Raspberry Pi Pico C/C++ SDK

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

Colophon

Copyright © 2020 Raspberry Pi (Trading) Ltd.

The documentation of the RP2040 microcontroller is licensed under a Creative Commons Attribution-NoDerivatives 4.0 International (CC BY-ND).

build-date: 2021-01-26
build-version: b4f84f3-dirty

About the SDK

Throughout the text "the SDK" refers to our Raspberry Pi Pico SDK. More details about the SDK can be found throughout this book.

Release History

Table 1. Release History

Release	Date	Description
1.0	21/Jan/2021	Initial release.
1.1	26/Jan/2021	* Minor corrections * Extra information about using DMA with ADC * Clarified M0+ and SIO CPUID registers * Added more discussion of Timers * Update Windows and macOS build instructions * Renamed books and optimised size of output PDFs

The latest release can be found at https://datasheets.raspberrypi.org/pico/sdk/raspberry-pi-pico-c-sdk.pdf.

Legal Disclaimer Notice

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

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

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

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

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

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

Table of Contents

Colophon . . . 1
Release History . . . 1
Legal Disclaimer Notice . . . 1
1. About the SDK . . . 6
1.1. Introduction . . . 6
1.2. Anatomy of a SDK Application . . . 6
2. SDK Architecture . . . 9
2.1. The Build System . . . 9
2.2. Every Library is an INTERFACE . . . 10
2.3. SDK Library Structure . . . 11
2.3.1. Higher-level Libraries . . . 11
2.3.2. Runtime Support (pico_runtime, pico_standard_link) . . . 11
2.3.3. Hardware Support Libraries . . . 12
2.3.4. Hardware Structs Library . . . 13
2.3.5. Hardware Registers Library . . . 14
2.3.6. TinyUSB Port . . . 15
2.4. Directory Structure . . . 15
2.4.1. Locations of Files . . . 16
2.5. Conventions for Library Functions . . . 17
2.5.1. Function Naming Conventions . . . 17
2.5.2. Return Codes and Error Handling . . . 18
2.5.3. Use of Inline Functions . . . 18
2.5.4. Builder Pattern for Hardware Configuration APIs . . . 19
2.6. Customisation and Configuration Using Preprocessor variables . . . 20
2.6.1. Preprocessor Variables via Board Configuration File . . . 20
2.6.2. Preprocessor Variables Per Binary or Library via CMake . . . 20
2.7. SDK Runtime . . . 21
2.7.1. Standard Input/Output (stdio) Support . . . 21
2.7.2. Floating-point support . . . 22
2.7.3. Hardware Divider . . . 25
2.8. Multi-core support . . . 26
2.9. Using C++ . . . 27
2.10. Next Steps . . . 27
3. Using Programmable I/O (PIO) . . . 28
3.1. What is Programmable I/O (PIO)? . . . 28
3.1.1. Background . . . 28
3.1.2. I/O Using dedicated hardware on your PC . . . 28
3.1.3. I/O Using dedicated hardware on your Raspberry Pi or microcontroller . . . 28
3.1.4. I/O Using software control of GPIOs (*"bit-banging"*) . . . 29
3.1.5. Programmable I/O Hardware using FPGAs and CPLDs . . . 30
3.1.6. Programmable I/O Hardware using PIO . . . 30
3.2. Getting started with PIO . . . 31
3.2.1. A First PIO Application . . . 31
3.2.2. A Real Example: WS2812 LEDs . . . 35
3.2.3. PIO and DMA (A Logic Analyser) . . . 42
3.2.4. Further examples . . . 47
3.3. Using PIOASM, the PIO Assembler . . . 47
3.3.1. Usage . . . 47
3.3.2. Directives . . . 48
3.3.3. Values . . . 49
3.3.4. Expressions . . . 49
3.3.5. Comments . . . 49
3.3.6. Labels . . . 49
3.3.7. Instructions . . . 50
3.3.8. Pseudoinstructions . . . 50

3.3.9. Output pass through . . . 50
3.3.10. Language generators . . . 51
3.4. PIO Instruction Set Reference . . . 56
3.4.1. Summary . . . 56
3.4.2. JMP . . . 57
3.4.3. WAIT . . . 58
3.4.4. IN . . . 59
3.4.5. OUT . . . 60
3.4.6. PUSH . . . 61
3.4.7. PULL . . . 62
3.4.8. MOV . . . 63
3.4.9. IRQ . . . 64
3.4.10. SET . . . 65
4. Library Documentation . . . 67
4.1. Hardware APIs . . . 67
4.1.1. hardware_adc . . . 67
4.1.2. hardware_base . . . 72
4.1.3. hardware_claim . . . 73
4.1.4. hardware_clocks . . . 75
4.1.5. hardware_divider . . . 81
4.1.6. hardware_dma . . . 89
4.1.7. channel_config . . . 96
4.1.8. hardware_flash . . . 100
4.1.9. hardware_gpio . . . 102
4.1.10. hardware_i2c . . . 111
4.1.11. hardware_interp . . . 118
4.1.12. interp_config . . . 123
4.1.13. hardware_irq . . . 126
4.1.14. hardware_pio . . . 131
4.1.15. sm_config . . . 145
4.1.16. hardware_pll . . . 151
4.1.17. hardware_pwm . . . 152
4.1.18. hardware_resets . . . 161
4.1.19. hardware_rtc . . . 163
4.1.20. hardware_spi . . . 165
4.1.21. hardware_sync . . . 170
4.1.22. hardware_timer . . . 175
4.1.23. hardware_uart . . . 179
4.1.24. hardware_vreg . . . 185
4.1.25. hardware_watchdog . . . 185
4.1.26. hardware_xosc . . . 187
4.2. High Level APIs . . . 188
4.2.1. pico_multicore . . . 189
4.2.2. fifo . . . 191
4.2.3. pico_stdlib . . . 192
4.2.4. pico_sync . . . 194
4.2.5. critical_section . . . 194
4.2.6. mutex . . . 196
4.2.7. sem . . . 198
4.2.8. pico_time . . . 200
4.2.9. timestamp . . . 200
4.2.10. sleep . . . 203
4.2.11. alarm . . . 204
4.2.12. repeating_timer . . . 210
4.2.13. pico_util . . . 213
4.2.14. datetime . . . 213
4.2.15. pheap . . . 214
4.2.16. queue . . . 214
4.3. Third-party Libraries . . . 218
4.3.1. tinyusb_device . . . 218

