                        ow_send (ow, 1);
                    }
                }
            } else if (a != 0 && b != 0) {  // (a, b) = (1, 1) error (e.g. device disconnected)
                num_found = -2;             // function will return -1
                finished = true;
                break;                      // terminate for loop
            } else {
                if (a == 0) {               // (a, b) = (0, 1) or (1, 0)
                    ow_send (ow, 0);
                    romcode &= ~(1ull << index);
                } else {
                    ow_send (ow, 1);
                    romcode |= (1ull << index);
                }
            }
        }                                   // end of for loop

        if (romcodes != NULL) {
            romcodes[num_found] = romcode;  // store the romcode
        }
        num_found += 1;
    }                                       // end of while loop

    onewire_sm_init (ow->pio, ow->sm, ow->offset, ow->gpio, 8); // restore 8-bit mode
    return num_found;
}
```

onewire_library/onewire_library.h

Header file for the 1-Wire user functions and types.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/onewire/onewire_library/onewire_library.h

```
#ifndef _ONEWIRE_LIBRARY_H
#define _ONEWIRE_LIBRARY_H

#include "hardware/pio.h"
#include "hardware/clocks.h"           // for clock_get_hz() in generated header
#include "onewire_library.pio.h"       // generated by pioasm

typedef struct {
    PIO pio;
    uint sm;
    uint jmp_reset;
    int offset;
    int gpio;
} OW;

bool ow_init (OW *ow, PIO pio, uint offset, uint gpio);
void ow_send (OW *ow, uint data);
uint8_t ow_read (OW *ow);
bool ow_reset (OW *ow);
int ow_romsearch (OW *ow, uint64_t *romcodes, int maxdevs, uint command);

#endif
```

onewire_library/onewire_library.pio

PIO assembly code for the 1-Wire driver.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/onewire/onewire_library/onewire_library.pio

```
;
; Copyright (c) 2023 mjcross
;
; SPDX-License-Identifier: BSD-3-Clause
;

; Implements a Maxim 1-Wire bus with a GPIO pin.
;
; Place data words to be transmitted in the TX FIFO and read the results from the
; RX FIFO. To reset the bus execute a jump to 'reset_bus' using the opcode from
; the provided function.
;
; At 1us per cycle as initialised below the timings are those recommended by:
; https://www.analog.com/en/technical-articles/1wire-communication-through-software.html
;
; Notes:
;   (1) The code will stall with the bus in a safe state if the FIFOs are empty/full.
;   (2) The bus must be pulled up with an external pull-up resistor of about 4k.
;       The internal GPIO resistors are too high (~50k) to work reliably for this.
;   (3) Do not connect the GPIO pin directly to a bus powered at more than 3.3V.

.program onewire
.side_set 1 pindirs

PUBLIC reset_bus:
        set x, 28       side 1  [15]    ; pull bus low                          16
loop_a: jmp x-- loop_a  side 1  [15]    ;                                     29 x 16
        set x, 8        side 0  [6]     ; release bus                            7
```

```
loop_b: jmp x-- loop_b  side 0  [6]     ;                                 9 x 7

        mov isr, pins   side 0          ; read all pins to ISR (avoids autopush) 1
        push            side 0          ; push result manually               1
        set x, 24       side 0  [7]     ;                                     8
loop_c: jmp x-- loop_c  side 0  [15]    ;                                25 x 16

.wrap_target
PUBLIC fetch_bit:
        out x, 1        side 0          ; shift next bit from OSR (autopull)     1
        jmp !x  send_0  side 1  [5]     ; pull bus low, branch if sending '0'    6

send_1: ; send a '1' bit
        set x, 2        side 0  [8]     ; release bus, wait for slave response   9
        in pins, 1      side 0  [4]     ; read bus, shift bit to ISR (autopush)  5
loop_e: jmp x-- loop_e  side 0  [15]    ;                                 3 x 16
        jmp fetch_bit   side 0          ;                                     1

send_0: ; send a '0' bit
        set x, 2        side 1  [5]     ; continue pulling bus low            6
loop_d: jmp x-- loop_d  side 1  [15]    ;                                 3 x 16
        in null, 1      side 0  [8]     ; release bus, shift 0 to ISR (autopush) 9
.wrap
;; (17 instructions)


% c-sdk {
static inline void onewire_sm_init (PIO pio, uint sm, uint offset, uint pin_num, uint bits_per_word) {

    // create a new state machine configuration
    pio_sm_config c = onewire_program_get_default_config (offset);

    // Input Shift Register configuration settings
    sm_config_set_in_shift (
        &c,
        true,           // shift direction: right
        true,           // autopush: enabled
        bits_per_word   // autopush threshold
    );

    // Output Shift Register configuration settings
    sm_config_set_out_shift (
        &c,
        true,           // shift direction: right
        true,           // autopull: enabled
        bits_per_word   // autopull threshold
    );

    // configure the input and sideset pin groups to start at `pin_num`
    sm_config_set_in_pins (&c, pin_num);
    sm_config_set_sideset_pins (&c, pin_num);

    // configure the clock divider for 1 usec per instruction
    float div = clock_get_hz (clk_sys) * 1e-6;
    sm_config_set_clkdiv (&c, div);

    // apply the configuration and initialise the program counter
    pio_sm_init (pio, sm, offset + onewire_offset_fetch_bit, &c);

    // enable the state machine
    pio_sm_set_enabled (pio, sm, true);
}
```

```
static inline uint onewire_reset_instr (uint offset) {
    // encode a "jmp reset_bus side 0" instruction for the state machine
    return pio_encode_jmp (offset + onewire_offset_reset_bus) | pio_encode_sideset (1, 0);
}
%}
```

Communicating as master and slave via SPI

This example code shows how to interface two RP2040 microcontrollers to each other using SPI.

Wiring information

Function	Master (RP2040)	Slave (RP2040)	Master (Pico)	Slave (Pico)
MOSI	DO0	DI0	25	21
SCLK	SCK0	SCK0	24	24
GND	GND	GND	23	23
CS	CS0	CS0	22	22
MISO	DI0	DO0	21	25

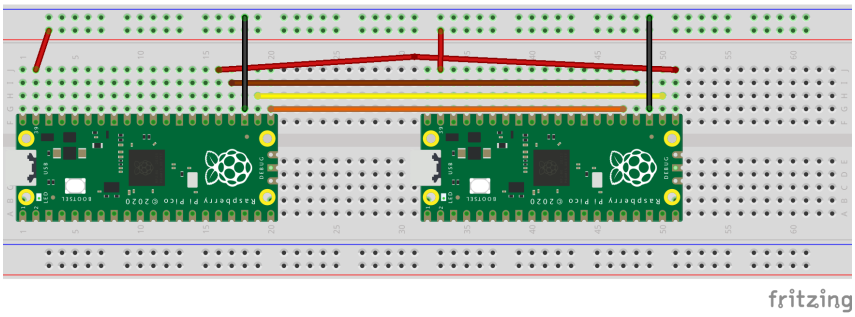


Figure 28. Wiring Diagram for SPI Master and Slave.

At least one of the boards should be powered, and will share power to the other.

If the master is not connected properly to a slave, the master will report reading all zeroes.

If the slave is not connected properly to a master, it will initialize but never transmit nor receive, because it's waiting for clock signal from the master.

Outputs

Both master and slave boards will give output to stdio.

With master and slave properly connected, the master should output something like this:

```
SPI master example
SPI master says: The following buffer will be written to MOSI endlessly:
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
```

```
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f
90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
SPI master says: read page 0 from the MISO line:
ff fe fd fc fb fa f9 f8 f7 f6 f5 f4 f3 f2 f1 f0
ef ee ed ec eb ea e9 e8 e7 e6 e5 e4 e3 e2 e1 e0
df de dd dc db da d9 d8 d7 d6 d5 d4 d3 d2 d1 d0
cf ce cd cc cb ca c9 c8 c7 c6 c5 c4 c3 c2 c1 c0
bf be bd bc bb ba b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
af ae ad ac ab aa a9 a8 a7 a6 a5 a4 a3 a2 a1 a0
9f 9e 9d 9c 9b 9a 99 98 97 96 95 94 93 92 91 90
8f 8e 8d 8c 8b 8a 89 88 87 86 85 84 83 82 81 80
7f 7e 7d 7c 7b 7a 79 78 77 76 75 74 73 72 71 70
6f 6e 6d 6c 6b 6a 69 68 67 66 65 64 63 62 61 60
5f 5e 5d 5c 5b 5a 59 58 57 56 55 54 53 52 51 50
4f 4e 4d 4c 4b 4a 49 48 47 46 45 44 43 42 41 40
3f 3e 3d 3c 3b 3a 39 38 37 36 35 34 33 32 31 30
2f 2e 2d 2c 2b 2a 29 28 27 26 25 24 23 22 21 20
1f 1e 1d 1c 1b 1a 19 18 17 16 15 14 13 12 11 10
0f 0e 0d 0c 0b 0a 09 08 07 06 05 04 03 02 01 00
```

The slave should output something like this:

```
SPI slave example
SPI slave says: When reading from MOSI, the following buffer will be written to MISO:
ff fe fd fc fb fa f9 f8 f7 f6 f5 f4 f3 f2 f1 f0
ef ee ed ec eb ea e9 e8 e7 e6 e5 e4 e3 e2 e1 e0
df de dd dc db da d9 d8 d7 d6 d5 d4 d3 d2 d1 d0
cf ce cd cc cb ca c9 c8 c7 c6 c5 c4 c3 c2 c1 c0
bf be bd bc bb ba b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
af ae ad ac ab aa a9 a8 a7 a6 a5 a4 a3 a2 a1 a0
9f 9e 9d 9c 9b 9a 99 98 97 96 95 94 93 92 91 90
8f 8e 8d 8c 8b 8a 89 88 87 86 85 84 83 82 81 80
7f 7e 7d 7c 7b 7a 79 78 77 76 75 74 73 72 71 70
6f 6e 6d 6c 6b 6a 69 68 67 66 65 64 63 62 61 60
5f 5e 5d 5c 5b 5a 59 58 57 56 55 54 53 52 51 50
4f 4e 4d 4c 4b 4a 49 48 47 46 45 44 43 42 41 40
3f 3e 3d 3c 3b 3a 39 38 37 36 35 34 33 32 31 30
2f 2e 2d 2c 2b 2a 29 28 27 26 25 24 23 22 21 20
1f 1e 1d 1c 1b 1a 19 18 17 16 15 14 13 12 11 10
0f 0e 0d 0c 0b 0a 09 08 07 06 05 04 03 02 01 00
SPI slave says: read page 0 from the MOSI line:
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f
90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
```

```
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
```

If you look at the communication with a logic analyzer, you should see this:

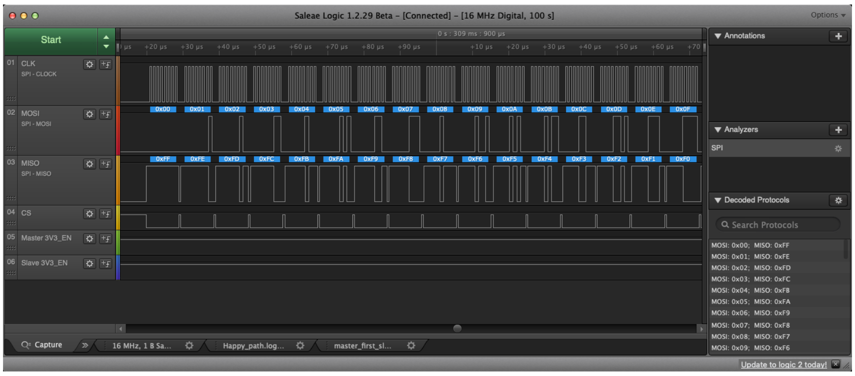


Figure 29. Data capture as seen in Saleae Logic.

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/blob/master/spi/spi_master_slave/CMakeLists.txt

```
add_subdirectory_exclude_platforms(spi_master)
add_subdirectory_exclude_platforms(spi_slave)
```

spi_master/spi_master.c

The example code for SPI master.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/spi/spi_master_slave/spi_master/spi_master.c

```
// Copyright (c) 2021 Michael Stoops. All rights reserved.
// Portions copyright (c) 2021 Raspberry Pi (Trading) Ltd.
//
// Redistribution and use in source and binary forms, with or without modification, are permitted provided that the
// following conditions are met:
//
// 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following
//    disclaimer.
// 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the
//    following disclaimer in the documentation and/or other materials provided with the distribution.
// 3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote
//    products derived from this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "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. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
```

```c
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 THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
// SPDX-License-Identifier: BSD-3-Clause
//
// Example of an SPI bus master using the PL022 SPI interface

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

#define BUF_LEN         0x100

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

    // append trailing newline if there isn't one
    if (i % 16) {
        putchar('\n');
    }
}

int main() {
    // Enable UART so we can print
    stdio_init_all();
#if !defined(spi_default) || !defined(PICO_DEFAULT_SPI_SCK_PIN) || !defined(PICO_DEFAULT_SPI_TX_PIN) || !defined(PICO_DEFAULT_SPI_RX_PIN) || !defined(PICO_DEFAULT_SPI_CSN_PIN)
#warning spi/spi_master example requires a board with SPI pins
    puts("Default SPI pins were not defined");
#else

    printf("SPI master example\n");

    // Enable SPI 0 at 1 MHz and connect to GPIOs
    spi_init(spi_default, 1000 * 1000);
    gpio_set_function(PICO_DEFAULT_SPI_RX_PIN, GPIO_FUNC_SPI);
    gpio_set_function(PICO_DEFAULT_SPI_SCK_PIN, GPIO_FUNC_SPI);
    gpio_set_function(PICO_DEFAULT_SPI_TX_PIN, GPIO_FUNC_SPI);
    gpio_set_function(PICO_DEFAULT_SPI_CSN_PIN, GPIO_FUNC_SPI);
    // Make the SPI pins available to picotool
    bi_decl(bi_4pins_with_func(PICO_DEFAULT_SPI_RX_PIN, PICO_DEFAULT_SPI_TX_PIN, PICO_DEFAULT_SPI_SCK_PIN, PICO_DEFAULT_SPI_CSN_PIN, GPIO_FUNC_SPI));

    uint8_t out_buf[BUF_LEN], in_buf[BUF_LEN];

    // Initialize output buffer
    for (size_t i = 0; i < BUF_LEN; ++i) {
        out_buf[i] = i;
    }

```

```
    printf("SPI master says: The following buffer will be written to MOSI endlessly:\n");
    printbuf(out_buf, BUF_LEN);

    for (size_t i = 0; ; ++i) {
        // Write the output buffer to MOSI, and at the same time read from MISO.
        spi_write_read_blocking(spi_default, out_buf, in_buf, BUF_LEN);

        // Write to stdio whatever came in on the MISO line.
        printf("SPI master says: read page %d from the MISO line:\n", i);
        printbuf(in_buf, BUF_LEN);

        // Sleep for ten seconds so you get a chance to read the output.
        sleep_ms(10 * 1000);
    }
#endif
}
```

spi_slave/spi_slave.c

The example code for SPI slave.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/spi/spi_master_slave/spi_slave/spi_slave.c

```
// Copyright (c) 2021 Michael Stoops. All rights reserved.
// Portions copyright (c) 2021 Raspberry Pi (Trading) Ltd.
//
// Redistribution and use in source and binary forms, with or without modification, are permitted provided that the
// following conditions are met:
//
// 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following
//    disclaimer.
// 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the
//    following disclaimer in the documentation and/or other materials provided with the distribution.
// 3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote
//    products derived from this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "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. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS 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 THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
// SPDX-License-Identifier: BSD-3-Clause
//
// Example of an SPI bus slave using the PL022 SPI interface

#include <stdio.h>
#include <string.h>
#include "pico/stdlib.h"
#include "pico/binary_info.h"
```

```c
#include "hardware/spi.h"

#define BUF_LEN         0x100

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

    // append trailing newline if there isn't one
    if (i % 16) {
        putchar('\n');
    }
}


int main() {
    // Enable UART so we can print
    stdio_init_all();
#if !defined(spi_default) || !defined(PICO_DEFAULT_SPI_SCK_PIN) || !defined(PICO_DEFAULT_SPI_TX_PIN) || !defined(PICO_DEFAULT_SPI_RX_PIN) || !defined(PICO_DEFAULT_SPI_CSN_PIN)
#warning spi/spi_slave example requires a board with SPI pins
    puts("Default SPI pins were not defined");
#else

    printf("SPI slave example\n");

    // Enable SPI 0 at 1 MHz and connect to GPIOs
    spi_init(spi_default, 1000 * 1000);
    spi_set_slave(spi_default, true);
    gpio_set_function(PICO_DEFAULT_SPI_RX_PIN, GPIO_FUNC_SPI);
    gpio_set_function(PICO_DEFAULT_SPI_SCK_PIN, GPIO_FUNC_SPI);
    gpio_set_function(PICO_DEFAULT_SPI_TX_PIN, GPIO_FUNC_SPI);
    gpio_set_function(PICO_DEFAULT_SPI_CSN_PIN, GPIO_FUNC_SPI);
    // Make the SPI pins available to picotool
    bi_decl(bi_4pins_with_func(PICO_DEFAULT_SPI_RX_PIN, PICO_DEFAULT_SPI_TX_PIN, PICO_DEFAULT_SPI_SCK_PIN, PICO_DEFAULT_SPI_CSN_PIN, GPIO_FUNC_SPI));

    uint8_t out_buf[BUF_LEN], in_buf[BUF_LEN];

    // Initialize output buffer
    for (size_t i = 0; i < BUF_LEN; ++i) {
        // bit-inverted from i. The values should be: {0xff, 0xfe, 0xfd...}
        out_buf[i] = ~i;
    }

    printf("SPI slave says: When reading from MOSI, the following buffer will be written to MISO:\n");
    printbuf(out_buf, BUF_LEN);

    for (size_t i = 0; ; ++i) {
        // Write the output buffer to MISO, and at the same time read from MOSI.
        spi_write_read_blocking(spi_default, out_buf, in_buf, BUF_LEN);

        // Write to stdio whatever came in on the MOSI line.
        printf("SPI slave says: read page %d from the MOSI line:\n", i);
        printbuf(in_buf, BUF_LEN);
    }
#endif
```

```
90 }
```

Bill of Materials

Table 54. A list of materials required for the example

Item	Quantity	Details
Breadboard	1	generic part
Raspberry Pi Pico	2	https://www.raspberrypi.com/products/raspberry-pi-pico/
M/M Jumper wires	8	generic part

Appendix B: Building the SDK API documentation

The SDK documentation can be viewed online, but is also part of the SDK itself and can be built directly from the command line. If you haven't already checked out the SDK repository you should do so,

```
$ cd ~/
$ mkdir pico
$ cd pico
$ git clone https://github.com/raspberrypi/pico-sdk.git --branch master
$ cd pico-sdk
$ git submodule update --init
$ cd ..
$ git clone https://github.com/raspberrypi/pico-examples.git --branch master
```

Install doxygen if you don't already have it,

```
$ sudo apt install doxygen
```

Then afterwards you can go ahead and build the documentation for all platforms:

```
$ cd pico-sdk
$ mkdir build
$ cd build
$ cmake -DPICO_EXAMPLES_PATH=../../pico-examples -DPICO_NO_PICOTOOL=1 -DPICO_PLATFORM=combined
-docs ..
$ make docs
```

The API documentation will be built and can be found in the `pico-sdk/build/docs/doxygen/html` directory, see Figure 30.

TIP

If you prefer to build documentation for a single platform only, then replace `-DPICO_PLATFORM=combined-docs` with `-DPICO_PLATFORM=rp2040` or `-DPICO_PLATFORM=rp2350` in the above, using a fresh build directory.

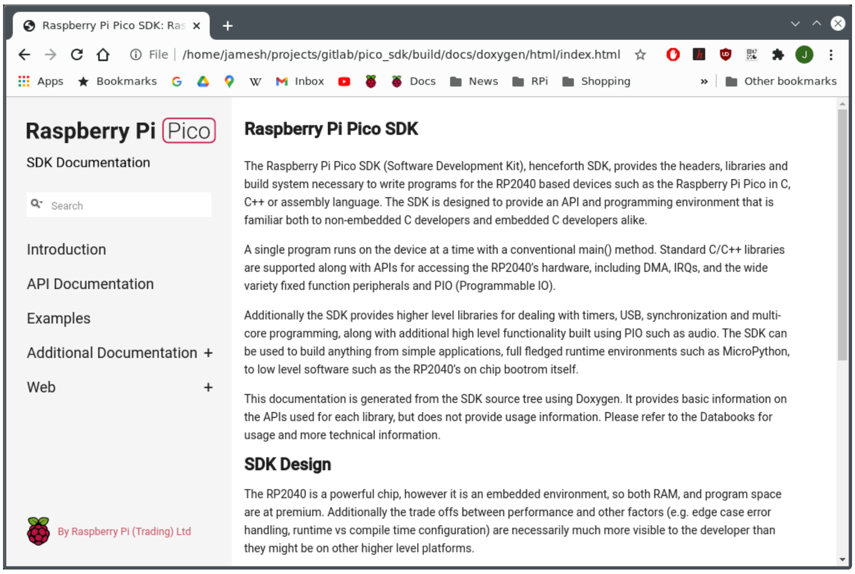


Figure 30. The SDK API documentation

Appendix C: SDK release history

Release 1.0.0 (20 January 2021)

- Initial release

Release 1.0.1 (01 February 2021)

- add `pico_get_unique_id` method to return a unique identifier for a Pico board using the identifier of the external flash
- exposed all 4 pacing timers on the DMA peripheral (previously only 2 were exposed)
- fixed ninja build (i.e. `cmake -G ninja .. ; ninja`)
- minor other improvements and bug fixes

Boot Stage 2

Additionally, a low level change was made to the way flash binaries start executing after `boot_stage2`. This was at the request of folks implementing other language runtimes. It is not generally of concern to end users, however it did require a change to the linker scripts so if you have cloned those to make modifications then you need to port across the relevant changes. If you are porting a different language runtime using the SDK boot_stage2 implementations then you should be aware that you should now have a vector table (rather than executable code) - at `0x10000100`.

Release 1.1.0 (05 March 2021)

- Added board headers for Adafruit, Pimoroni & SparkFun boards
  - new values for `PICO_BOARD` are `adafruit_feather_rp2040`, `adafruit_itsybitsy_rp2040`, `adafruit_qtpy_rp2040`, `pimoroni_keybow2040`, `pimoroni_picosystem`, `pimoroni_tiny2040`, `sparkfun_micromod`, `sparkfun_promicro`, `sparkfun_thingplus`, in addition to the existing `pico` and `vgaboard`.
  - Added additional definitions for a default SPI, I2C pins as well as the existing ones for UART
  - Allow *default* pins to be undefined (not all boards have UART for example), and SDK will compile but warn as needed in the absence of default.
  - Added additional definition for a default WS2812 compatible pin (currently unused).
- New reset options
  - Added `pico_bootsel_via_double_reset` library to allow reset to `BOOTSEL` mode via double press of a `RESET` button
  - When using `pico_stdio_usb` i.e. `stdio` connected via USB CDC to host, setting baud rate to 1200 (by default) can optionally be used to reset into `BOOTSEL` mode.
  - When using `pico-stdio_usb` i.e. `stdio` connected via USB CDC to host, an additional interface may be added to give `picotool` control over resetting the device.
- Build improvement for non-SDK or existing library builds
  - Removed additional compiler warnings (register headers now use `_u(x)` macro for unsigned values though).
  - Made build more clang friendly.

This release also contains many bug fixes, documentation updates and minor improvements.

Backwards incompatibility

There are some nominally backwards incompatible changes not worthy of a major version bump:

- `PICO_DEFAULT_UART_` defines now default to undefined if there is no default rather than `-1` previously
- The broken `multicore_sleep_core1()` API has been removed; `multicore_reset_core1` is already available to put core 1 into a deep sleep.

Release 1.1.1 (01 April 2021)

This fixes a number of bugs, and additionally adds support for a board configuration header to choose the `boot_stage2`.

Introduced absolutely no jokes at all.

Release 1.1.2 (07 April 2021)

Fixes issues with `boot_stage2` selection

Release 1.2.0 (03 June 2021)

This release contains numerous bug fixes and documentation improvements. Additionally it contains the following improvements/notable changes:

CAUTION

The `lib/tinyusb` submodule has been updated from 0.8.0 and now tracks upstream https://github.com/hathach/tinyusb.git. It is worth making sure you do a

```
$ git submodule sync
$ git submodule update
```

to make sure you are correctly tracking upstream TinyUSB if you are not checking out a clean `pico-sdk` repository.

Moving from TinyUSB 0.8.0 to TinyUSB 0.10.1 may require some minor changes to your USB code.

New/improved Board headers

- New board headers support for PICO_BOARDs `arduino_nano_rp240_connect`, `pimoroni_picolipo_4mb` and `pimoroni_picolipo_16mb`
- Missing/new `#defines` for default SPI and I2C pins have been added

Updated TinyUSB to 0.10.1

The `lib/tinyusb` submodule has been updated from 0.8.0 and now tracks upstream https://github.com/hathach/tinyusb.git

Added CMSIS core headers

CMSIS core headers (e.g. `core_cm0plus.h` and `RP2040.h`) are made available via `cmsis_core` INTERFACE library. Additionally, CMSIS standard exception naming is available via `PICO_CMSIS_RENAME_EXCEPTIONS=1`

API improvements

`pico_sync`

- Added support for recursive mutexes via `recursive_mutex_init()` and `auto_init_recursive_mutex()`
- Added `mutex_enter_timeout_us()`
- Added `critical_section_deinit()`
- Added `sem_acquire_timeout_ms()` and `sem_acquire_block_until()`

`hardware_adc`

- Added `adc_get_selected_input()`

`hardware_clocks`

- `clock_get_hz()` now returns actual achieved frequency rather than desired frequency

`hardware_dma`

- Added `dma_channel_is_claimed()`
- Added new methods for configuring/acknowledging DMA IRQs. `dma_irqn_set_channel_enabled()`, `dma_irqn_set_channel_mask_enabled()`, `dma_irqn_get_channel_status()`, `dma_irqn_acknowledge_channel()` etc.

`hardware_exception`

New library for setting ARM exception handlers:

- Added `exception_set_exclusive_handler()`, `exception_restore_handler()`, `exception_get_vtable_handler()`

`hardware_flash`

- Exposed previously private function `flash_do_cmd()` for low level flash command execution

`hardware_gpio`

- Added `gpio_set_input_hysteresis_enabled()`, `gpio_is_input_hysteresis_enabled()`, `gpio_set_slew_rate()`, `gpio_get_slew_rate()`, `gpio_set_drive_strength()`, `gpio_get_drive_strength()`, `gpio_get_out_level()`, `gpio_set_irqover()`

`hardware_i2c`

- Corrected a number of incorrect hardware register definitions
- A number of edge cases in the i2c code fixed

hardware_interp

- Added interp_lane_is_claimed(), interp_unclaim_lane_mask()

hardware_irq

- Notably fixed the PICO_LOWEST/HIGHEST_IRQ_PRIORITY values which were backwards!

hardware_pio

- Added new methods for configuring/acknowledging PIO interrupts (pio_set_irqn_source_enabled(), pio_set_irqn_source_mask_enabled(), pio_interrupt_get(), pio_interrupt_clear() etc.)
- Added pio_sm_is_claimed()

hardware_spi

- Added spi_get_baudrate()
- Changed spi_init() to return the set/achieved baud rate rather than void
- Changed spi_is_writable() to return bool not size_t (it was always 1/0)

hardware_sync

- Notable documentation improvements for spin lock functions
- Added spin_lock_is_claimed()

hardware_timer

- Added busy_wait_ms() to match busy_wait_us()
- Added hardware_alarm_is_claimed()

pico_float/pico_double

- Correctly save/restore divider state if floating point is used from interrupts

pico_int64_ops

- Added PICO_INT64_OPS_IN_RAM flag to move code into RAM to avoid veneers when calling code is in RAM

pico_runtime

- Added ability to override panic function by setting PICO_PANIC_FUNCTION=foo to then use foo as the implementation, or setting PICO_PANIC_FUNCITON= to simply breakpoint, saving some code space

pico_unique_id

- Added pico_get_unique_board_id_string().

General code improvements

- Removed additional classes of compiler warnings
- Added some missing `const` to method parameters

SVD

- USB DPRAM for device mode is now included

pioasm

- Added `#pragma once` to C/C++ output

RTOS interoperability

Improvements designed to make porting RTOSes either based on the SDK or supporting SDK code easier.

- Added `PICO_DIVIDER_DISABLE_INTERRUPTS` flag to optionally configure all uses of the hardware divider to be guarded by disabling interrupts, rather than requiring on the RTOS to save/restore the divider state on context switch
- Added new abstractions to `pico/lock_core.h` to allow an RTOS to inject replacement code for SDK based low level wait, notify and sleep/timeouts used by synchronization primitives in `pico_sync` and for `sleep_` methods. If an RTOS implements these few simple methods, then all SDK semaphore, mutex, queue, sleep methods can be safely used both within/to/from RTOS tasks, but also to communicate with non-RTOS task aware code, whether it be existing libraries and IRQ handlers or code running perhaps (though not necessarily) on the other core

CMake build changes

Substantive changes have been made to the CMake build, so if you are using a hand crafted non-CMake build, you **will** need to update your compile/link flags. Additionally changed some possibly confusing status messages from CMake build generation to be debug only

Boot Stage 2

- New boot stage 2 for `AT25SF128A`

Release 1.3.0 (02 November 2021)

This release contains numerous bug fixes and documentation improvements. Additionally, it contains the following notable changes/improvements:

Updated TinyUSB to 0.12.0

- The `lib/tinyusb` submodule has been updated from 0.10.1 to 0.12.0. See https://github.com/hathach/tinyusb/releases/tag/0.11.0 and https://github.com/hathach/tinyusb/releases/tag/0.12.0 for release notes.
- Improvements have been made for projects that include TinyUSB and also compile with enhanced warning levels and `-Werror`. Warnings have been fixed in RP2040 specific TinyUSB code, and in TinyUSB headers, and a new cmake function `suppress_tinyusb_warnings()` has been added, that you may call from your `CMakeLists.txt` to suppress warnings in other TinyUSB C files.

New Board Support

The following boards have been added and may be specified via PICO_BOARD:

- adafruit_trinkey_qt2040
- melopero_shake_rp2040
- pimoroni_interstate75
- pimoroni_plasma2040
- pybstick26_rp2040
- waveshare_rp2040_lcd_0.96
- waveshare_rp2040_plus_4mb
- waveshare_rp2040_plus_16mb
- waveshare_rp2040_zero

Updated SVD, hardware_regs, hardware_structs

The RP2040 SVD has been updated, fixing some register access types and adding new documentation.

The hardware_regs headers have been updated accordingly.

The hardware_structs headers which were previously hand coded, are now generated from the SVD, and retain select documentation from the SVD, including register descriptions and register bit-field tables.

e.g. what was once

```
typedef struct {
    io_rw_32 ctrl;
    io_ro_32 fstat;
    ...
```

becomes:

```
// Reference to datasheet: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf#tab-
registerlist_pio
//
// The _REG_ macro is intended to help make the register navigable in your IDE (for example, using
the "Go to Definition" feature)
// _REG_(x) will link to the corresponding register in hardware/regs/pio.h.
//
// Bit-field descriptions are of the form:
// BITMASK [BITRANGE]: FIELDNAME (RESETVALUE): DESCRIPTION

typedef struct {
    _REG_(PIO_CTRL_OFFSET) // PIO_CTRL
    // PIO control register
    // 0x00000f00 [11:8]  : CLKDIV_RESTART (0): Restart a state machine's clock divider from an
initial phase of 0
    // 0x000000f0 [7:4]   : SM_RESTART (0): Write 1 to instantly clear internal SM state which may
be otherwise difficult...
    // 0x0000000f [3:0]   : SM_ENABLE (0): Enable/disable each of the four state machines by
writing 1/0 to each of these four bits
    io_rw_32 ctrl;