4.3.2. tinyusb_host . . . 218
4.4. Runtime Infrastructure . . . 218
4.4.1. boot_stage2 . . . 219
4.4.2. pico_base . . . 219
4.4.3. pico_bit_ops . . . 219
4.4.4. pico_bootrom . . . 220
4.4.5. pico_cxx_options . . . 221
4.4.6. pico_divider . . . 221
4.4.7. pico_double . . . 229
4.4.8. pico_float . . . 229
4.4.9. pico_int64_ops . . . 230
4.4.10. pico_malloc . . . 230
4.4.11. pico_mem_ops . . . 230
4.4.12. pico_platform . . . 230
4.4.13. pico_printf . . . 230
4.4.14. pico_runtime . . . 230
4.4.15. pico_stdio . . . 230
4.4.16. pico_stdio_semihosting . . . 232
4.4.17. pico_stdio_uart . . . 232
4.4.18. pico_stdio_usb . . . 234
4.4.19. pico_standard_link . . . 234
4.5. External API Headers . . . 234
4.5.1. boot_picoboot . . . 234
4.5.2. boot_uf2 . . . 234
Appendix A: App Notes . . . 235
Attaching a 7 segment LED via GPIO . . . 235
Wiring information . . . 235
List of Files . . . 235
Bill of Materials . . . 237
DHT-11, DHT-22, and AM2302 Sensors . . . 238
Wiring information . . . 238
List of Files . . . 239
Bill of Materials . . . 241
Attaching a BME280 temperature/humidity/pressure sensor via SPI . . . 241
Wiring information . . . 241
List of Files . . . 242
Bill of Materials . . . 246
Attaching a MPU9250 accelerometer/gyroscope via SPI . . . 246
Wiring information . . . 247
List of Files . . . 247
Bill of Materials . . . 250
Attaching a MPU6050 accelerometer/gyroscope via I2C . . . 250
Wiring information . . . 251
List of Files . . . 251
Bill of Materials . . . 253
Attaching a 16x2 LCD via I2C . . . 254
Wiring information . . . 254
List of Files . . . 254
Bill of Materials . . . 257
Appendix B: SDK Configuration . . . 258
Configuration Parameters . . . 258
Appendix C: CMake Build Configuration . . . 262
Configuration Parameters . . . 262
Control of binary type produced (advanced) . . . 263
Appendix D: Board Configuration . . . 264
Board Configuration . . . 264
The Configuration files . . . 264
Building applications with a custom board configuration . . . 265
Available configuration parameters . . . 265
Appendix E: Building the SDK API documentation . . . 266

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 RP2040-based devices such as Raspberry Pi Pico 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 RP2040'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, USB, synchronization and multi-core programming, along with additional high level functionality built using PIO such as audio. 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, 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, full fledged runtime environments such as MicroPython, to low level software such as RP2040's on-chip bootrom itself.

> **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** 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**, in more detail.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/master/blink/blink.c Lines 1 - 19

```
1  /**
2   * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
3   *
4   * SPDX-License-Identifier: BSD-3-Clause
5   */
6
7  #include "pico/stdlib.h"
8
9  int main() {
10     const uint LED_PIN = 25;
11     gpio_init(LED_PIN);
12     gpio_set_dir(LED_PIN, GPIO_OUT);
13     while (true) {
14         gpio_put(LED_PIN, 1);
15         sleep_ms(250);
16         gpio_put(LED_PIN, 0);
17         sleep_ms(250);
18     }
```

```
19 }
```

This program consists only of a single C file, with a single function. 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, after doing things like initialising static variables with their values. 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 RP2040 (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 RP2040 on-chip hardware.

So, using mainly the `hardware_gpio` and `pico_time` libraries, this C program will blink an LED connected to GPIO25 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

```
blink
├── blink.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 RP2040-based microcontroller 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/tree/master/blink/CMakeLists.txt Lines 1 - 12

```
 1 add_executable(blink
 2         blink.c
 3         )
 4
 5 # Pull in our pico_stdlib which pulls in commonly used features
 6 target_link_libraries(blink pico_stdlib)
 7
 8 # create map/bin/hex file etc.
 9 pico_add_extra_outputs(blink)
10
11 # add url via pico_set_program_url
12 example_auto_set_url(blink)
```

The `add_executable` function in this file declares that a program called `blink` should be built from the C file shown earlier. This is also the target name used to build the program: in the `pico-examples` repository you can say `make blink` 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, like accessing the DMA hardware, you can call those libraries out 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` 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 RP2040's RAM or external flash through the Serial Wire Debug port, with a debugger setup like `gdb` and `openocd`. It's often easier to program your Raspberry Pi Pico or other RP2040 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 with an internal SDK tool called `elf2uf2`, which is bootstrapped automatically as part of the build process.

The `example_auto_set_url` function is to do with how you are able to read this source file in *this* document you are reading right now, and click links to take you to the listing on Github. You'll see this on the `pico-examples` applications, but it's not necessary on your own programs. You are seeing how the sausage is made.

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 components like `pico_standard_link` — which contains linker scripts and `crt0` for SDK — and `pico_runtime`, which contains code running between `crt0` and `main()`, getting the system into a state ready to run code by putting things like clocks and resets in a safe initial state. 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 *explicit dependencies* of your program, and you can choose not to use them, whilst still building against the SDK and using things like the `hardware` libraries.

Chapter 2. SDK Architecture

RP2040 is a powerful chip, and in particular was designed with a disproportionate amount of system RAM for its 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 tradeoffs 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 RP2040 *board* you are building for, to configure things like default pin mappings and features of SDK libraries. These defines are listed in Appendix B, and Board Configuration files are covered in more detail in Appendix D. Additionally Appendix C 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/tree/master/blink/CMakeLists.txt Lines 1 - 12

```
 1 add_executable(blink
 2         blink.c
 3         )
 4
 5 # Pull in our pico_stdlib which pulls in commonly used features
 6 target_link_libraries(blink pico_stdlib)
 7
 8 # create map/bin/hex file etc.
 9 pico_add_extra_outputs(blink)
10
11 # add url via pico_set_program_url
12 example_auto_set_url(blink)
```

Looking here at the blink example, we are defining a new executable `blink` with a single source file `blink.c`, with a single

dependency `pico_stdlib`. We also are using a 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 floating point and other optimized code contained in the bootrom within RP2040).

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

2.2. Every Library is an INTERFACE

All libraries within the SDK are `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` 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. The APIs are listed in Section 4.2. 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 RP2040's 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 will be forthcoming in the future (e.g. - Audio support (via PIO), DPI/VGA/MIPI Video support (via PIO) file system support, SDIO support via (PIO)), most of which are available but not yet fully supported/stable/documented in the pico-extras GitHub repository.

2.3.2. Runtime Support (pico_runtime, pico_standard_link)

These are libraries that bundle functionality which is common to most RP2040-based applications. APIs are listed in Section 4.4.

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

- Runtime startup and initialization

- Choice of language level single/double precision floating point support (and access to the fast on-RP2040 implementations)
- Compact `printf` support, and mapping of `stdout`
- Language level `/` and `%` support for fast division using RP2040`s hardware dividers
- The function `runtime_init()` which performs minimal hardware initialisation (e.g. default PLL and clock configuration), and calls functions with `constructor` attributes before entering `main()`

`pico_standard_link` encapsulates the standard linker setup needed to configure the type of application binary layout in memory, and link to any additional C and/or C++ runtime libraries. It also includes the default `crt0`, which provides the initial entry point from the flash second stage bootloader, contains the initial vector table (later relocated to RAM), and initialises static data and RAM-resident code if the application is running from flash.

NOTE

There is more high-level discussion of `pico_runtime` in Section 2.7

TIP

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

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 Section 4.1.

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 do not 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 RP2040. As such they can be used by low-level or other specialized applications that doesn'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/tree/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 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.

2.3.4. Hardware Structs Library

The `hardware_structs` library provides a set of C structures which represent the memory mapped layout of RP2040 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-proneness 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/tree/master/src/rp2040/hardware_structs/include/hardware/structs/pll.h Lines 14 - 22

```
14 typedef struct {
15     io_rw_32 cs;
16     io_rw_32 pwr;
17     io_rw_32 fbdiv_int;
18     io_rw_32 prim;
19 } pll_hw_t;
20
```

```
21 #define pll_sys_hw ((pll_hw_t *const)PLL_SYS_BASE)
22 #define pll_usb_hw ((pll_hw_t *const)PLL_USB_BASE)
```

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

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"
```

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 RP2040 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. This is done by naming either `tinyusb_dev` *or* `tinyusb_host` as a dependency of your application

IMPORTANT

RP2040 USB hardware supports both Host and Device modes, but the two can not be used concurrently.

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

2.4. Directory Structure

We have discussed libraries such as `pico_stdlib` and `hardware_gpio` above. Imagine you wanted to add some code using RP2040's 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
)
```

And 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 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 actually 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 2. Top-level directories

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

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

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

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

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

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 3. 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, or an integer return code from the `PICO_ERROR_` family; non error returns are `>= 0`.

2. Other items 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 buidls); You can set `PARAMS_ASSERTIONS_ENABLE_ALL=1` or `PARAMS_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 behavior for the `hardware_i2c` module

   In the future we expect to 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.

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.

One reason this is *particularly* effective in the context of hardware register access is that these functions often:

1. Have relatively many parameters, which
2. Are immediately shifted and masked to combine with some register value, and
3. 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/tree/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.

Appendix B has 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 behavior 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 RP2040 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 default `PICO_BOARD` if not specified is `pico`.

The board configuration provides a header file which specifies defaults if not otherwise specified; for example https://github.com/raspberrypi/pico-sdk/tree/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 may wish to specify your own board configuration in which case you can set PICO_BOARD_HEADER_DIRS in the environment or CMake to a semicolon separated list of paths to search for `my_board_name.h`.

2.6.2. Preprocessor Variables Per Binary or Library via CMake

We could modify the https://github.com/raspberrypi/pico-examples/tree/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 runtime packages a lightweight `printf` library by Marco Paland, linked as `pico_printf`. It also contains infrastructure for routing `stdout` and `stdin` to various hardware interfaces, which is documented under pico_stdio:

- 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 RP2040's dedicated USB hardware interface, in Device mode.
- (Experimental) minimal semihosting support to direct `stdout` to an external debug host connected via the Serial Wire Debug link on RP2040

These can be accessed using 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)
```

2.7.2. Floating-point support

The SDK provides a highly optimized single and double precision floating point implementation. In addition to being fast, many of the functions are actually implemented using support provided in the RP2040 bootrom. 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 RP2040 has single-cycle access (with a dedicated arbiter on the RP2040 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.

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

2.7.2.1. Functions

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

2.7.2.2. Speed / Tradeoffs

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

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

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

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

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

The following table shows the results from a benchmark

NOTE

Whilst the SDK floating point support makes use of the routines in the RP2040 bootrom, it hides some of the limitations of the raw ROM functions (e.g. limited sin/cos range), in order to be largely indistinguishable from the compiler-provided functionality. Certain smaller functions have also been re-implemented for even more speed outside of the limited bootrom space.