    _REG_(PIO_FSTAT_OFFSET) // PIO_FSTAT
```

```
    // FIFO status register
    // 0x0f000000 [27:24] : TXEMPTY (0xf): State machine TX FIFO is empty
    // 0x000f0000 [19:16] : TXFULL (0): State machine TX FIFO is full
    // 0x00000f00 [11:8]  : RXEMPTY (0xf): State machine RX FIFO is empty
    // 0x0000000f [3:0]   : RXFULL (0): State machine RX FIFO is full
    io_ro_32 fstat;
    ...
```

Behavioural Changes

There were some behavioural changes in this release:

pico_sync

SDK 1.2.0 previously added recursive mutex support using the existing (previously non-recursive) `mutex_` functions. This caused a performance regression, and the only clean way to fix the problem was to return the `mutex_` functions to their pre-SDK 1.2.0 behaviour, and split the recursive mutex functionality out into separate `recursive_mutex_` functions with a separate `recursive_mutex_` type.

Code using the SDK 1.2.0 recursive mutex functionality will need to be changed to use the new type and functions, however as a convenience, the pre-processor define `PICO_MUTEX_ENABLE_SDK120_COMPATIBILITY` may be set to 1 to retain the SDK 1.2.0 behaviour at the cost of an additional performance penalty. The ability to use this pre-processor define will be removed in a subsequent SDK version.

pico_platform

- `pico.h` and its dependencies have been slightly refactored so it can be included by assembler code as well as C/C code. This ensures that assembler code and C/C code follow the same board configuration/override order and see the same configuration defines. This should not break any existing code, but is notable enough to mention.
- `pico/platform.h` is now fully documented.

pico_standard_link

`-Wl,max-page-size=4096` is now passed to the linker, which is beneficial to certain users and should have no discernible impact on the rest.

Other Notable Improvements

hardware_base

- Added `xip_noalloc_alias(addr)`, `xip_nocache_alias(addr)`, `xip_nocache_noalloc_alias(addr)` macros for converting a flash address between XIP aliases (similar to the `hw_xxx_alias(addr)` macros).

hardware_dma

- Added `dma_timer_claim()`, `dma_timer_unclaim()`, `dma_claim_unused_timer()` and `dma_timer_is_claimed()` to manage ownership of DMA timers.
- Added `dma_timer_set_fraction()` and `dma_get_timer_dreq()` to facilitate pacing DMA transfers using DMA timers.

hardware_i2c

- Added i2c_get_dreq() function to facilitate configuring DMA transfers to/from an I2C instance.

hardware_irq

- Added irq_get_priority().
- Fixed implementation when PICO_DISABLE_SHARED_IRQ_HANDLERS=1 is specified, and allowed irq_add_shared_handler to be used in this case (as long as there is only one handler - i.e. it behaves exactly like irq_set_exclusive_handler).
- Sped up IRQ priority initialization which was slowing down per core initialization.

hardware_pio

- pio_encode_ functions in hardware/pico_instructions.h are now documented.

hardware_pwm

- Added pwm_get_dreq() function to facilitate configuring DMA transfers to a PWM slice.

hardware_spi

- Added spi_get_dreq() function to facilitate configuring DMA transfers to/from an SPI instance.

hardware_uart

- Added uart_get_dreq() function to facilitate configuring DMA transfers to/from a UART instance.

hardware_watchdog

- Added watchdog_enable_caused_reboot() to distinguish a watchdog reboot caused by a watchdog timeout after calling watchdog_enable() from other watchdog reboots (e.g. that are performed when a UF2 is dragged onto a device in BOOTSEL mode).

pico_bootrom

- Added new constants and function signature typedefs to pico/bootrom.h to facilitate calling bootrom functions directly.

pico_multicore

- Improved documentation in pico/multicore.h; particularly, multicore_lockout_ functions are newly documented.

pico_platform

- PICO_RP2040 is now defined to 1 in PICO_PLATFORM=rp2040 (i.e. normal) builds.

pico_stdio

- Added puts_raw() and putchar_raw() to skip CR/LF translation if enabled.

- Added stdio_usb_connected() to detect CDC connection when using stdio_usb.
- Added PICO_STDIO_USB_CONNECT_WAIT_TIMEOUT_MS define that can be set to wait for a CDC connection to be established during initialization of stdio_usb. Note: value -1 means indefinite. This can be used to prevent initial program output being lost, at the cost of requiring an active CDC connection.
- Fixed semihosting_putc which was completely broken.

pico_usb_reset_interface

- This new library contains pico/usb_reset_interface.h split out from stdio_usb to facilitate inclusion in external projects.

CMake build

- OUTPUT_NAME target property is now respected when generating supplemental files (.BIN, .HEX, .MAP, .UF2)

pioasm

- Operator precedence of *, /, -, + have been fixed
- Incorrect MicroPython output has been fixed.

elf2uf2

- A bug causing an error with binaries produced by certain other languages has been fixed.

Release 1.3.1 (18 May 2022)

This release contains numerous bug fixes and documentation improvements which are not all listed here; you can see the full list of individual commits here.

New Board Support

The following boards have been added and may be specified via PICO_BOARD:

- adafruit_kb2040
- adafruit_macropad_rp2040
- eetree_gamekit_rp2040
- garatronic_pybstick26_rp2040 (renamed from pybstick26_rp2040)
- pimoroni_badger2040
- pimoroni_motor2040
- pimoroni_servo2040
- pimoroni_tiny2040_2mb
- seeed_xiao_rp2040
- solderparty_rp2040_stamp_carrier
- solderparty_rp2040_stamp

- `wiznet_w5100s_evb_pico`

Notable Library Changes/Improvements

`hardware_dma`

- New documentation has been added to the `dma_channel_abort()` function describing errata RP2040-E13, and how to work around it.

`hardware_irq`

- Fixed a bug related to removing and then re-adding shared IRQ handlers. It is now possible to add/remove handlers as documented.
- Added new documentation clarifying the fact the shared IRQ handler ordering "priorities" have values that increase with higher priority vs. Cortex M0+ IRQ priorites which have values that decrease with priority!

`hardware_pwm`

- Added a `pwm_config_set_clkdiv_int_frac()` method to complement `pwm_config_set_clkdiv_int()` and `pwm_config_set_clkdiv()`.

`hardware_pio`

- Fixed the `pio_set_irqn_source_mask_enabled()` method which previously affected the wrong IRQ.

`hardware_rtc`

- Added clarification to `rtc_set_datetime()` documentation that the new value may not be visible to a `rtc_get_datetime()` very soon after, due to crossing of clock domains.

`pico_platform`

- Added a `busy_wait_at_least_cycles()` method as a convenience method for a short tight-loop counter-based delay.

`pico_stdio`

- Fixed a bug related to removing stdio "drivers". `stdio_set_driver_enabled()` can now be used freely to dynamically enable and disable drivers during runtime.

`pico_time`

- Added an `is_at_the_end_of_time()` method to check if a given time matches the SDK's maximum time value.

Runtime

A bug in `__ctzdi2()` aka `__builtin_ctz(uint64_t)` was fixed.

Build

- Compilation with GCC 11 is now supported.
- PIOASM_EXTRA_SOURCE_FILES is now actually respected.

pioasm

- Input files with Windows (CRLF) line endings are now accepted.
- A bug in the python output was fixed.

elf2uf2

- Extra padding was added to the UF2 output of misaligned or non-contiguous binaries to work around errata RP2040-E14.

NOTE

The 1.3.0 release of the SDK incorrectly squashed the history of the changes. A new merge commit has been added to restore the full history, and the 1.3.0 tag has been updated

Release 1.4.0 (30 June 2022)

This release adds wireless support for the Raspberry Pi Pico W, adds support for other new boards, and contains various bug fixes, documentation improvements, and minor improvements/added functionality. You can see the full list of individual commits here.

New Board Support

The following boards have been added and may be specified via PICO_BOARD:

- pico_w
- datanoisetv_rp2040_dsp
- solderparty_rp2040_stamp_round_carrier

Wireless Support

- Support for the Raspberry Pi Pico W is now included with the SDK (PICO_BOARD=pico_w). The Pico W uses a driver for the wireless chip called cyw43_driver which is included as a submodule of the SDK. You need to initialize this submodule for Pico W wireless support to be available. Note that the LED on the Pico W board is only accessible via the wireless chip, and can be accessed via cyw43_arch_gpio_put() and cyw43_arch_gpio_get() (part of the pico_cyw43_arch library described below). As a result of the LED being on the wireless chip, there is no PICO_DEFAULT_LED_PIN setting and the default LED based examples in pico-examples do not work with the Pico W.
- IP support is provided by lwIP which is also included as a submodule which you should initialize if you want to use it.

  The following libraries exposing lwIP functionality are provided by the SDK:

  - pico_lwip_core (included in pico_lwip)
  - pico_lwip_core4 (included in pico_lwip)

  - pico_lwip_core6 (included in pico_lwip)
  - pico_lwip_netif (included in pico_lwip)
  - pico_lwip_sixlowpan (included in pico_lwip)
  - pico_lwip_ppp (included in pico_lwip)
  - pico_lwip_api (this is a blocking API that may be used with FreeRTOS and is not included in pico_lwip)

  As referenced above, the SDK provides a pico_lwip which aggregates all of the commonly needed lwIP functionality. You are of course free to use the substituent libraries explicitly instead.

  The following libraries are provided that contain the equivalent lwIP application support:

  - pico_lwip_snmp
  - pico_lwip_http
  - pico_lwip_makefsdata
  - pico_lwip_iperf
  - pico_lwip_smtp
  - pico_lwip_sntp
  - pico_lwip_mdns
  - pico_lwip_netbios
  - pico_lwip_tftp
  - pico_lwip_mbedtls

- Integration of the IP stack and the cyw43_driver network driver into the user's code is handled by pico_cyw43_arch. Both the IP stack and the driver need to do work in response to network traffic, and pico_cyw43_arch provides a variety of strategies for servicing that work. Four architecture variants are currently provided as libraries:

  - pico_cyw43_arch_lwip_poll - For using the RAW lwIP API (NO_SYS=1 mode) with polling. With this architecture the user code must periodically poll via cyw43_arch_poll() to perform background work. This architecture matches the common use of lwIP on microcontrollers, and provides no multicore safety
  - pico_cyw43_arch_lwip_threadsafe_background - For using the RAW lwIP API (NO_SYS=1 mode) with multicore safety, and automatic servicing of the cyw43_driver and lwIP in the background. User polling is not required with this architecture, but care should be taken as lwIP callbacks happen in an IRQ context.
  - pico_cyw43_arch_lwip_sys_freertos - For using the full lwIP API including blocking sockets in OS mode (NO_SYS=0), along with multicore/task safety, and automatic servicing of the cyw43_driver and the lwIP stack in a separate task. This powerful architecture works with both SMP and non-SMP variants of the RP2040 port of FreeRTOS-Kernel. Note you must set FREERTOS_KERNEL_PATH in your build to use this variant.
  - pico_cyw43_arch_none - If you do not need the TCP/IP stack but wish to use the on-board LED or other wireless chip connected GPIOs.

  See the library documentation or the pico/cyw43_arch.h header for more details.

Notable Library Changes/Improvements

hardware_dma

- Added dma_unclaim_mask() function for un-claiming multiple DMA channels at once.
- Added channel_config_set_high_priority() function to set the channel priority via a channel config object.

hardware_gpio

- Improved the documentation for the pre-existing gpio IRQ functions which use the "one callback per core" callback mechanism, and added a `gpio_set_irq_callback()` function to explicitly set the callback independently of enabling per pin GPIO IRQs.
- Reduced the latency of calling the existing "one callback per core" GPIO IRQ callback.
- Added new support for the user to add their own shared GPIO IRQ handler independent of the pre-existing "one callback per core" callback mechanism, allowing for independent usage of GPIO IRQs without having to share one handler.

  See the documentation in `hardware/irq.h` for full details of the functions added:

  - `gpio_add_raw_irq_handler()`
  - `gpio_add_raw_irq_handler_masked()`
  - `gpio_add_raw_irq_handler_with_order_priority()`
  - `gpio_add_raw_irq_handler_with_order_priority_masked()`
  - `gpio_remove_raw_irq_handler()`
  - `gpio_remove_raw_irq_handler_masked()`
- Added a `gpio_get_irq_event_mask()` utility function for use by the new "raw" IRQ handlers.

hardware_irq

- Added `user_irq_claim()`, `user_irq_unclaim()`, `user_irq_claim_unused()` and `user_irq_is_claimed()` functions for claiming ownership of the **user** IRQs (the ones numbered 26-31 and not connected to any hardware). Uses of the **user** IRQs have been updated to use these functions. For `stdio_usb`, the `PICO_STDIO_USB_LOW_PRIORITY_IRQ` define is still respected if specified, but otherwise an unclaimed one is chosen.
- Added an `irq_is_shared_handler()` function to determine if a particular IRQ uses a shared handler.

pico_sync

- Added a `sem_try_acquire()` function, for non-blocking acquisition of a semaphore.

pico_stdio

- `stderr` is now supported and goes to the same destination as `stdout`.
- Zero timeouts for `getchar_timeout_us()` are now correctly honored (previously they were a 1μs minimum).

stdio_usb

- The use of a 1ms timer to handle background TinyUSB work has been replaced with use of a more interrupt driven approach using a **user** IRQ for better performance. Note this new feature is disabled if shared IRQ handlers are disabled via `PICO_DISABLE_SHARED_IRQ_HANDLERS=1`

Miscellaneous

- `get_core_num()` has been moved to `pico/platform.h` from `hardware/sync.h`.
- The C library function `realloc()` is now multicore safe too.
- The minimum PLL frequency has been increased from 400 MHz to 750 MHz to improve stability across operating conditions. This should not affect the majority of users in any way, but may impact those trying to set particularly

low clock frequencies. If you do wish to return to the previous minimum, you can set `PICO_PLL_VCO_MIN_FREQ_MHZ` back to `400`. There is also a new `PICO_PLL_VCO_MAX_FREQ_MHZ` which defaults to `1600`.

Build

- Compilation with GCC 12 is now supported.

Release 1.5.0 (11 February 2023)

This release contains new libraries and functionality, along with numerous bug fixes and documentation improvements.

Highlights are listed below, or you can see the full list of individual commits here, and the full list of resolved issues here.

New Board Support

The following boards have been added and may be specified via `PICO_BOARD`:

- `nullbits_bit_c_pro`
- `waveshare_rp2040_lcd_1.28`
- `waveshare_rp2040_one`

Library Changes/Improvements

`hardware_clocks`

- `clock_gpio_init()` now takes a `float` for the clock divider value, rather than an `int`.
- Added `clock_gpio_init_int_frac()` function to allow initialization of integer and fractional part of the clock divider value, without using `float`.
- Added `--ref-min` option to `vcocalc.py` to override the minimum reference frequency allowed.
- `vcocalc.py` now additionally considers reference frequency dividers greater than 1.

`hardware_divider`

- Improved the performance of `hw_divider_` functions.

`hardware_dma`

- Added `dma_sniffer_set_output_invert_enabled()` and `dma_sniffer_set_output_reverse_enabled()` functions to configure the DMA sniffer.
- Added `dma_sniffer_set_data_accumulator()` and `dma_sniffer_get_data_accumulator()` functions to access the DMA sniffer accumulator.

`hardware_i2c`

- Added `i2c_get_instance()` function for consistency with other `hardware_` libraries.

- Added i2c_read_byte_raw(), i2c_write_byte_raw() functions to directly read and write the I2C data register for an I2C instance.

hardware_timer

- Added hardware_alarm_claim_unused() function to claim an unused hardware timer.

pico_cyw43_arch

- Added cyw43_arch_wifi_connect_bssid_ variants of cyw43_arch_wifi_connect_ functions to allow connection to a specific access point.
- Blocking cyw43_arch_wifi_connect_ functions now continue trying to connect rather than failing immediately if the network is not found.
- cyw43_arch_wifi_connect_ functions now return consistent return codes (PICO_OK, or PICO_ERROR_XXX).