Table 4. SDK implementation vs GCC 9 implementation for ARM AEABI floating point functions (these unusually named

functions provide the support for basic operations on float and double types)

Function	ROM/SDK (µs)	GCC 9 (µs)	Speedup
__aeabi_fadd	72.4	99.8	138%
__aeabi_fsub	86.7	133.6	154%

__aeabi_frsub	89.8	140.6	157%
__aeabi_fmul	61.5	145	236%
__aeabi_fdiv	74.7	437.5	586%
__aeabi_fcmplt	39	61.1	157%
__aeabi_fcmple	40.5	61.1	151%
__aeabi_fcmpgt	40.5	61.2	151%
__aeabi_fcmpge	41	61.2	149%
__aeabi_fcmpeq	40	41.5	104%
__aeabi_dadd	99.4	142.5	143%
__aeabi_dsub	114.2	182	159%
__aeabi_drsub	108	181.2	168%
__aeabi_dmul	168.2	338	201%
__aeabi_ddiv	197.1	412.2	209%
__aeabi_dcmplt	53	88.3	167%
__aeabi_dcmple	54.6	88.3	162%
__aeabi_dcmpgt	54.4	86.6	159%
__aeabi_dcmpge	55	86.6	157%
__aeabi_dcmpeq	54	64.3	119%
__aeabi_f2iz	17	24.5	144%
__aeabi_f2uiz	42.5	106.5	251%
__aeabi_f2lz	63.1	1240.5	1966%
__aeabi_f2ulz	46.1	1157	2510%
__aeabi_i2f	43.5	63	145%
__aeabi_ui2f	41.5	55.8	134%
__aeabi_l2f	75.2	643.3	855%
__aeabi_ul2f	71.4	531.5	744%
__aeabi_d2iz	30.6	44.1	144%
__aeabi_d2uiz	75.7	159.1	210%
__aeabi_d2lz	81.2	1267.8	1561%
__aeabi_d2ulz	65.2	1148.3	1761%
__aeabi_i2d	44.4	61.9	139%
__aeabi_ui2d	43.4	51.3	118%
__aeabi_l2d	104.2	559.3	537%
__aeabi_ul2d	102.2	458.1	448%
__aeabi_f2d	20	31	155%
__aeabi_d2f	36.4	66	181%

2.7.2.3. 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
`pico`	Map all functions to an runtime assertion. You can use this when you know you don't want any floating point support to make sure it isn't accidentally pulled in by some library.

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

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

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

TIP

The `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.

2.7.2.3.1. 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.

2.7.3. Hardware Divider

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

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.

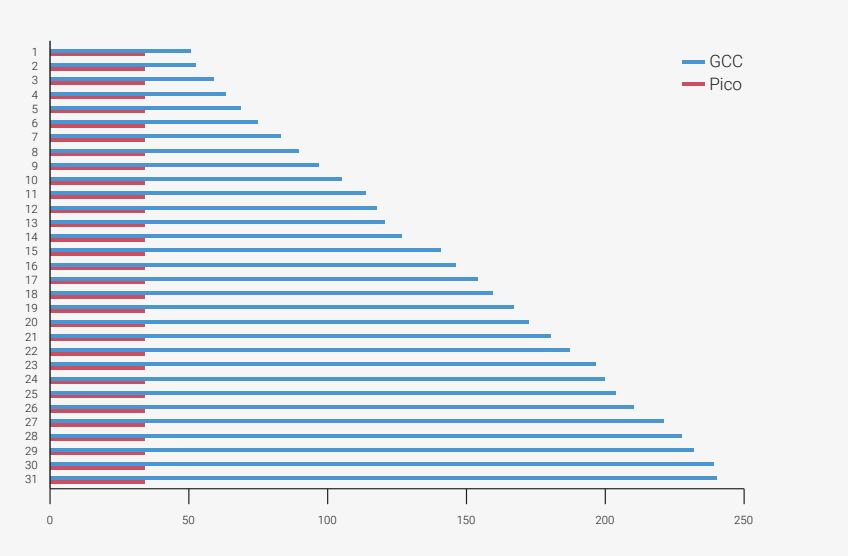
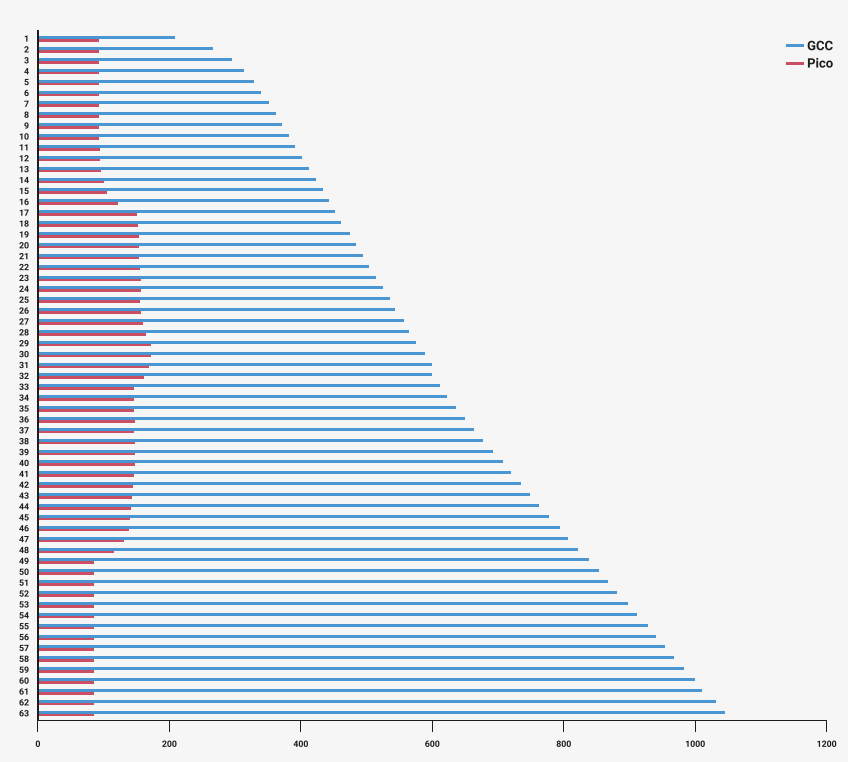


Figure 2. 64-bit divides by divider size using GCC library (blue), or the SDK library (red) with the RP2040 hardware divider.