- The pico_cyw43_arch library has been completely rewritten on top of the new pico_async_context library that generically abstracts the different types of asynchronous operation (poll, threadsafe_background and freertos) previously handled in a bespoke fashion by pico_cyw43_arch. Many edge case bugs have been fixed as a result of this. Note that this change should be entirely backwards compatible from the user point of view.
- cyw43_arch_init() and cyw43_arch_deinit() functions are now very thin layers which handle async_context life-cycles, along with adding support for the cyw43_driver, lwIP, BTstack etc. to that async_context. Currently, these mechanisms remain the preferred documented way to initialize Pico W networking, however you are free to do similar initialization/de-initialization yourself.
- Added cyw43_arch_set_async_context() function to specify a custom async_context prior to calling cyw43_arch_init*()
- Added cyw43_arch_async_context() function to get the async_context used by the CYW43 architecture support.
- Added cyw43_arch_init_default_async_context() function to return the async_context that cyw43_arch_init*() would initialize if one has not been set by the user.
- Added cyw43_arch_wait_for_work_until() function to block until there is networking work to be done. This is most useful for poll style applications that have no other work to do and wish to sleep until cyw43_arch_poll() needs to be called again.

pico_cyw43_driver

- The functionality has been clarified into 3 separate libraries:
  - cyw43_driver - the raw cyw43_driver code.
  - cyw43_driver_picow - additional support for communication with the Wi-Fi chip over SPI on Pico W.
  - pico_cyw43_driver - integration of the cyw43_driver with the pico-sdk via async_context
- Added CYW43_WIFI_NVRAM_INCLUDE_FILE define to allow user to override the NVRAM file.

pico_divider

- Improved the performance of 64-bit divider functions.

pico_platform

- Add panic_compact() function that discards the message to save space in non-debug (NEBUG defined) builds.

`pico_runtime`

- Added proper implementation of certain missing `newlib` system APIs: `_gettimeofday()`, `_times()`, `_isatty()`, `_getpid()`.
- The above changes enable certain additional C/C++ library functionality such as `gettimeofday()`, `times()` and `std::chrono`.
- Added `settimeofday()` implementation such that `gettimeofday()` can be meaningfully used.
- Added default (return `-1`) implementations of the remaining `newlib` system APIs: `_open()`, `_close()`, `_lseek()`, `_fstat()`, `_isatty()`, `_kill()`, to prevent warnings on GCC 12.
- Made all `newlib` system API implementations *weak* so the user can override them.

`pico_stdio`

- `pico_stdio` allows for outputting from within an IRQ handler that creates the potential for deadlocks (especially with `pico_stdio_usb`), and the intention is to *not* deadlock but instead discard output in any cases where a deadlock would otherwise occur. The code has been revamped to avoid more deadlock cases, and a new define `PICO_STDIO_DEADLOCK_TIMEOUT_MS` has been added to catch remaining cases that might be caused by user level locking.
- Added `stdio_set_chars_available_callback()` function to set a callback to be called when input is available. See also the new `PICO_STDIO_USB_SUPPORT_CHARS_AVAILABLE_CALLBACK` and `PICO_STDIO_UART_SUPPORT_CHARS_AVAILABLE_CALLBACK` defines which both default to `1` and control the availability of this new feature for USB and UART stdio respectively (at the cost of a little more code).
- Improved performance of `stdio_semihosting`.
- Give the user more control over the USB descriptors of `stdio_usb` via `USBD_VID`, `USBD_PID`, `USBD_PRODUCT`, `PICO_STDIO_USB_CONNECTION_WITHOUT_DTR` and `PICO_STDIO_USB_DEVICE_SELF_POWERED`

`pico_sync`

- Added `critical_section_is_initialized()` function to test if a critical section has been initialized.
- Added `mutex_try_enter_block_until()` function to wait only up to a certain time to acquire a mutex.

`pico_time`

- Added `from_us_since_boot()` function to convert a `uint64_t` timestamp to an `absolute_time_t`.
- Added `absolute_time_min()` function to return the earlier of two `absolute_time_t` values.
- Added `alarm_pool_create_with_unused_hardware_alarm()` function to create an alarm pool using a hardware alarm number claimed using `hardware_alarm_claim()`.
- Added `alarm_pool_core_num()` function to determine what core an alarm pool runs on.
- Added `alarm_pool_add_alarm_at_force_in_context()` function to add an alarm, and have it always run in the IRQ context even if the target time is in the past, or during the call. This may be simpler in some cases than dealing with the `fire_if_past` parameters to existing functions, and avoids some callbacks happening from non-IRQ context.

`pico_lwip`

- Added `pico_lwip_mqtt` library to expose the MQTT app functionality in lwIP.
- Added `pico_lwip_mdns` library to expose the MDNS app functionality in lwIP.
- Added `pico_lwip_freertos` library for `NO_SYS=0` with FreeRTOS as a complement to `pico_lwip_nosys` for `NO_SYS=1`.

TinyUSB

- TinyUSB has upgraded from 0.12.0 to 0.15.0. See TinyUSB release notes here for details.
- Particularly *host* support should be massively improved.
- Defaulted new TinyUSB `dcd_rp2040` driver's `TUD_OPT_RP2040_USB_DEVICE_UFRAME_FIX` variable to `1` as a workaround for errata RP2040-E15. This fix is required for correctness, but comes at the cost of some performance, so applications that won't ever be plugged into a Pi 4 or Pi 400 can optionally disable this by setting the value of `TUD_OPT_RP2040_USB_DEVICE_UFRAME_FIX` to `0` either via `target_compile_definitions` in their `CMakeLists.txt` or in their `tusb_config.h`.

New Libraries

`pico_async_context`

- Provides support for asynchronous events (timers/IRQ notifications) to be handled in a safe context without concurrent execution (as required by many asynchronous 3rd party libraries).
- Provides implementations matching those previously implemented in `pico_cyw43_arch`:
  - `poll` - Not thread-safe; the user must call `async_context_poll()` periodically from their main loop, but can call `async_context_wait_for_work_until()` to block until work is required.
  - `threadsafe_background` - No polling is required; instead asynchronous work is performed in a low priority IRQ. Locking is provided such that IRQ/non-IRQ or multiple cores can interact safely.
  - `freertos` - Asynchronous work is performed in a separate FreeRTOS task.
- `async_context` guarantees all callbacks happen on a single core.
- `async_context` supports multiple instances for providing independent context which can execute concurrently with respect to each other.

`pico_i2c_slave`

- A (slightly modified) pico_i2c_slave library from https://github.com/vmilea/pico_i2c_slave
- Adds a callback style event API for handling I2C slave requests.

`pico_mbedtls`

- Added `pico_mbedtls` library to provide MBed TLS support. You can depend on both `pico_lwip_mbedtls` and `pico_mbedtls` to use MBed TLS and lwIP together. See the tls_client example in `pico-examples` for more details.

`pico_rand`

- Implements a new Random Number Generator API.
- `pico_rand` generates random numbers at runtime utilizing a number of possible entropy sources, and uses those sources to modify the state of a 128-bit 'Pseudo Random Number Generator' implemented in software.
- Adds `get_rand_32()`, `get_rand_64()` and `get_rand_128()` functions to return largely unpredictable random numbers (which should be different on each board/run for example).

Miscellaneous

- Added a new header hardware/structs/nvic.h with a struct for the Arm Cortex M0+ NVIC available via the nvic_hw pointer.
- Added new PICO_CXX_DISABLE_ALLOCATION_OVERRIDES which can be set to 1 if you do not want pico_standard_link to include non-exceptional overrides of std::new, std::new[], std::delete and std::delete[] when exceptions are disabled.
- elf2uf2 now correctly uses LMA instead of VMA of the entry point to determine binary type (flash/RAM). This is required to support some exotic binaries correctly.

Build

- The build will now check for a functional compiler via the standard CMake mechanism.
- The build will pick up pre-installed elf2uf2 and pioasm if found via an installed pico-sdk-tools CMake package. If it can do so, then no native compiler is required for the build!
- It is now possible to switch the board type PICO_BOARD in an existing CMake build directory.
- ARCHIVE_OUTPUT_DIRECTORY is now respected in build for UF2 output files.
- Spaces are now supported in the path to the pico-sdk
- All libraries xxx in the pico-sdk now support a xxx_headers variant that just pulls in the libraries' headers. These xxx_headers libraries correctly mirror the dependencies of the xxx libraries, so you can use xxx_headers instead of xxx as your dependency if you do not want to pull in any implementation files (perhaps if you are making a STATIC library). Actually the "all" is not quite true, non-code libraries such as pico_standard_link and pico_cxx_options are an exception.

Bluetooth Support for Pico W (BETA)

The support is currently available as a beta. More details will be forthcoming with the actual release. In the meantime, there are examples in pico-examples.

Key changes:

- The Bluetooth API is provided by BTstack.
- The following new libraries are provided that expose core BTstack functionality:
  - pico_btstack_ble - Adds Bluetooth Low Energy (LE) support.
  - pico_btstack_classic - Adds Bluetooth Classic support.
  - pico_btstack_sbc_encoder - Adds Bluetooth Sub Band Coding (SBC) encoder support.
  - pico_btstack_sbc_decoder - Adds Bluetooth Sub Band Coding (SBC) decoder support.
  - pico_btstack_bnep_lwip - Adds Bluetooth Network Encapsulation Protocol (BNEP) support using LwIP.
  - pico_btstack_bnep_lwip_sys_freertos - Adds Bluetooth Network Encapsulation Protocol (BNEP) support using LwIP with FreeRTOS for NO_SYS=0.
- The following integration libraries are also provided:
  - pico_btstack_run_loop_async_context - provides a common async_context backed implementation of a BTstack *"run loop"* that can be used for all BTstack use with the pico-sdk.
  - pico_btstack_flash_bank - provides a sample implementation for storing required Bluetooth state in flash.
  - pico_btstack_cyw43 - integrates BTstack with the CYW43 driver.
- Added CMake function pico_btstack_make_gatt_header that can be used to run the BTstack compile_gatt tool to make a

GATT header file from a BTstack GATT file.

- Updated pico_cyw43_driver and cyw43_driver to support HCI communication for Bluetooth.
- Updated cyw43_driver_picow to support Pico W specific HCI communication for Bluetooth over SPI.
- Updated cyw43_arch_init() and cyw43_arch_deinit() to additionally handle Bluetooth support if CYW43_ENABLE_BLUETOOTH is 1 (as it will be automatically if you depend on pico_btstack_cyw43).

Release 1.5.1 (14 June 2023)

This release is largely a bug fix release, however it also makes Bluetooth support official and adds some new libraries and functionality.

Highlights are listed below, or you can see the full list of individual commits here, and the full list of resolved issues here.

Board Support

The following board has been added and may be specified via PICO_BOARD:

- pololu_3pi_2040_robot

The following board configurations have been modified:

- adafruit_itsybitsy_rp2040 - corrected the mismatched PICO_DEFAULT_I2C bus number (favors the breadboard pins not the stemma connector).
- sparkfun_thingplus - added WS2812 pin config.

Library Changes/Improvements

hardware_dma

- Added dma_channel_cleanup() function that can be used to clean up a dynamically claimed DMA channel after use, such that it won't be in a surprising state for the next user, making sure that any in-flight transfer is aborted, and no interrupts are left pending.

hardware_spi

- The spi_set_format, spi_set_slave, spi_set_baudrate functions that modify the configuration of an SPI instance, now disable the SPI while changing the configuration as specified in the data sheet.

pico_async_context

- Added user_data member to async_when_pending_worker_t to match async_at_time_worker_t.

pico_cyw43_arch

- Added cyw43_arch_disable_sta_mode() function to complement cyw43_arch_enable_sta_mode().
- Added cyw43_arch_disable_ap_mode() function to complement cyw43_arch_enable_ap_mode().

pico_stdio_usb

- The 20-character limit for descriptor strings `USBD_PRODUCT` and `USBD_MANUFACTURER` can now be extended by defining `USBD_DESC_STR_MAX`.
- `PICO_STDIO_USB_CONNECT_WAIT_TIMEOUT_MS` is now supported in the build as well as compiler definitions; if it is set in the build, it is added to the compile definitions.

pico_rand

- Fixed poor randomness when `PICO_RAND_ENTROPY_SRC_BUS_PERF_COUNTER=1`.

PLL and Clocks

- The `set_sys_clock_pll` and `set_sys_clock_khz` methods now reference a pre-processor define `PICO_CLOCK_AJDUST_PERI_CLOCK_WITH_SYS_CLOCK`. If set to `1`, the peripheral clock is updated to match the new system clock, otherwise the preexisting behavior (of setting the peripheral clock to a safe 48 MHz) is preserved.
- Support for non-standard crystal frequencies, and compile-time custom clock configurations:
  - The new define `XOSC_KHZ` is used in preference to the preexisting `XOSC_MHZ` to define the crystal oscillator frequency. This value is now also correctly plumbed through the various clock setup functions, such that they behave correctly with a crystal frequency other than 12 MHz. `XOSC_MHZ` will be automatically defined for backwards compatibility if `XOSC_KHZ` is an exact multiple of 1000 KHz. Note that either `XOSC_MHZ` or `XOSC_KHZ` may be specified by the user, but not both.
  - The new define `PLL_COMMON_REFDIV` can be specified to override the default reference divider of 1.
  - The new defines `PLL_SYS_VCO_FREQ_KHZ`, `PLL_SYS_POSTDIV1` and `PLL_SYS_POSTDIV2` are used to configure the system clock PLL during runtime initialization. These are defaulted for you if `SYS_CLK_KHZ=125000`, `XOSC_KHZ=12000` and `PLL_COMMON_REFDIV=1`. You can modify these values in your `CMakeLists.txt` if you want to configure a different system clock during runtime initialization, or are using a non-standard crystal.
  - The new defines `PLL_USB_VCO_FREQ_KHZ`, `PLL_USB_POSTDIV1` and `PLL_USB_POSTDIV2` are used to configure the USB clock PLL during runtime initialization. These are defaulted for you if `USB_CLK_KHZ=48000`, `XOSC_KHZ=12000` and `PLL_COMMON_REFDIV=1`. You can modify these values in your `CMakeLists.txt` if you want to configure a different USB clock if you are using a non-standard crystal.
  - The new define `PICO_PLL_VCO_MIN_FREQ_KHZ` is used in preference to the pre-existing `PICO_PLL_VCO_MIN_FREQ_MHZ`, though specifying either is supported.
  - The new define `PICO_PLL_VCO_MAX_FREQ_KHZ` is used in preference to the pre-existing `PICO_PLL_VCO_MAX_FREQ_MHZ`, though specifying either is supported.

New Libraries

pico_flash

- This is a new higher level library than `hardware_flash`. It provides helper functions to facilitate getting into a state where it is safe to write to flash (the default implementation disables interrupts on the current core, and if necessary, makes sure the other core is running from RAM, and has interrupts disabled).
- Adds a `flash_safe_execute()` function to execute a callback function while in the "safe" state.
- Adds a `flash_safe_execute_core_init()` function which must be called from the "other core" when using `pico_multicore` to enable the cooperative support for entering a "safe" state.
- Supports user override of the mechanism by overriding the `get_flash_safety_helper()` function.

Miscellaneous

- All assembly (including inline) in the SDK now uses the `unified` syntax.
  - New C macros `pico_default_asm( … )` and `pico_default_asm_volatile( … )` are provided that are equivalent to `asm andasm volatile` blocks, but with a `.syntax unified` at the beginning.
- A new assembler macro `pico_default_asm_setup` is provided to configure the correct CPU and dialect.
- Some code cleanup to make the SDK code at least compile cleanly on Clang and IAR.

Build

- `PICO_BOARD` and `PICO_BOARD_HEADER_DIRS` now correctly use the latest environment variable value if present.
- A CMake performance regression due to repeated calls to `find_package` has been fixed.
- Experimental support is provided for compiling with Clang. As an example, you can build with the LLVM Embedded Toolchain for Arm, noting however that currently only version 14.0.0 works, as later versions use `picolib` rather than `newlib`.
  - Note that if you are using TinyUSB you need to use the latest master to compile with Clang.