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 with 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 (https://github.com/raspberrypi/pico-sdk/tree/master/src/common/pico_sync/include/pico/mutex.h) 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` 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 (Appendix C).

2.10. 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** book, which has detailed instructions on connecting to your RP2040 board and loading an application built with the SDK.

Chapter 3 gives some background on RP2040's 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 RP2040. It allows you to create new types of (or additional) hardware interfaces on your RP2040-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 3 of the **RP2040 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 RP2040 on Raspberry Pi Pico), 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 5. Types of hardware

Interface Speed	Interface
1-10 Hz	Push buttons, indicator LEDs
300 Hz	HDMI CEC
10-100 kHz	Temperature sensors (DHT11), one-wire serial
<100 kHz	I2C Standard mode
22-100+ kHz	PCM audio
300+ kHz	PWM audio
400-1200 kHz	WS2812 LED string
10-3000 kHz	UART serial
12 MHz	USB Full Speed
1-100 MHz	SPI
20-300 MHz	DPI/VGA video
480 MHz	USB High Speed
10-4000 MHz	Ethernet LAN
12-4000 MHz	SD card
250-20000 MHz	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 133 MHz, 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 RP2040 allows you to write small, simple programs for what are called *PIO state machines*, of which RP2040 has eight split across two PIO *instances*. A state machine is responsible for setting and reading one or more GPIOs, buffering data to or from the processor (or RP2040's ultra-fast DMA subsystem), and notifying the processor, via IRQ or polling, when 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 Cortex-M0+ processors on RP2040. In fact, they are similar in size (and therefore cost) to a standard SPI peripheral, such as the PL022 SPI also found on RP2040, 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 8 of them!

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 tradeoff 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 RP2040, don't! Use the PIO instead. Frankly this is true for anything that repeatedly reads or writes from GPIOs, but certainly anything which aims to transfer data.

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 RP2040-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/tree/master/pio/hello_pio/hello.pio Lines 7 - 15

```
 7 .program hello
 8
 9 ; Repeatedly get one word of data from the TX FIFO, stalling when the FIFO is
10 ; empty. Write the least significant bit to the OUT pin group.
11
12 loop:
13     pull
14     out pins, 1
15     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/tree/master/pio/hello_pio/hello.pio Lines 18 - 33

```
18 static inline void hello_program_init(PIO pio, uint sm, uint offset, uint pin) {
19     pio_sm_config c = hello_program_get_default_config(offset);
20
21     // Map the state machine's OUT pin group to one pin, namely the `pin`
22     // parameter to this function.
23     sm_config_set_out_pins(&c, pin, 1);
24     // Set this pin's GPIO function (connect PIO to the pad)
25     pio_gpio_init(pio, pin);
26     // Set the pin direction to output at the PIO
27     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
28
29     // Load our configuration, and jump to the start of the program
30     pio_sm_init(pio, sm, offset, &c);
31     // Set the state machine running
```

```
32     pio_sm_set_enabled(pio, sm, true);
33 }
```

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/tree/master/pio/hello_pio/hello.c Lines 1 - 38

```
1  /**
2   * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
3   *
4   * SPDX-License-Identifier: BSD-3-Clause
5   */
6
7  #include "pico/stdlib.h"
8  #include "hardware/pio.h"
9  // Our assembled program:
10 #include "hello.pio.h"
11
12 int main() {
13     // Choose which PIO instance to use (there are two instances)
14     PIO pio = pio0;
15
16     // Our assembled program needs to be loaded into this PIO's instruction
17     // memory. This SDK function will find a location (offset) in the
18     // instruction memory where there is enough space for our program. We need
19     // to remember this location!
20     uint offset = pio_add_program(pio, &hello_program);
21
22     // Find a free state machine on our chosen PIO (erroring if there are
23     // none). Configure it to run our program, and start it, using the
24     // helper function we included in our .pio file.
25     uint sm = pio_claim_unused_sm(pio, true);
26     hello_program_init(pio, sm, offset, PICO_DEFAULT_LED_PIN);
27
28     // The state machine is now running. Any value we push to its TX FIFO will
29     // appear on the LED pin.
30     while (true) {
31         // Blink
32         pio_sm_put_blocking(pio, sm, 1);
33         sleep_ms(500);
34         // Blonk
35         pio_sm_put_blocking(pio, sm, 0);
36         sleep_ms(500);
37     }
```

```
38 }
```

You might recall that RP2040 has two PIO blocks, each of them 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 **RP2040 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 Raspberry Pi Pico board. 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 Raspberry Pi Pico or other RP2040-based board.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/master/pio/hello_pio/CMakeLists.txt Lines 1 - 15

```
 1 add_executable(hello_pio)
 2
 3 pico_generate_pio_header(hello_pio ${CMAKE_CURRENT_LIST_DIR}/hello.pio)
 4
 5 target_sources(hello_pio PRIVATE hello.c)
 6
 7 target_link_libraries(hello_pio PRIVATE
 8         pico_stdlib
 9         hardware_pio
10         )
11
12 pico_add_extra_outputs(hello_pio)
13
14 # add url via pico_set_program_url
15 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 Raspberry Pi Pico 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.