```
$ mkdir clang_build
$ cd clang_build
$ cmake -DPICO_COMPILER=pico_arm_clang -DPICO_TOOLCHAIN_PATH=/path/to/arm-embedded-llvm
-14.0.0 ..
$ make
```

Bluetooth Support for Pico W

The support is now official. Please find examples in pico-examples.

- The Bluetooth API is provided by BTstack.
- The following libraries are provided that expose core BTstack functionality:
  - `pico_btstack_ble` - Adds Bluetooth Low Energy (LE) support.
  - `pico_btstack_classic` - Adds Bluetooth Classic support.
  - `pico_btstack_sbc_encoder` - Adds Bluetooth Sub Band Coding (SBC) encoder support.
  - `pico_btstack_sbc_decoder` - Adds Bluetooth Sub Band Coding (SBC) decoder support.
  - `pico_btstack_bnep_lwip` - Adds Bluetooth Network Encapsulation Protocol (BNEP) support using LwIP.
  - `pico_btstack_bnep_lwip_sys_freertos` - Adds Bluetooth Network Encapsulation Protocol (BNEP) support using LwIP with FreeRTOS for `NO_SYS=0`.
- The following integration libraries are also provided:
  - `pico_btstack_run_loop_async_context` - provides a common `async_context` backed implementation of a BTstack *"run loop"* that can be used for all BTstack use with the `pico-sdk`.
  - `pico_btstack_flash_bank` - provides a sample implementation for storing required Bluetooth state in flash.
  - `pico_btstack_cyw43` - integrates BTstack with the CYW43 driver.
- The `CMake` function `pico_btstack_make_gatt_header` can be used to run the BTstack `compile_gatt` tool to make a GATT header file from a BTstack `GATT` file.

- pico_cyw43_driver and cyw43_driver now support HCI communication for Bluetooth.
- cyw43_driver_picow now supports Pico W specific HCI communication for Bluetooth over SPI.
- cyw43_arch_init() and cyw43_arch_deinit() automatically handle Bluetooth support if CYW43_ENABLE_BLUETOOTH is 1 (as it will be automatically if you depend on pico_btstack_cyw43).

Key changes since 1.5.0:

- Added Raspberry Pi specific BTstack license.
- The storage offset in flash for pico_btstack_flash_bank can be specified at runtime by defining pico_flash_bank_get_storage_offset_func to your own function to return the offset within flash.
- pico_btstack_flash_bank is now safe for multicore / FreeRTOS SMP use, as it uses the new pico_flash library to make sure the other core is not accessing flash during flash updates. If you are using pico_multicore you must have called flash_safe_execute_core_init from the "other" core (to the one Bluetooth is running on).
- Automatically set Bluetooth MAC address to the correct MAC address (Wi-Fi MAC address + 1), as some devices do not have it set in OTP and were using the same default MAC from the Bluetooth chip causing collisions.
- Various bug-fixes and stability improvements (especially with concurrent Wi-Fi), including updating cyw43_driver and btstack to the newest versions.

Release 2.0.0 (08 August 2024)

This is a major release which adds support for the new RP2350 and for compiling RISC-V code in addition to Arm.

- There is a lot of new functionality in the RP2350 microcontroller, it is recommended that you read the RP2350 Datasheet
- There is a lot of new functionality in the SDK, it is also worth reading the Raspberry Pi Pico-series C/C++ SDK book. This also includes documentation for RP2040 and RP2350 APIs, along with much more complete documentation of SDK #defines and CMake build variables.

Notices

IMPORTANT

You should delete/recreate all build directories when upgrading from previous versions of the Raspberry Pi Pico SDK.

Major New Features

Support for RP2350

Many programs you have written for RP2040 (say a Raspberry Pi Pico) should work unmodified on RP2350 (say a Raspberry Pi Pico 2) even when compiled for RISC-V.

- You can now specify rp2350-arm-s (Arm Secure) or rp2350-riscv (RISC-V) as well as the previous rp2040 (default) and host.
- Setting PICO_BOARD=some_board will now set PICO_PLATFORM if one is specified in some_board.h since most boards either use exclusively RP2040 or RP2350.
- PICO_PLATFORM also supports rp2350 but this gets replaced with the value PICO_DEFAULT_RP2350_PLATFORM which you can set in your environment or CMakeLists.txt. Many of the boards for RP2350 - including pico2- select rp2350 as the PICO_BOARD to honour your preference.

> **NOTE**
>
> This release of the SDK does not support writing Arm Non-Secure binaries to run under the wing of an Arm Secure binary. This support will be added in a subsequent release.

Security and Code Signing

- The RP2350 bootrom contains support for signed images and a variety of other security features. The SDK supports building signed images etc. as part of the CMake build. For further information, please read RP2350 Datasheet "Bootrom Concepts" section, and also the Raspberry Pi Pico-series C/C++ SDK book for details on configuring your build to sign code. Note that signed code is only applicable to chips that have been locked down for security, but you can also hash your image for integrity checking.

Board Support

The following boards have been added and may be specified via `PICO_BOARD`:

- `defcon32_badge`
- `gen4_rp2350_24`
- `gen4_rp2350_24ct`
- `gen4_rp2350_24t`
- `gen4_rp2350_28`
- `gen4_rp2350_28ct`
- `gen4_rp2350_28t`
- `gen4_rp2350_32`
- `gen4_rp2350_32ct`
- `gen4_rp2350_32t`
- `gen4_rp2350_35`
- `gen4_rp2350_35ct`
- `gen4_rp2350_35t`
- `hellbender_2350A_devboard`
- `ilabs_challenger_rp2350_bconnect`
- `ilabs_challenger_rp2350_wifi_ble`
- `melopero_perpetuo_rp2350_lora`
- `phyx_rick_tny_rp2350`
- `pico2`
- `pimoroni_pga2350`
- `pimoroni_pico_plus2_rp2350`
- `pimoroni_plasma2350`
- `pimoroni_tiny2350`
- `seeed_xiao_rp2350`
- `solderparty_rp2350_stamp`

- `solderparty_rp2350_stamp_xl`
- `sparkfun_promicro_rp2350`
- `switchscience_picossci2_conta_base`
- `switchscience_picossci2_dev_board`
- `switchscience_picossci2_micro`
- `switchscience_picossci2_rp2350_breakout`
- `switchscience_picossci2_tiny`
- `tinycircuits_thumby_color_rp2350`

New Libraries

`hardware_boot_lock` (RP2350)

- New library for accessing the BOOT locks from secure code.

`hardware_dcp` (RP2350 Arm)

- Contains assembler macros for individual DCP (Double Co-Processor) instructions
- Contains assembler macros for canned instruction sequences for higher-level operations
- `HAS_DOUBLE_COPROCESSOR` define indicates hardware support

`hardware_hazard3` (RP2350 RISC-V)

- Assembler macros and inline functions for accessing Hazard3 extensions

`hardware_powman` (RP2350)

- Hardware APIs for the Power Management hardware.
- `HAS_POWMAN_TIMER` define indicates hardware support.

`hardware_rcp` (RP2350 Arm)

- Contains inline functions and assembler macros for the RCP (Redundancy Co-Processor) instructions.
- `HAS_REDUNDANCY_COPROCESSOR` define indicates hardware support.

`hardware_riscv_platform_timer` (RP2350)

- Hardware APIs for the RISC-V Platform Timer (which is also made available on Arm).

`hardware_sha256` (RP2350)

- Hardware APIs for the SHA256 hashing hardware.

hardware_ticks

- Hardware APIs for the RP2350 tick generators.
- On RP2040 the same API is used, but only one tick generator TICK_WATCHDOG is used, which is backed by the hardware in the RP2040 WatchDog hardware.

pico_aon_timer

- Abstraction for a hardware timer that is "Always-On", and can wake the processor up even from a low power state at a given time.
  - On RP2040 this uses the RTC.
  - On RP2350 this uses the Powman Timer.

pico_atomic

- Additional support for C11 atomic functions using spin lock number PICO_SPINLOCK_ID_ATOMIC.
  - On RP2040, all functions are implemented via spinlock.
  - On RP2350, only 64-bit or arbitrary-sized atomics are implemented via spin lock; the rest use processor exclusive/atomic instructions.
    - ACTLR.EXTEXCLALL must be set to 1 on each processor for the exclusive instructions to work. This is done automatically in the SDK by one of the per-core initializers in pico_runtime_init.
- Included by pico_runtime by default.

pico_boot_lock (RP2350)

- Support for acquiring and releasing locks to prevent concurrent use of hardware resources used by bootrom functions.
- Enabled via PICO_BOOTROM_LOCKING_ENABLED which defaults to 1 on RP2350.
- Some bootrom functions use shared resources such as the single SHA256 or put hardware such as the OTP or XIP interface into a state that cannot execute concurrently with certain other code. The bootrom supports checking that the resource is owned, and this library turns that checking on.
- The bootrom function wrappers in pico_bootrom call the functions in pico_boot_lock around affected bootrom functions, and thus will take and release locks if PICO_BOOTROM_LOCKING_ENABLED=1.
- NUM_BOOT_LOCKS define indicates the number of boot locks (8 on 'RP2350', 0 on 'RP2040').

pico_clib_interface

- New library to encapsulate the interface between the SDK and the C library.
- Supports
  - newlib (full).
  - picolibc (preview).
  - llvm-libc (preview).
- Included by pico_runtime by default.

`pico_crt0`

- New library split out of `pico_standard_link` to encapsulate the earliest startup code before the runtime initialisation, and shutdown code after the runtime.
- Repository for the default RP2040 and RP2350 linker scripts.
    - The flash size specified in the board header is now used when linking which is handy if you have >2MB of flash and >2MB of code/data.
    - **Note:** The linker scripts have changed since the previous release of the SDK. If you have custom linker scripts, it is recommended that you update them to match.
        - In particular the new linker scripts include an "embedded block" which is required for a binary to boot on RP2350.
        - `__HeapLimit` is now defined to be the end of RAM rather than the end of a `PICO_HEAP_SIZE` chunk, to better match the standard behaviour. `PICO_HEAP_SIZE` is the minimum heap size required, and space is required for it at link time. `sbrk` in the previous SDK ignored it anyway and used the end of RAM so there is no functional change there.
- Included by `pico_runtime` by default

`pico_cxx_options`

- New library split out of `pico_standard_link` to configure C++ options.
- Included by `pico_standard_link` by default.

`pico_platform_compiler`

- New library split out of `pico_platform` with the functions/macros related to the compiler.
- Included by `pico_platform` by default.

`pico_platform_panic`

- New library split out of `pico_platform` with the panic function implementation.
- Included by `pico_platform` by default.

`pico_platform_sections`

- New library split out of `pico_platform` with the section macros such as `__not_in_flash_func`.
- Included by `pico_platform` by default.

`pico_runtime_init`

- Contains the standard initialisers that should get run before main, or per core.
- Unlike in the previous SDK version where `runtime_init()` was a monolithic function which also called some `__preinit_array` initialisers, the new runtime_init library:
    - Separates each initialiser out individually, for say initialiser "foo".
        - Defines `PICO_RUNTIME_INIT_FOO` which is a "12345" *line number* ordering of the initialiser with respect to others.
        - Declares `runtime_init_foo()` which is the actual initialiser.

    - If PICO_RUNTIME_SKIP_INIT_FOO is not set, it adds the initialiser entry to call runtime_init_foo() before main (or per core initialisation).
    - If PICO_RUNTIME_NO_INIT_FOO is not set, it adds the (weak) implementation of runtime_init_foo().
    - This gives the user full control to customise runtime initialisation, either skipping or replacing parts.
- Included by pico_runtime by default.

pico_sha256

- High level APIs for generating SHA256 hashes both synchronously and asynchronously

pico_standard_binary_info

- New library split out of pico_standard_link that adds the "common" binary info items to the binary.
- Included by pico_standard_link by default.

Library Changes / Improvements

Note that all hardware libraries now support the increased number of GPIOs on RP2350B in APIs that take a GPIO number; this is not noted for every library.

pico_base

- More error return codes were added to pico/error.h, mostly because these are the same values returned by RP2350 bootrom API functions, but also a number of new SDK APIs also return meaningful errors.
- In pico/types.h, by popular demand, absolute_time_t now always defaults to uint64_t regardless of the type of build. You can set PICO_OPAQUE_ABSOLUTE_TIME_T=1 to make it a struct in all build types.

pico_binary_info

- Now supports > 32 GPIO pins when PICO_BINARY_INFO_USE_PINS_64=1 - this is defaulted for you based on the number of GPIOs on the board.

hardware_adc

- PARAM_ASSERTIONS_ENABLED_ADC is renamed to PARAM_ASSERTIONS_ENABLED_HARDWARE_ADC - the old define is still supported as a fallback.
- ADC_TEMPERATURE_CHANNEL_NUM added since this value varies between RP2040 and RP2350.

hardware_clocks

- set_sys_clock_ functions are now in hardware/clocks.h.
- Clock configuration.
    - PLL_COMMON_REFDIV is deprecated in favour of PLL_SYS_REFDIV and PLL_USB_REFDIV.
    - PLL_SYS_VCO_FREQ_HZ is new and preferred over PLL_SYS_VCO_FREQ_KHZ.
    - PLL_USB_VCO_FREQ_HZ is new and preferred over PLL_USB_VCO_FREQ_KHZ.
    - XOSC_HZ, SYS_CLK_HZ, USB_CLK_HZ now added, and take preference over the still supported XOSC_KHZ, SYS_CLK_KHZ, and

USB_CLK_KHZ.
  - set_sys_clock_hz() and check_sys_clock_hz() added.
  - clock_configure_undivided() and clock_configure_int_divider() for no divisor or a whole integer divider as the code doesn't require 64-bit arithmetic and thus saves space.
  - The enum clock_index no longer exists and has been replaced with clock_num_t. However, all clock functions now take clock_handle_t to allow for future enhancement. This is currently just an alias for clock_num_t
  - vcocalc.py can now be used to generate the CMake configuration for a particular clock setting.
  - The default system clock on RP2350 is 150 MHz.

hardware_divider

- Since the RP2350 processors have efficient divider instructions, RP2350 has no SIO HW Divider. Software versions of the hardware_divider functions are provided for RP2350.
- HAS_SIO_DIVIDER define is now provided for you.

hardware_dma

- PARAM_ASSERTIONS_ENABLED_DMA is renamed to PARAM_ASSERTIONS_ENABLED_HARDWARE_DMA - the old define is still supported as a fallback.
- Added dma_get_irq_num()`function and `DMA_IRQ_NUM() macro to return the process IRQ Number for the *n* th DMA IRQ.
- NUM_DMA_IRQS define is provided for you.
  - it is 2 on RP2040 and 4 on RP2350.

hardware_exception

- PARAM_ASSERTIONS_ENABLED_EXCEPTION is renamed to PARAM_ASSERTIONS_ENABLED_HARDWARE_EXCEPTION - the old define is still supported as a fallback.
- Added RISC-V support.
  - exception numbers are processor exception cause numbers.
- exeception_[get|set]_priority() are added for Arm.

hardware_flash

- PARAM_ASSERTIONS_ENABLED_FLASH is renamed to PARAM_ASSERTIONS_ENABLED_HARDWARE_FLASH - the old define is still supported as a fallback.
- flash_flush_cache() is added.

hardware_gpio

- PARAM_ASSERTIONS_ENABLED_GPIO is renamed to PARAM_ASSERTIONS_ENABLED_HARDWARE_GPIO - the old define is still supported as a fallback.
- The enum gpio_function no longer exists and has been replaced with gpio_function_t.
- gpio_xxx_masked() functions now have a gpio_xxx_masked64() variant that takes a 64-bit mask of GPIO indexes.
- gpio_xxx_mask() functions now have a gpio_xxx_mask64() variant that takes a 64-bit mask of GPIO indexes.
- gpio_get_all64() added to read the state of >32 pins.

- gpio_put_all64() added to write the state of >32 pins.
- On Arm RP2350 GPIO Co-Processor instructions are used by default. This is controlled via PICO_USE_GPIO_COPROCESSOR.
- HAS_GPIO_COPROCESSOR define indicates hardware support.

hardware_i2c

- PARAM_ASSERTIONS_ENABLED_I2C is renamed to PARAM_ASSERTIONS_ENABLED_HARDWARE_I2C - the old define is still supported as a fallback.
- PICO_DEFAULT_I2C_INSTANCE() macro added which is equivalent to the pre-existing i2c_default
- Added I2C_NUM(), I2C_INSTANCE(), I2C_DREQ_NUM() macros to abstract differences between platforms.
- Fixed per-character timeouts.

hardware_interp

- PARAM_ASSERTIONS_ENABLED_INTERP is renamed to PARAM_ASSERTIONS_ENABLED_HARDWARE_INTERP - the old define is still supported as a fallback.

hardware_irq

- PARAM_ASSERTIONS_ENABLED_IRQ is renamed to PARAM_ASSERTIONS_ENABLED_HARDWARE_IRQ - the old define is still supported as a fallback.
- irq_xxx_mask_xxx() functions now have a gpio_xxx_mask_n_xxx() variant that affects the *n* th set of 32 IRQs
- Expose runtime_init_per_core_irq_priorities() function
- Added irq_set_riscv_vector_handler() function to replace code entries in the machine vector table.

hardware_pio

- PARAM_ASSERTIONS_ENABLED_PIO is renamed to PARAM_ASSERTIONS_ENABLED_HARDWARE_PIO - the old define is still supported as a fallback.
- PICO_PIO_VERSION is used to determine whether new RP2350 functionality (PICO_PIO_VERSION=1) is supported. This is defaulted based on the platform.
- PICO_PIO_USE_GPIO_BASE is used to determine whether support is enabled for GPIOs above 32. TThe default value is set based on the chip package.
- Added pio_sm_set_jmp_pin().
- Added pio_claim_free_sm_and_add_program(), pio_claim_free_sm_and_add_program_for_gpio_range() and `pio_remove_program_and_unclaim_sm() to simplify finding and claiming a free PIO instance and state machine and installing programs.
- Added `pio_get_irq_num()`function to return the process IRQ Number for the *n* th PIO IRQ for a PIO instance.
- Added PIO_NUM(), PIO_INSTANCE(), PIO_IRQ_NUM(), PIO_DREQ_NUM() and PIO_FUNCSEL_NUM() macros to abstract differences between platforms.
- Added sm_config_set_out_pin_base() and sm_config_set_out_pin_count().
- Added sm_config_set_in_pin_base() and sm_config_set_in_pin_count(). Note the latter is only meaningful on PICO_PIO_VERSION=1 which supports a limit.
- Added sm_config_set_set_pin_base() and sm_config_set_set_pin_count().

- Added `sm_config_set_sideset_pin_base()` and `sm_config_set_sideset_pin_count()`.
- For `PICO_PICO_VERSION=1` i.e. RP2350:
  - Added `pio_set_gpio_base()` and `pio_get_gpio_base()` to assign the PIO instance to pins 0-31 or 16-47.
  - Added `pio_set_sm_multi_mask_enabled()`.
  - Added `pio_clkdiv_restart_sm_multi_mask()`.
  - Added `pio_enable_sm_multi_mask_in_sync()`.
- `NUM_PIO_IRQS` define is now provided for you (2 on both RP2040 and RP2350).

hardware_pll

- `PICO_PLL_VCO_MIN_FREQ_HZ` is new and now preferred to `PICO_PLL_VCO_MIN_FREQ_KHZ` or `PICO_PLL_VCO_MIN_FREQ_MHZ`.
- `PICO_PLL_VCO_MAX_FREQ_HZ` is new and now preferred to `PICO_PLL_VCO_MAX_FREQ_KHZ` or `PICO_PLL_VCO_MAX_FREQ_MHZ`.
- `PLL_RESET_NUM()` macro added to abstract differences between platforms.

hardware_pwm

- `PARAM_ASSERTIONS_ENABLED_PWM` is renamed to `PARAM_ASSERTIONS_ENABLED_HARDWARE_PWM` - the old define is still supported as a fallback.
- `PICO_DEFAULT_PWM_INSTANCE()` macro added which is equivalent to the pre-existing `pwm_default`.
- Added `PWM_SLICE_NUM()` and `PWM_DREQ_NUM()` macros to abstract differences between platforms.
- Added `PWM_DEFAULT_IRQ_NUM()` since RP2350 supports 2 PWM IRQs to indicate which IRQ the pre-existing RP2040 functions use.
- Added `pwm_set_irq0_enabled()`, `pwm_set_irq1_enabled()` and `pwm_irqn_set_slice_enabled()` to differentiate between the IRQs.
- Added `pwm_set_irq0_mask_enabled()`, `pwm_set_irq1_mask_enabled()` and `pwm_irqn_set_mask_enabled()` to differentiate between the IRQs.
- Added `pwm_get_irq0_status_mask()`, `pwm_get_irq1_status_mask()` and `pwm_irqn_get_status_mask()` to differentiate between the IRQs.
- Added `pwm_pwm_force_irq0()`, `pwm_force_irq1()` and `pwm_irqn_force()` to differentiate between the IRQs.

hardware_resets

- `PARAM_ASSERTIONS_ENABLED_RESETS` is renamed to `PARAM_ASSERTIONS_ENABLED_HARDWARE_RESETS` - the old define is still supported as a fallback.
- `reset_block()` is renamed to `reset_block_mask()` but the old name is still supported.
- `unreset_block()` is renamed to `unreset_block_mask()` but the old name is still supported.
- `unreset_block_wait()` is renamed to `unreset_block_mask_wait_blocking()` but the old name is still supported.
- `reset_block_num()`, `unreset_block_num()`, `unreset_block_num_wait_blocking()` and `reset_unreset_block_num_wait_blocking()` added to reset or unreset a single block by `reset_num_t` index.

hardware_rtc

- Note this library is only available on RP2040, since the RP2350 lacks the RTC hardware.
- There is a similar always-on timer in `hardware_powman`.

* A common API for both RP2040 and RP2350 is provided in `pico_aon_timer`.
* `HAS_RP2040_RTC` define is now provided for you.

`hardware_spi`

* `PARAM_ASSERTIONS_ENABLED_SPI` is renamed to `PARAM_ASSERTIONS_ENABLED_HARDWARE_SPI` - the old define is still supported as a fallback.
* `PICO_DEFAULT_SPI_INSTANCE()` macro added which is equivalent to the pre-existing `spi_default`.
* Added `SPI_NUM()`, `SPI_INSTANCE()`, `SPI_DREQ_NUM()` macros to abstract differences between platforms.
* Fixed per-character timeouts.

`hardware_sync`

* `restore_interrupts_from_disabled()` is added as a variant for `restore_interrupts()` which **must** be paired with a matching `save_and_disable_interrupts()`. This is the common usage and produces smaller/faster code on RISC-V.
* Spinlock functionality has been delegated to a separate hardware_sync_spinlock library, which is included for you.
* hardware_sync_spin_lock.
    * Whilst RP2350 has the same SIO spin locks as RP2040, due to Errata RP2350-E2, these are not used by default.
    * Instead, a software implementation using atomic instructions is used.
    * You can set `PICO_USE_SW_SPIN_LOCKS=0` to disable this if you know you aren't affected by RP2350-E2 and want to use the h/w spin locks instead.
    * Added `spin_try_lock_unsafe()` function.

`hardware_timer`

* `PARAM_ASSERTIONS_ENABLED_TIMER` is renamed to `PARAM_ASSERTIONS_ENABLED_HARDWARE_TIMER` - the old define is still supported as a fallback.
* RP2350 supports two timer instances.
    * `PICO_DEFAULT_TIMER_INSTANCE()` macro added based on `PICO_DEFAULT_TIMER` (0 on RP2040, 0/1 on RP2350).
    * Added `TIMER_NUM()`, `TIMER_INSTANCE()`, `TIMER_ALARM_NUM_FROM_IRQ()` and `TIMER_ALARM_NUM_FROM_IRQ()` macros to abstract differences between platforms
    * Added `hardware_alarm_get_irq_num()` to get the processor IRQ number for a particular alarm on a timer.
    * New versions of all functions added with a `timer_` prefix and a timer instance passed as the first argument. The pre-existing functions call these with the default timer instance.
* `NUM_TIMERS` has been renamed to `NUM_ALARMS` as that's what it was (4).
* `NUM_GENERIC_TIMERS` has been added which is 1 on RP2040 and 2 on RP2350.

`hardware_uart`

* `PARAM_ASSERTIONS_ENABLED_UART` is renamed to `PARAM_ASSERTIONS_ENABLED_HARDWARE_UART` - the old define is still supported as a fallback.
* `PICO_DEFAULT_UART_INSTANCE()` macro added which is equivalent to the pre-existing `uart_default`.
* Added `UART_NUM()`, `UART_INSTANCE()`, `UART_DREQ_NUM()`, `UART_IRQ_NUM()`, `UART_CLOCK_NUM()`, `UART_RESET_NUM()`, `UART_FUNCSEL_NUM()` macros to abstract differences between platforms.

- `uart_set_irq_enables()` is renamed to `uart_set_irqs_enabled()` but the old name is still supported.
- `uart_get_dreq()` is renamed to `uart_get_dreq_num()` but the old name is still supported.
- `uart_get_reset_num()` is added.
- Incorrect baud setting for certain frequencies was fixed.

hardware_vreg

- `vreg_disable_voltage_limit()` added to allow full range of DVDD voltage selection on RP2350

hardware_watchdog

- `PARAM_ASSERTIONS_ENABLED_WATCHDOG` is renamed to `PARAM_ASSERTIONS_ENABLED_HARDWARE_WATCHDOG` - the old define is still supported as a fallback.
- Added `watchdog_disable()`.
- `watchdog_get_count()` is renamed to `watchdog_get_time_remaining_ms()` but the old name is still supported.

hardware_xosc

- `XOSC_HZ` is new and now preferred to `XOSC_KHZ`.

hardware_regs

- `enum irq_num_[rp2040|rp2350]` (typedef-ed as `irq_num_t`) added with the constants from `inctrl.h`. Note these remain as `#defines` when included from assembly.
- `enum dreq_num_[rp2040|rp2350]` (typedef-ed as `dreq_num_t`) added with the constants from `dreq.h`. Note these remain as `#defines` when included from assembly.

hardware_structs

- `enum bus_ctrl_perf_counter_[rp2040|rp2350]` (typedef-ed as `bus_ctrl_perf_counter_t`) added.
  - **Note** enum bus_ctrl_per_counter no longer exists.
- `enum clock_num_[rp2040|rp2350]` (typedef-ed as `clock_num_t`) added.
  - **Note** enum clock_index no longer exists.
- `enum clock_dest_num_[rp2040|rp2350]` (typedef-ed as `clock_dest_num_t`) added.
- `enum gpio_function_[rp2040|rp2350]` (typedef-ed as `gpio_function_t`) added.
  - **Note** enum gpio_function no longer exists.
- `enum gpio_function1_[rp2040|rp2350]` (typedef-ed as `gpio_function1_t`) added (for QSPI bank).
- `enum reset_num_[rp2040|rp2350]` (typedef-ed as `reset_num_t`) added.
- `enum tick_gen_num_rp2350` (typedef-ed as `reset_num_t`) added.
- Various naming consistencies have been fixed.
  - `iobank0.h` → `io_bank00.h`, `iobank0_hw` → `io_bank0_hw` - shims are provided for the old versions.
  - `ioqspi.h` → `io_qspi0.h`, `ioqspi_hw` → `io_qspi_hw` - shims are provided for the old versions.
  - `padsbank0.h` → `pads_bank0.h`, `padsbank0_hw` → `pads_bank0_hw` - shims are provided for the old versions.

  - padsqspi.h → pads_qspi.h, padsqspi_hw → pads_qspi_hw - shims are provided for the old versions.
  - bus_ctrl.h → busctrl.h, bus_ctrl_hw → busctrl_hw (don't ask! but hardware_struct headers now match hardware_regs names at least!).

boot_stage2

- There are now separate implementations for RP2040 and RP2350.
- A boot_stage2 is not needed on RP2350, but one can be included via the define PICO_EMBED_XIP_SETUP=1.

cmsis

- CMSIS headers are updated to CMSIS 6.1
- Device headers RP2040.h and RP2350.h are generated, and now include basic hardware structures as per the latest SVDConv defaults.

pico_bootrom

- New RP2350 bootrom APIs added.
- rom_xxx() inline function wrappers added for all xxx() ROM functions.
- Additional rom_get_boot_random() and rom_add_flash_runtime_partition() for RP2350 which use underlying bootrom functionality but aren't just wrapper functions.

pico_bt_stack

- BTStack updated to 1.6.1 from 1.5.6
  - Lots of additions, fixes and changes, for the full list see the change log

pico_cyw43_arch

- PARAM_ASSERTIONS_ENABLED_CYW43_ARCH is renamed to PARAM_ASSERTIONS_ENABLED_PICO_CYW43_ARCH - the old define is still supported as a fallback.
- lib/cyw43-driver has been updated to the latest version
  - Mostly bug fixes.
  - Adds WPA3 support for Pico W. To use this, use CYW43_AUTH_WPA3_SAE_AES_PSK or CYW43_AUTH_WPA3_WPA2_AES_PSK instead of CYW43_AUTH_WPA2_AES_PSK when connecting to wifi with cyw43_arch_wifi_connect_timeout_ms or cyw43_arch_enable_ap_mode.

pico_cyw43_driver

- cyw43_driver updated to commit faf36381.
- Added support for changing the clock speed of the SPI connection to the Wi-Fi chip. See CYW43_PIO_CLOCK_DIV_INT, CYW43_PIO_CLOCK_DIV_FRAC and CYW43_PIO_CLOCK_DIV_DYNAMIC.

pico_divider

- Functions that returned a quotient and divider in a uint64_t or int64_t now return a divmod_result_t - the signed-ness of the value before was meaningless anyway, and the compiler will still return it as a 64-bit value.

- Extra functions in `pico/divider.h` now implemented for `pico_set_divider_implemtation(compiler)` as well as for RP2350 which has no RP2040 hardware divider.

`pico_double`

- `pico_set_double_implementation(pico)` (the default) now uses the Double Co-Processor (DCP) for double-precision floating-point arithmetic on Arm RP2350, and highly optimised Arm VFP implementations of the double-precision scientific functions, for much improved performance over the C library versions.
- Extra functions exposed from `pico` implementation
  - `int2double()`
  - `uint2double()`
  - `int642double()`
  - `uint642double()`
  - `double2uint()`
  - `double2uint64()`
- Extra functions exposed from `pico` implementation for Arm RP2350 only
  - `ddiv_fast()`
  - `sqrt_fast()`
  - `mla()`

`pico_float`

- `pico_set_float_implementation(pico)` (the default) now uses the compiler for single-precision floating point arithmetic on Arm RP2350 since the processor has VFP instructions, but includes custom optimised scientific functions also using the VFP.
- `pico_set_diouble_implementation(pico_dcp)` uses the Double Co-Processor (DCP) for single-precision floating point arithmetic on Arm RP2350, and highly optimised Arm M33 implementations of the single-precision scientific functions, for much improved performance over the C library versions. This library is intended for those situations where you cannot (or don't want to) use the VFP instructions.
- Extra functions exposed from `pico` implementation.
  - `int2float()`
  - `uint2float()`
  - `int642float()`
  - `uint642float()`
  - `float2uint()`
  - `float2uint64()`
  - `float2uint_z()`
  - `float2uint64_z()`
- Extra functions exposed from `pico` implementation for Arm R2350 only.
  - `float2fix64_z()`
  - `fdiv_fast()`
  - `fsqrt_fast()`

pico_lwip

- Update lib/lwip to 2.2.0
  - There have been some bugs fixed, and some new features were added (most notably full Address Conflict Detection support).

pico_mbedtls

- Update to lib/mbedtls to 2.28.8 from 2.28.1
  - This release of Mbed TLS provides bug fixes and minor enhancements. This release includes fixes for security issues.
- Added support for hardware SHA256 calculation on RP2350
  - To use this in mbedtls you need to define MBEDTLS_SHA256_ALT in your mbedtls_config.h. Use LIB_PICO_SHA256 to check if hardware SHA256 is supported and fallback to defining MBEDTLS_SHA256_C for the software SHA256 calculation.

pico_multicore

- Added SIO_FIFO_IRQ_NUM() to get the IRQ number for the FIFO IRQ on a particular core, since RP2040 and RP2350 are different.
  - **note** that RP2350 uses the same IRQ number on both cores, so if you have IRQ handlers for both cores, you should share the same function and check the core number in the IRQ handler. This strategy of course works on RP2040 too.
- Added multicore_fifo_push_blocking_inline() and multicore_fifo_pop_blocking_inline().
- Added multicore_doorbell_ functions for the new inter-core Doorbells on RP2350.
  - NUM_DOORBELLS is provided which is 8 on RP2350, 0 on RP2040.

pico_rand

- Added the hardware TRNG as an additional entropy source on RP2350.
  - HAS_RP2350_TRNG indicates hardware support.
- Many, but not all, of the pre-existing entropy sources are disabled on RP2350 in favour of using the TRNG.

pico_runtime

- A shadow of its former self, it now just:
  - aggregates other default libraries required for getting to main() and having the C runtime work.
  - provides low level runtime_run_initializers() and runtime_run_per_core_initializers() which run initializers from the __preinit_array.
- The runtime_init() entrypoint has moved to pico_clib_interface.

pico_standard_link

- Much previously included functionality has been split out into pico_crt0, pico_cxx_options and pico_standard_binary_info.
- What remains is entirely focused on setting up the linker configuration.

- **Finally** fixed a bug where changes to the linker script did not cause a relink.

pico_stdio

- Some internal reorganisation to separate functionality between here and `pico_clib_interface`.