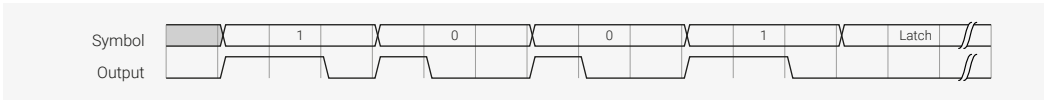


Figure 3. WS2812 line format. Wide positive pulse for 1, narrow positive pulse for 0, very long negative pulse for latch enable

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 Raspberry Pi Pico. 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/tree/master/pio/ws2812/ws2812.pio Lines 7 - 26

```
7 .program ws2812
8 .side_set 1
```

```
 9
10 .define public T1 2
11 .define public T2 5
12 .define public T3 3
13
14 .lang_opt python sideset_init = pico.PIO.OUT_HIGH
15 .lang_opt python out_init     = pico.PIO.OUT_HIGH
16 .lang_opt python out_shiftdir = 1
17
18 .wrap_target
19 bitloop:
20     out x, 1       side 0 [T3 - 1] ; Side-set still takes place when instruction stalls
21     jmp !x do_zero side 1 [T1 - 1] ; Branch on the bit we shifted out. Positive pulse
22 do_one:
23     jmp  bitloop   side 1 [T2 - 1] ; Continue driving high, for a long pulse
24 do_zero:
25     nop            side 0 [T2 - 1] ; Or drive low, for a short pulse
26 .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 **RP2040 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 **RP2040 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.

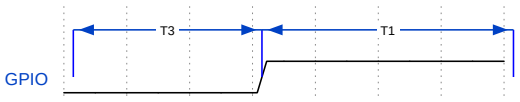


Figure 4. The state machine drives the line low for time T1 as it shifts out one data bit from the OSR, and then high for time T2 whilst branching on the value of the bit.

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:

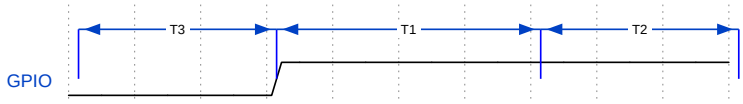


Figure 5. On a one data bit, the line is driven low for time T3, high for time T1, then high for an additional time T2

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:

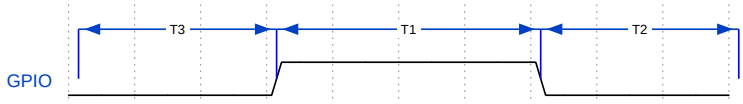


Figure 6. On a zero data bit, the line is driven low for time T3, high for time T1, then low again for time T1

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:

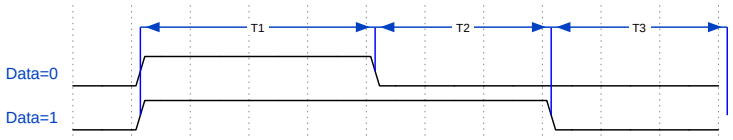


Figure 7. The line is initially low in the idle (latch) state, and the LED is waiting for the first rising edge. It sees our pulse timings in the order T1-T2-T3, until the very last T3, where it sees a much longer negative period once the state machine runs out of data.

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 **RP2040 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 RP2040, 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/tree/master/pio/ws2812/ws2812.pio Lines 31 - 47

```
31 static inline void ws2812_program_init(PIO pio, uint sm, uint offset, uint pin, float freq,
   bool rgbw) {
32
33     pio_gpio_init(pio, pin);
34     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
35
36     pio_sm_config c = ws2812_program_get_default_config(offset);
37     sm_config_set_sideset_pins(&c, pin);
38     sm_config_set_out_shift(&c, false, true, rgbw ? 32 : 24);
39     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
40
41     int cycles_per_bit = ws2812_T1 + ws2812_T2 + ws2812_T3;
42     float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit);
43     sm_config_set_clkdiv(&c, div);
44
45     pio_sm_init(pio, sm, offset, &c);
46     pio_sm_set_enabled(pio, sm, true);
47 }
```

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 RP2040's two 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_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_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_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_enable(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/tree/master/pio/ws2812/ws2812.c Lines 15 - 17

```
15 static inline void put_pixel(uint32_t pixel_grb) {
16     pio_sm_put_blocking(pio0, 0, pixel_grb << 8u);
17 }
```

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/master/pio/ws2812/ws2812.c Lines 19 - 24

```
19 static inline uint32_t urgb_u32(uint8_t r, uint8_t g, uint8_t b) {
20     return
21             ((uint32_t) (r) << 8) |
22             ((uint32_t) (g) << 16) |
23             (uint32_t) (b);
24 }
```

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/tree/master/pio/ws2812/ws2812.c Lines 40 - 45

```
40 void pattern_random(uint len, uint t) {
41     if (t % 8)
42         return;
43     for (int i = 0; i < len; ++i)
44         put_pixel(rand());
45 }
```

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/tree/master/pio/ws2812/ws2812.c Lines 76 - 100

```
 76 int main() {
 77     //set_sys_clock_48();
 78     stdio_init_all();
 79     puts("WS2812 Smoke Test");
 80
 81     // todo get free sm
 82     PIO pio = pio0;
 83     int sm = 0;
 84     uint offset = pio_add_program(pio, &ws2812_program);
 85
 86     ws2812_program_init(pio, sm, offset, PIN_TX, 800000, true);
 87
 88     int t = 0;
 89     while (1) {
 90         int pat = rand() % count_of(pattern_table);
 91         int dir = (rand() >> 30) & 1 ? 1 : -1;
 92         puts(pattern_table[pat].name);
 93         puts(dir == 1 ? "(forward)" : "(backward)");
 94         for (int i = 0; i < 1000; ++i) {
 95             pattern_table[pat].pat(150, t);
 96             sleep_ms(10);
 97             t += dir;
 98         }
 99     }
100 }
```

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.

RP2040 is 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 RP2040'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/tree/master/pio/logic_analyser/logic_analyser.c Lines 29 - 49