- Added `PICO_STDIO_SHORT_CIRCUIT_CLIB_FUNCS` to control whether `printf`, `vprintf`, `puts`, `putchar` and `getchar` go thru the C library (thus usually pulling in all the FILE handling APIs resulting in huge bloat - but more sensible behaviour when mixing say `printf` with `fprintf(stdout` etc.) This defaults to 0, i.e. "do short-circuit the c lib" which was the behaviour in the previous SDK version.
- Add support for Segger RTT stdio.
- Implemented `stdio_flush()` for UART and USB CDC.
- Added `stdio_deinit_all()` and individual `stdio_deinit_xxx` functions.

pico_stdio_usb

- Now supports MS OS2 descriptors by default. See `PICO_STDIO_USB_RESET_INTERFACE_SUPPORT_MS_OS_20_DESCRIPTOR`.
- `PICO_STDIO_USB_ENABLE_RESET_VIA_VENDOR_INTERFACE` and `PICO_STDIO_USB_ENABLE_RESET_VIA_BAUD_RATE` are now both supported even if the user is using `tinyusb_device` directly themselves.
- Bug that could cause deadlock with FreeRTOS SMP and printing from IRQs fixed.

pico_stdlib

- `pico/stdlib.h` no longer declares `set_sys_clock_` functions. You must include `hardware/clocks.h` explicitly.

pico_time

- `remaining_alarm_time_ms()`, `remaining_alarm_time_us()`, `alarm_pool_remaining_alarm_time_ms()` and `alarm_pool_remaining_alarm_time_us()` were added.
- Implementation of alarm pools completely rewritten for much lower overhead, jitter and higher throughput in the majority of cases. The pairing heap has been replaced with a linked list which is faster and uses less memory in most normal use cases too.

> **IMPORTANT**
>
> `fire_if_past` now always fires asynchronously in the same way as a normal timeout (rather than being called synchronously during the call). Thus `alarm_pool_add_alarm_at_force_in_context` is now no different to `alarm_pool_add_alarm_at`.

- New `pico_timer_adapter` abstraction added so `pico_time` could be backed by other types of timer hardware in the future, and so `pico_time` no longer depends directly on a `hardware_timer` abstraction which simplifies `PICO_PLATFORM=host`.
- Support for two hardware timer blocks on RP2350.
  - `alarm_pool_timer_t` abstraction added to represent the time "counter" backing the alarm pool.
  - `alarm_pool_t` now has an associated `alarm_pool_timer_t` instance.
  - `alarm_pool_create_on_timer()` is added to create an alarm pool on a specific alarm pool timer.
  - `alarm_pool_get_default_timer()` is added which is used when not explicitly passing an alarm pool timer. `PICO_DEFAULT_TIMER` selects which timer instance is the default (0 on `RP2040`, 0/1 on `RP2350`).

- PARAM_ASSERTIONS_ENABLED_TIME is renamed to PARAM_ASSERTIONS_ENABLED_PICO_TIME - the old define is still supported as a fallback.
- check_timeout_fn now takes two parameters. This was likely unused outside the pico_time implementation anyway.
- Expose runtime_init_default_alarm_pool() function.

pico_util

- time_to_datetime(), datetime_to_time() and datetime_to_str() functions relating to hardware_rtc are now guarded by PICO_INCLUDE_RTC_DATETIME which defaults to 0 on RP2350, since RP2350 does not include the RP2040 RTC hardware.
- timespec_to_ms(), timespec_to_us(), ms_to_timespec(), and ms_to_timespec() added to convert between C-library high-resolution time offset and millisecond or microsecond precision offsets.
- queue_try_remove(), queue_remove_blocking() and queue_peek_blocking() now support passing NULL as the element out pointer if the caller doesn't care.

tinyusb

- TinyUSB moved from release 0.15.0 to commit 42326428 (0.17.0 WIP)
- Note that bsp/board.h has been renamed by TinyUSB to bsp/board_api.h the SDK adds a re-director header for you for now.
- Support added for RP2350. Requires a custom memcpy implementation in the rp2040 tinusb driver, as unaligned 32 bit access to device memory causes a hard fault on the Cortex M33.
- See the TinyUSB changelog for full details.

pioasm

- pioasm now supports the full RP2350 PIO (PICO_PIO_VERSION=1) instruction set
- Additionally, it supports many new directives. See the RP2350 Datasheet for full details.

NOTE

currently not all output formats support PICO_PIO_VERSION=1 as they are community provided.

FreeRTOS integration

- You should use this repo for the current FreeRTOS-Kernel supporting RP2040 and RP2350: https://github.com/raspberrypi/FreeRTOS-Kernel.
- Dropped legacy support for configNUM_CORES for the correct configNUMBER_OF_CORES, which is 2 for SMP support and 1 for non-SMP support.
- RP2350_ARM_NTZ (non-trust-zone), and RP2350_RISC-V are available as well as an updated RP2040 version; the former two basically give you the same "single privilege/security domain" experience as on RP2040.
- SMP and non-SMP support (along with running FreeRTOS on either core) are available for all.
- A nasty, but rare pre-existing RP2040 deadlock (especially with TinyUSB printf from IRQs) has been fixed on all three versions; If you were setting configSUPPORT_PICO_SYNC_INTEROP=0 as a workaround, you should no longer do so. Generally, if you are using printf (or anything else using SDK locking primitives) then you do really want configSUPPORT_PICO_SYNC_INTEROP=1 for the best concurrency
- FreeRTOS on RISC-V does not currently support IRQ preemption (which is a Hazard3 only feature anyway).

Backwards Incompatibilities

There are a handful of minor backwards incompatibilities, that hopefully should affect very few people:

- boot_picobin library is now called boot_picobin_headers.
- boot picoboot library is now called boot_picoboot_headers.
- boot_uf2 library is now called boot_uf2_headers.
- pico_base library is now called pico_base_headers.
  - pico/error.h - PICO_ERROR_GENERIC is now -1 because there were pre-existing APIs that returned -1 for any error. PICO_ERROR_TIMEOUT is now -2 (they are swapped from their previous values).
- pico_stdlib
  - pico/stdlib.h no longer declares set_sys_clock_ functions. You must include hardware/clocks.h explicitly.
- pico_time
  - check_timeout_fn now takes two parameters. This was likely unused outside the pico_time implementation anyway.
  - fire_if_past now always fires asynchronously in the same way as a normal timeout (rather than being called synchronously during the call). Thus alarm_pool_add_alarm_at_force_in_context is now no different to alarm_pool_add_alarm_at.
- hardware_clocks
  - The enum clock_index no longer exists and has been replaced with clock_num_t. However, all clock functions now take clock_handle_t to allow for future enhancement. This is currently just an alias for clock_num_t.
- hardware_structs
  - enum bus_ctrl_perf_counter_[rp2040|rp2350] (typedef-ed as bus_ctrl_perf_counter_t) added.
  - **Note** enum bus_ctrl_per_counter no longer exists.
  - enum clock_num_[rp2040|rp2350] (typedef-ed as clock_num_t) added.
  - **Note** enum clock_index no longer exists.
  - enum clock_dest_num_[rp2040|rp2350] (typedef-ed as clock_dest_num_t) added.
  - enum gpio_function_[rp2040|rp2350] (typedef-ed as gpio_function_t) added.
  - **Note** enum gpio_function no longer exists.
- hardware_timer
- NUM_TIMERS has been renamed to NUM_ALARMS as that's what it was (4).

Build

- There are major CMake build changes. If you are maintaining your own non-CMake build, you will have to make extensive changes by looking at the differences yourself.
- All SDK headers are now "system" includes.
- You can now specify rp2350-arm-s (Arm Secure) and rp2350-riscv (RISC-V) as well as the previous rp2040 (default) and host.
- Setting PICO_BOARD=some_board will now set PICO_PLATFORM if one is specified in some_board.h since most boards either use exclusively RP2040 or RP2350.
- PICO_PLATFORM also supports rp2350 but this gets replaced with the value PICO_DEFAULT_RP2350_PLATFORM which you can set in your environment or CMakeLists.txt. Many of the boards for RP2350 - including pico2- select rp2350 as the PICO_BOARD to honour your preference.

- PICO_PLATFORM, PICO_BOARD and other variables will be taken from your environment if not otherwise defined now retain their value after the first CMake invocation. i.e. a pre-existing CMake build configuration directory will not change based on your environment if you re-run cmake.
- PICO_BOARD=pico_w is no longer an odd child out requiring a CMake board file; support for CYW43 Wi-Fi can now be specified in the board header.
- ELF2UF2 is now replaced by use of picotool which will be built as part of your build if not installed on the system. See the picotool GitHub repository for more details on building and installing it locally.
- PICO_GCC_TRIPLE can now be a ';' separated list as well as a single value.
- NOTE: This release of the SDK does not support writing Arm Non-Secure binaries to run under the wing of an Arm Secure binary. This support will be added in a subsequent release.
- Compiler support is widening - we always recommend a recent version.)
- All recent GCCs are supported on Arm. (GCC 14 has not yet been tested for full support though).
  - Very recent GCCs are required on RISC-V due to the bleeding-edge nature of some of the processor instructions.
  - Recent LLVM Embedded Toolchain for ArmRM versions are supported on Arm.
  - Pigweed LLVM is supported for Arm.
  - For further details see the Raspberry Pi Pico-series C/C++ SDK book.
- Bazel may be used to build the SDK on Arm. See the README. Note that the Bazel build is community-provided and maintained.

Building Documentation

- The docs build target to build the HTML code documentation now builds a set of documentation peculiar to your particular PICO_PLATFORM setting.
- PICO_PLATFORM=combined-docs can be used (just for building docs) to build the combined documentation for both RP2040 and RP2350.

Fixed Issues

You can see a list of individual commits here, and a list of resolved issues here.

Note these only include public changes made since version 1.5.1. The majority of new code and collateral fixes for the previously unannounced RP2350 were developed and committed in private and delivered as a single "squashed" commit.

Release 2.1.0 (25 November 2024)

Adds support for Pico 2 W.

Board Support

The following boards have been added and may be specified via PICO_BOARD:

- adafruit_feather_rp2350
- datanoisetv_rp2350_dsp

- hellbender_0001
- machdyne_werkzeug
- pico2_w
- pimoroni_pico_plus2_w_rp2350
- sparkfun_thingplus_rp2350

The following board configurations have been modified:

- pimoroni_plasma2350 - corrected flash size, renamed SPICE to SPCE
- pimoroni_tiny2350 - corrected flash size

Notable Library Changes/Improvements

Clock dividers in general

A variety of methods which set clock dividers using an integer part and a fractional part, which might have been hardware_xxx_set_clkdiv_int_frac(uint16_t div_int, uint8_t div_frac) have been modified to hardware_xx_set_clkdiv_int_frac8(uint32_t div_int, uint8_t div_frac). This has been done for consistency and to make the APIs more resistant to hardware changes. The old APIs are preserved for backwards compatibility.

Previously, when converting from floating-point clock divider values to the fixed point use by the hardware, the floating-point value was rounded down. The new default (as configured by PICO_CLKDIV_ROUND_NEAREST) is to round to the nearest achievable value. This minor change in behavior was deemed better in general, which is why the default was changed. You may set PICO_CLKDIV_ROUND_NEAREST=0 to restore the previous behaviour by default (note that individual libraries have their own configuration values which can be used to change the behaviour on a per-library basis).

cmsis

- Fixed exception renaming for RP2350.

hardware_adc

- Added PICO_ADC_CLKDIV_ROUND_NEAREST for controlling rounding of floating-point clock dividers.

hardware_clocks

- Corrected spelling of PICO_CLOCK_AJDUST_PERI_CLOCK_WITH_SYS_CLOCK to PICO_CLOCK_ADJUST_PERI_CLOCK_WITH_SYS_CLOCK. The former is still supported.
- vco_calc.py now outputs SYS_CLK_HZ in the CMake output, which is required for clock_get_hz(clk_sys) to return the correct value.
- Renamed clock_gpio_init_int_frac() to clock_gpio_init_int_frac8() to clarify that it takes an 8-bit fraction; the old name is still supported.
- Added clock_gpio_init_int_frac16() to specify the fraction with 16-bit precision (RP2350 has 16 bits of precision). This method can still be called on RP2040 in which case the low 8-bits are ignored.
- Added PICO_CLOCK_GPIO_CLKDIV_ROUND_NEAREST for controlling rounding of floating-point clock dividers.

hardware_dma

- Fixed dma_channel_cleanup() to disable the channel with the new DMA IRQs added in RP2350.

hardware_exception

- Added missing Cortex-M33 exception numbers.

hardware_flash

- Prevented flash functions flash_range_eraae(), flash_range_program(), and flash_do_cmd() from trashing the user's CS1 QMI configuration on RP2350.
- Fixed issue with flash_safe_execute on FreeRTOS SMP.

hardware_i2c

- Added i2c_write_burst_blocking and i2c_read_burst_blocking to send/receive multiple bytes without intervening stops.
- Fixed rare hang during i2c_read_blocking.

hardware_interp

- Renamed interp_add_accumulater() to interp_add_accumulator(). The old incorrect spelling is still supported.

hardware_pio

- Added pio_sm_set_pins64(), pio_sm_set_pins_with_mask64() and pio_sm_set_pindirs_with_mask64() to allow setting of >32 pins.
- Much improved documentation of how GPIO numbers > 32 are handled.
- Fixed a bug in the use of a "jmp pin" > 32.
- Fixed implementation of sm_config_set_in_pin_count().
- Renamed sm_config_set_clkdiv_int_frac() to sm_config_set_clkdiv_int_frac8() to clarify that it takes an 8-bit fraction; the old name is still supported. Note that "int" part in the new method is 32-bit not 16-bit for consistency with other clkdiv methods.
- Renamed pio_calculate_clkdiv_from_float() to pio_calculate_clkdiv8_from_float() to clarify that it produces an 8-bit fraction; the old name is still supported. Note that "int" part in the new method is 32-bit not 16-bit for consistency with other clkdiv methods.
- Added PICO_PIO_CLKDIV_ROUND_NEAREST for controlling rounding of floating-point clock dividers.

hardware_pwm

- Renamed pwm_config_set_clkdiv_int_frac() to pwm_config_set_clkdiv_int_frac4() to clarify that it takes an 4-bit fraction; the old name is still supported. Note that "int" part in the new method is 32-bit not 8-bit for consistency with other clkdiv methods.
- Added PICO_PWM_CLKDIV_ROUND_NEAREST for controlling rounding of floating-point clock dividers.

hardware_timer

- Fixed bug with alarms when using RP2350's new TIMER1.
- Corrected signature of hardware_alarm_get_irq_num() method added in SDK version 2.0.0. The variant that takes (and uses) a timer instance is called timer_hardware_alarm_get_irq_num().

pico_aon_timer

- Added aon_timer_start_calendar(), aon_timer_set_time_calendar(), aon_timer_get_time_calendar() and aon_timer_enable_alarm_calendar() methods. These are equivalent to the non-_calendar() variants, except they deal in calendar (date/) time, rather than time intervals. These new variants are preferred on RP2040 since otherwise a date/time conversion must be performed which pulls in a lot of C library code. For the same reason, the pre-existing variants are preferred on RP2350. This discrepancy results from the different hardware used for the AON timer on RP2040 and RP2350.

pico_atomic

- Fixed atomic use between core 0 and core 1.

pico_async_context

- Fixed possible HardFault in execute_sync() on FreeRTOS.

pico_binary_info

- bi_Xpins_with_names() macros now work correctly when pin numbers are not in order.

pico_bootrom

- Added rom_reset_usb_boot_extra() which supports an "activity" GPIO pin > 32 and GPIO pin inversion (active low).
- Bootrom methods that may write to flash are now protected with flash_safe_execute(). This affects rom_flash_op() and rom_explicit_buy().

pico_bootsel_via_double_reset

- Fixed implementation on RP2350. Note the RP2350 bootrom also provides this support if enabled via OTP, however this library can be used when that is not enabled.