```
29 void logic_analyser_init(PIO pio, uint sm, uint pin_base, uint pin_count, float div) {
```

```
30      // Load a program to capture n pins. This is just a single `in pins, n`
31      // instruction with a wrap.
32      uint16_t capture_prog_instr = pio_encode_in(pio_pins, pin_count);
33      struct pio_program capture_prog = {
34              .instructions = &capture_prog_instr,
35              .length = 1,
36              .origin = -1
37      };
38      uint offset = pio_add_program(pio, &capture_prog);
39
40      // Configure state machine to loop over this `in` instruction forever,
41      // with autopush enabled.
42      pio_sm_config c = pio_get_default_sm_config();
43      sm_config_set_in_pins(&c, pin_base);
44      sm_config_set_wrap(&c, offset, offset);
45      sm_config_set_clkdiv(&c, div);
46      sm_config_set_in_shift(&c, true, true, 32);
47      sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_RX);
48      pio_sm_init(pio, sm, offset, &c);
49  }
```

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 **RP2040 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 125 MHz.

`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 **RP2040 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/tree/master/pio/logic_analyser/logic_analyser.c Lines 51 - 70

```
51 void logic_analyser_arm(PIO pio, uint sm, uint dma_chan, uint32_t *capture_buf, size_t
   capture_size_words,
52                        uint trigger_pin, bool trigger_level) {
53     pio_sm_set_enabled(pio, sm, false);
54     pio_sm_clear_fifos(pio, sm);
55
56     dma_channel_config c = dma_channel_get_default_config(dma_chan);
57     channel_config_set_read_increment(&c, false);
58     channel_config_set_write_increment(&c, true);
59     channel_config_set_dreq(&c, pio_get_dreq(pio, sm, false));
60
61     dma_channel_configure(dma_chan, &c,
62         capture_buf,        // Destinatinon pointer
63         &pio->rxf[sm],      // Source pointer
64         capture_size_words, // Number of transfers
65         true                // Start immediately
66     );
67
68     pio_sm_exec(pio, sm, pio_encode_wait_gpio(trigger_level, trigger_pin));
69     pio_sm_set_enabled(pio, sm, true);
70 }
```

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/tree/master/pio/logic_analyser/logic_analyser.c Lines 72 - 86

```
72 void print_capture_buf(const uint32_t *buf, uint pin_base, uint pin_count, uint32_t
   n_samples) {
73     // Display the capture buffer in text form, like this:
74     // 00: __--__--__--__--__--__--
75     // 01: ____----____----____----
76     printf("Capture:\n");
77     for (int pin = 0; pin < pin_count; ++pin) {
78         printf("%02d: ", pin + pin_base);
79         for (int sample = 0; sample < n_samples; ++sample) {
80             uint bit_index = pin + sample * pin_count;
```

```
81              bool level = !!(buf[bit_index / 32] & 1u << (bit_index % 32));
82              printf(level ? "-" : "_");
83          }
84          printf("\n");
85      }
86  }
```

We have everything we need now for RP2040 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 15 MHz on two GPIOs, and attaching our brand spanking new logic analyser to those same two GPIOs.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/master/pio/logic_analyser/logic_analyser.c Lines 88 - 125

```
 88 int main() {
 89     stdio_init_all();
 90     printf("PIO logic analyser example\n");
 91
 92     uint32_t capture_buf[(CAPTURE_PIN_COUNT * CAPTURE_N_SAMPLES + 31) / 32];
 93
 94     PIO pio = pio0;
 95     uint sm = 0;
 96     uint dma_chan = 0;
 97
 98     logic_analyser_init(pio, sm, CAPTURE_PIN_BASE, CAPTURE_PIN_COUNT, 1.f);
 99
100     printf("Arming trigger\n");
101     logic_analyser_arm(pio, sm, dma_chan, capture_buf, //;
102         (CAPTURE_PIN_COUNT * CAPTURE_N_SAMPLES + 31) / 32,
103         CAPTURE_PIN_BASE, true);
104
105     printf("Starting PWM example\n");
106     // PWM example: ------------------------------------------------------------
107     gpio_set_function(CAPTURE_PIN_BASE, GPIO_FUNC_PWM);
108     gpio_set_function(CAPTURE_PIN_BASE + 1, GPIO_FUNC_PWM);
109     // Topmost value of 3: count from 0 to 3 and then wrap, so period is 4 cycles
110     pwm_hw->slice[0].top = 3;
111     // Divide frequency by two to slow things down a little
112     pwm_hw->slice[0].div = 4 << PWM_CH0_DIV_INT_LSB;
113     // Set channel A to be high for 1 cycle each period (duty cycle 1/4) and
114     // channel B for 3 cycles (duty cycle 3/4)
115     pwm_hw->slice[0].cc =
116             (1 << PWM_CH0_CC_A_LSB) |
117             (3 << PWM_CH0_CC_B_LSB);
118     // Enable this PWM slice
119     pwm_hw->slice[0].csr = PWM_CH0_CSR_EN_BITS;
120     // ------------------------------------------------------------------------
121
122     dma_channel_wait_for_finish_blocking(dma_chan);
123
124     print_capture_buf(capture_buf, CAPTURE_PIN_BASE, CAPTURE_PIN_COUNT, CAPTURE_N_SAMPLES);
125 }
```

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 **RP2040 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 commandline 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 Pico SDK
                        python
                            Python file suitable for use with MicroPython
                        hex
                            Raw hex output (only valid for single program inputs)
-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 6. 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 the 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
.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 of a program
.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 (not using 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 7. 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 3.4.2)
(<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 8. 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>	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, unmodifed, 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_some_group` if the 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/tree/master/pio/ws2812/ws2812.pio Lines 1 - 85