pico_crt0

- __HeapLimit is now correctly set by the default linker scripts again.
- Fixed linker option -Wl,--print-memory-usage showing 100% RAM used.

pico_clib_interface

- Made some small improvements to picolibc integration.

pico_cyw43_driver

- Allow user configuration of Wi-Fi pins (including pin numbers >32) and SPI clock, including dynamic SPI clock configuration at runtime.

- Updated cye43_driver to revision cf924bb.
- Renamed cyw43_set_pio_clkdiv_int_frac() to cyw43_set_pio_clkdiv_int_frac8() to clarify that it takes an 8-bit fraction; the old name is still supported. Note that "int" part in the new method is 32-bit not 16-bit for consistency with other clkdiv methods.
- Renamed CYW43_PIO_CLOCK_DIV_FRAC8 to CYW43_PIO_CLOCK_DIV_FRAC. The old name is still supported.
- RISC-V is now supported.
- Added PICO_BTSTACK_CYW43_MAX_HCI_PROCESS_LOOP_COUNT configuration option, which can be used to prevent starvation in high frequency Bluetooth scenarios.

pico_flash

- Support serial flash with >8 byte unique id, using the last 8 bytes rather than the first.

pico_float

- Added optimized add/sub/mul implementations for Hazard3 for better floating point speed.

pico_malloc

- Fixed deadlock in calloc() and realloc() with picolibc.

pico_platform

- Added pico_default_asm_volatile_goto().

pico_standard_binary_info

- Added back boot_stage2 binary info (missing in SDK version 2.0.0).

pico_stdio_uart

- Fixed stdio_flush() when used with stdio_uart_init_full().
- Fixed race condition in stdio_set_chars_available_callback().

pico_stdio_usb

- Fixed Windows issue with the device not showing up if the reset interface is disabled.
- Added support for resetting to USB boot with an activity LED pin > 32 or with the LED active low (on RP2350).
- Added PICO_STDIO_USB_RESET_BOOTSEL_FIXED_ACTIVITY_LED_ACTIVE_LOW setting for RP2350.

pico_time

- Fixed race condition which could cause alarms to be lost.
- Fixed continuous wakeup in best_effort_wfe_or_timeout() on RP2350.

pico_util

- Added datetime_to_tm() and tm_to_datetime() for converting C library date/times to/from RP2040 RTC date/times.
- Added pico_localtime_r() and pico_mktime() for use by pico_util time conversion code. These methods cass the equivalent C library function, but are defined weakly so the user can provide their own.

TinyUSB

- Updated TinyUSB to 0.17.0.

New Libraries

boot_bootrom_headers

Split out the headers defining the bootrom interface - that might be used outside the SDK - from pico_bootrom which is focused on calling the bootrom from the SDK, and has non-trivial dependencies.

hardware_xip_cache

Provides XIP cache maintenance APIs:

- RP2040 support for cache invalidation.
- RP2350 support for cache invalidation/cleaning/pinning.

Miscellaneous

- Numerous documentation corrections/improvements.
- Various build warnings fixed in exotic compiler configurations.
- RP2350 A0/A1 silicon are no longer supported.

pioasm

- Fixed disassembly of mov rx_fifo, … and mov …, rx_fifo instructions/

Build

- Made build dependent on any signature files or embedded-partition-table JSON.
- Added back .hex file output (lost in SDK version 2.0.0).
- Made PICO_FLASH_SIZE_BYTES and PICO_CYW43_SUPPORTED if specified in CMake, correctly affect the compiled code.
- Various corrections to library dependencies.
- Added PANIC and AUTO_INIT_MUTEX options to pico_minimize_runtime().
- Made boot_stage2 build reproducible (same binary if no source changes).

Bazel Build

- Add support for building on Raspberry Pi OS.
- More CMake build configuration options supported.
- Preview support for Wi-Fi builds.

New Examples

This release adds the following examples to the `pico_examples` repository:

Example	Description
`binary_info/blink_any`	Uses bi_ptr variables to create a configurable blink binary - see the separate readme for mote details
`binary_info/hello_anything`	Uses bi_ptr variables to create a configurable hello_world binary - see the separate readme for more details
`i2c/slave_mem_i2c_burst`	i2c slave example where the slave implements a 256 byte memory. This version inefficiently writes each byte in a separate call to demonstrate read and write burst mode.
`pico_w/wifi/picow_blink_slow_clock`	Blinks the on-board LED (which is connected via the WiFi chip) with a slower system clock to show how to reconfigure communication with the WiFi chip at run time under those circumstances
`pico_w/wifi/picow_blink_fast_clock`	Blinks the on-board LED (which is connected via the WiFi chip) with a faster system clock to show how to reconfigure communication with the WiFi chip at build time under those circumstances
`pico_w/wifi/picow_http_client`	Demonstrates how to make http and https requests
`pico_w/wifi/picow_http_client_verify`	Demonstrates how to make a https request with server authentication
`pico_w/wifi/freertos/picow_freertos_http_client_sys`	Demonstrates how to make a https request in NO_SYS=0 (i.e. full FreeRTOS integration)
`universal/blink`	Same as the blink example, but universal.
`universal/nuke_universal`	Same as the flash/nuke example, but universal. On RP2350 runs as a packaged SRAM binary, so is written to flash and copied to SRAM by the bootloader

Release 2.1.1 (18 February 2025)

This is a minor release of the SDK with many bug fixes and documentation improvements, along with some new features.

Highlights are listed below, or you can see the full list of individual commits here, and the full list of resolved issues here. Board Support

The following boards have been added and may be specified via `PICO_BOARD`:

- `sparkfun_iotnode_lorawan_rp2350`
- `waveshare_pico_cam_a`

- waveshare_rp2040_ble
- waveshare_rp2040_eth
- waveshare_rp2040_geek
- waveshare_rp2040_matrix
- waveshare_rp2040_pizero
- waveshare_rp2040_power_management_hat_b
- waveshare_rp2040_tiny
- waveshare_rp2040_touch_lcd_1.28
- waveshare_rp2350_eth
- waveshare_rp2350_geek
- waveshare_rp2350_lcd_0.96
- waveshare_rp2350_lcd_1.28
- waveshare_rp2350_one
- waveshare_rp2350_plus_4mb
- waveshare_rp2350_plus_16mb
- waveshare_rp2350_tiny
- waveshare_rp2350_touch_lcd_1.28
- waveshare_rp2350_zero

The following board configurations have been modified:

- adafruit_feather_rp2350 - Increased the XOSC startup delay
- seeed_xiao_rp2350 - Increased the default SPI clock divider
- waveshare_rp2040_lcd_0.96 - Renamed WAVESHARE_RP2040_LCD_ constants to WAVESHARE_LCD_
- waveshare_rp2040_lcd_1.28 - Renamed WAVESHARE_RP2040_LCD_ constants to WAVESHARE_LCD_

NOTE

This release changes the default PICO_XOSC_STARTUP_DELAY_MULTIPLIER from 1 to 6, unless specified by a board header file. This creates a delay of 6ms, since testing of the recommended crystal shows it can take up to this long to stabilize.

200 MHz Clock Support for RP2040

RP2040 has now been certified to run at a system clock of 200 MHz when using a regulator voltage of at least 1.15 volts.

The SDK by default performs clock setup for you before your program enters main(). If you haven't customized the clock configuration in any way, it will attempt to configure the system clock based on the value of SYS_CLK_MHZ (or SYS_CLK_KHZ /SYS_CLK_HZ if specified instead). Without further information from you, it can only do this for specific clock frequencies.

In prior versions of the SDK, only one specific clock frequency was defined per platform, 125 MHz for RP2040 and 150 MHz for RP2350, which also happen to be the default values for SYS_CLK_MHZ.

With this version of the SDK, you can now select a 200 MHz clock for RP2040 by setting SYS_CLK_MHZ=200 via preprocessor define. The regulator voltage will automatically be raised for you if necessary.

We may certify new frequencies for the different platforms in the future. The original SYS_CLK_MHZ defaults are left unchanged because not all programs would function correctly at a different system clock frequency. If, however, your project would always benefit from the fastest clock, you may now define `PICO_USE_FASTEST_SUPPORTED_CLOCK=1` via CMake variable or as a preprocessor define, and it will always use the fastest supported system clock frequency for the platform in the future.

Notable Library Changes/Improvements

hardware_clocks

- Corrected documentation and implementation of `clock_configure()` supporting the full range of clock dividers.
- Added `PICO_USE_FASTEST_SUPPORTED_CLOCK` and PLL configuration for 200 MHz on RP2040.

hardware_flash

- Moved internal flash helper function to run from RAM instead of flash. This adds support to builds other than `COPY_TO_RAM`.

hardware_irq

- Added significantly improved documentation around IRQ handlers when using both cores.
- Added `enable_interrupts()` and `disable_interrupts()` methods for when you don't care about saving or restoring the current interrupt state.
- Added `irq_has_handler()` method to tell if a handler is installed for a particular IRQ number.

hardware_pio

- Fixed support for WAIT gpio with GPIO number >= 32.

pico_aon_timer

- Added a 2 RTC-clock propagation delay at the end of `aon_timer_set_time_calendar()` on RP2040, such that reading back the time immediately afterwards returns the correct value.

pico_bootrom

- Added `rom_data_lookup_inline()` to complement `rom_data_lookup()`.

pico_btstack

- Updated BTStack to 1.6.2 from 1.6.1.
- Updated Raspberry Pi BTStack license to cover Pico 2 W, Pico 2 WH, and RM2.

pico_cyw43_driver

- Updated cyw43_driver to revision `c1075d4b`.
- Fixed rare issue when loading firmware.

`pico_double`

- Significantly cleaned up and improved documentation.
- Implemented the full complement of double conversion functions defined in `pico/double.h` across both RP2040 and RP2350 variants of `pico_double_pico`.

`pico_float`

- Significantly cleaned up and improved documentation.
- RP2350 `pico_float_pico_dcp` variant now enables `-msoft-float`, since if you've chosen to use DCP instead of VFP for single-precision floating-point, you probably don't want the compiler emitting inline VFP instructions either.
- Implemented the full complement of float conversion functions defined in `pico/float.h` across RP2040 and all RP2350 variants of `pico_float_pico`.

`pico_flash`

- Fixed a build error when using FreeRTOS with config `SUPPORT_DYNAMIC_ALLOCATION=0`.

`pico_lwip`

- Fixed build with `PPP_SUPPORT=1` when using `pico_lwip_nosys`.

`pico_mbedtls`

- Added correct cleanup of RP2350 SHA256 state during `mbedtls_sha256_free()`.

`pico_multicore`

- Added `multicore_lockout_victim_deinit()`
- `multicore_reset_core1()` now marks Core 1 as de-initialized w.r.t. `multicore_lockout_victim_` functions, allowing `multicore_lockout_victim_init()` to perform correctly after the reset.

`pico_runtime_init`

- Added `SYS_CLK_VREG_VOLTAGE_AUTO_ADJUST` to indicate the voltage regular should be set to `SYS_CLK_VREG_VOLTAGE_MIN` during default clock setup in order to support the configured system clock frequency.

`pico_sha256`

- Added `pico_sha256_cleanup()` to clean up from an in-progress SHA256 operation which was not completed via `pico_sha256_finish()`.

`pico_stdio_usb`

- Allow user to override `CFG_TUD_CDC_RX_BUFSIZE`, `CFG_TUD_CDC_TX_BUFSIZE`, and `CFG_TUD_CDC_EP_BUFSIZE` defines to increase performance.

pico_time

- Fixed a rare race condition that could cause alarms/repeating timers to get "lost".

TinyUSB

- Updated TinyUSB to 0.18.0 from 0.17.0

FreeRTOS

- Upstreamed FreeRTOS support for RP2350 (Arm/RISC-V) to https://github.com/FreeRTOS/FreeRTOS-Kernel. However, this is not yet in any official release, so you should use the latest from the main branch there. Be sure to initialize the submodules, since RP2350 support is in a submodule.
- If your project embeds FreeRTOS_Kernel_import.cmake, you should update to the latest version here which works for both RP2040 and RP2350.

Pioasm

- Fixed encoding of WAIT GPIO with GPIO number >= 32.
- Python output now correctly emits word(x) for all PIO version 1 (RP2350) PIO instructions.

SVD

- Fixed access type for DMA `CHAN_ABORT` register to be read-write (with clear-on-write) for both RP2040 and RP2350.

Build

- Added support for GCC 14.
- Added support for LLVM Embedded Toolchain For Arm 19.x.
- Added support for multiple `.pio` files in `pico_generate_pio_header()`.
- Updated `.DIS` files for builds using LLVM/Clang on RP2350 to contain correct disassembly for VFP floating point instructions.
- Fixed some newer CMake version deprecation warnings.
- Added explicit license to pico_sdk_import.cmake as it is copied into external projects.

Bazel Build

- Updated LLVM/Clang toolchain to fix stack overflow issue with `fma()`-related math functions.

New Examples

This release adds the following examples to the `pico_examples` repository:

Example	Description
`pico_w/wifi/mqtt/picow_mqtt_client`	Demonstrates how to implement a MQTT client application
`pio/uart_pio_dma`	Send and receive data from a UART implemented using the PIO and DMA

Example	Description
`usb/device/dev_multi_cdc`	A USB CDC device example with two serial ports, one of which is used for standard SDK stdio. The example exposes two serial ports over USB to the host. The first port is used for stdio, and the second port is used for a simple echo loopback. You can connect to the second port and send some characters, and they will be echoed back on the first port while you will receive a "OK\r\n" message on the second port indicating that the data was received.

Documentation Release History

20 February 2025

- Updated to include information about the 2.1.1 release of the Raspberry Pi Pico C SDK.
- Added some missing examples to the 2.1.0 release notes.
- Added information about running RP2040 at 200 MHz.

05 December 2024

- Added support for Pico 2 W.

25 November 2024

- Added support for Pico 2 W.
- Updated instructions for the 2.1.0 release of the Raspberry Pi Pico C SDK.

15 October 2024

- Corrected minor typos and formatting issues.
- Switched back to separate release histories per PDF.

02 May 2024

- Corrected minor typos and formatting issues.
- Fixed formatting of Pico C SDK API level documentation.

02 February 2024

- Corrected minor typos and formatting issues.
- Updated documentation to include information about Raspberry Pi 5.

14 June 2023

- Corrected minor typos and formatting issues.
- Updated instructions for the 1.5.1 release of the Raspberry Pi Pico C SDK.
- Added documentation around Bluetooth support for Raspberry Pi Pico W.

03 March 2023

- Corrected minor typos and formatting issues.
- Updated instructions for the 1.5.0 release of the Raspberry Pi Pico C SDK.

01 December 2022

- Corrected minor typos and formatting issues.
- Replaced SDK library documentation with links to the online version.

30 June 2022

- Corrected minor typos and formatting issues.

17 June 2022

- Corrected minor typos and formatting issues.
- Elaborated explanation of SDK configuration.

04 November 2021

- Corrected minor typos and formatting issues.

03 November 2021

- Corrected minor typos and formatting issues.
- Described SDK "panic" handling.

30 Sepe 2021

- Corrected minor typos and formatting issues.

23 June 2021

- Corrected minor typos and formatting issues.

07 June 2021

- Corrected minor typos and formatting issues.
- Added SDK release history.

13 April 2021

- Corrected minor typos and formatting issues.
- Clarified that all source code in the documentation is under the 3-Clause BSD license.

07 April 2021

- Corrected minor typos and formatting issues.

05 March 2021

- Corrected minor typos and formatting issues.

23 February 2021

- Corrected minor typos and formatting issues.
- Changed font.

01 February 2021

- Corrected minor typos and formatting issues.

26 January 2021

- Corrected minor typos and formatting issues.

21 January 2021

- Initial release.

Raspberry Pi