```
 1 ;
 2 ; Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
 3 ;
 4 ; SPDX-License-Identifier: BSD-3-Clause
 5 ;
 6
 7 .program ws2812
 8 .side_set 1
 9
10 .define public T1 2
11 .define public T2 5
12 .define public T3 3
13
14 .lang_opt python sideset_init = pico.PIO.OUT_HIGH
15 .lang_opt python out_init     = pico.PIO.OUT_HIGH
16 .lang_opt python out_shiftdir = 1
17
18 .wrap_target
19 bitloop:
20     out x, 1       side 0 [T3 - 1] ; Side-set still takes place when instruction stalls
21     jmp !x do_zero side 1 [T1 - 1] ; Branch on the bit we shifted out. Positive pulse
22 do_one:
23     jmp  bitloop   side 1 [T2 - 1] ; Continue driving high, for a long pulse
24 do_zero:
25     nop            side 0 [T2 - 1] ; Or drive low, for a short pulse
26 .wrap
27
28 % c-sdk {
29 #include "hardware/clocks.h"
```

```
30
31 static inline void ws2812_program_init(PIO pio, uint sm, uint offset, uint pin, float freq,
   bool rgbw) {
32
33     pio_gpio_init(pio, pin);
34     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
35
36     pio_sm_config c = ws2812_program_get_default_config(offset);
37     sm_config_set_sideset_pins(&c, pin);
38     sm_config_set_out_shift(&c, false, true, rgbw ? 32 : 24);
39     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
40
41     int cycles_per_bit = ws2812_T1 + ws2812_T2 + ws2812_T3;
42     float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit);
43     sm_config_set_clkdiv(&c, div);
44
45     pio_sm_init(pio, sm, offset, &c);
46     pio_sm_set_enabled(pio, sm, true);
47 }
48 %}
49
50 .program ws2812_parallel
51
52 .define public T1 2
53 .define public T2 5
54 .define public T3 3
55
56 .wrap_target
57     out x, 32
58     mov pins, !null [T1-1]
59     mov pins, x     [T2-1]
60     mov pins, null  [T3-2]
61 .wrap
62
63 % c-sdk {
64 #include "hardware/clocks.h"
65
66 static inline void ws2812_parallel_program_init(PIO pio, uint sm, uint offset, uint
   pin_base, uint pin_count, float freq) {
67     for(uint i=pin_base; i<pin_base+pin_count; i++) {
68         pio_gpio_init(pio, i);
69     }
70     pio_sm_set_consecutive_pindirs(pio, sm, pin_base, pin_count, true);
71
72     pio_sm_config c = ws2812_parallel_program_get_default_config(offset);
73     sm_config_set_out_shift(&c, true, true, 32);
74     sm_config_set_out_pins(&c, pin_base, pin_count);
75     sm_config_set_set_pins(&c, pin_base, pin_count);
76     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
77
78     int cycles_per_bit = ws2812_parallel_T1 + ws2812_parallel_T2 + ws2812_parallel_T3;
79     float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit);
80     sm_config_set_clkdiv(&c, div);
81
82     pio_sm_init(pio, sm, offset, &c);
83     pio_sm_set_enabled(pio, sm, true);
84 }
85 %}
```

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/tree/master/pio/ws2812/generated/ws2812.pio.h Lines 1 - 112

```
1  // -------------------------------------------------- //
2  // This file is autogenerated by pioasm; do not edit! //
3  // -------------------------------------------------- //
4
5  #if !PICO_NO_HARDWARE
6  #include "hardware/pio.h"
7  #endif
8
9  // ------ //
10 // ws2812 //
11 // ------ //
12
13 #define ws2812_wrap_target 0
14 #define ws2812_wrap 3
15
16 #define ws2812_T1 2
17 #define ws2812_T2 5
18 #define ws2812_T3 3
19
20 static const uint16_t ws2812_program_instructions[] = {
21             //     .wrap_target
22     0x6221, //  0: out    x, 1            side 0 [2]
23     0x1123, //  1: jmp    !x, 3           side 1 [1]
24     0x1400, //  2: jmp    0               side 1 [4]
25     0xa442, //  3: nop                    side 0 [4]
26             //     .wrap
27 };
28
29 #if !PICO_NO_HARDWARE
30 static const struct pio_program ws2812_program = {
31     .instructions = ws2812_program_instructions,
32     .length = 4,
33     .origin = -1,
34 };
35
36 static inline pio_sm_config ws2812_program_get_default_config(uint offset) {
37     pio_sm_config c = pio_get_default_sm_config();
38     sm_config_set_wrap(&c, offset + ws2812_wrap_target, offset + ws2812_wrap);
39     sm_config_set_sideset(&c, 1, false, false);
40     return c;
41 }
42
43 #include "hardware/clocks.h"
44 static inline void ws2812_program_init(PIO pio, uint sm, uint offset, uint pin, float freq,
   bool rgbw) {
45     pio_gpio_init(pio, pin);
46     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
47     pio_sm_config c = ws2812_program_get_default_config(offset);
```

```
48      sm_config_set_sideset_pins(&c, pin);
49      sm_config_set_out_shift(&c, false, true, rgbw ? 32 : 24);
50      sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
51      int cycles_per_bit = ws2812_T1 + ws2812_T2 + ws2812_T3;
52      float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit);
53      sm_config_set_clkdiv(&c, div);
54      pio_sm_init(pio, sm, offset, &c);
55      pio_sm_set_enabled(pio, sm, true);
56  }
57
58  #endif
59
60  // --------------- //
61  // ws2812_parallel //
62  // --------------- //
63
64  #define ws2812_parallel_wrap_target 0
65  #define ws2812_parallel_wrap 3
66
67  #define ws2812_parallel_T1 2
68  #define ws2812_parallel_T2 5
69  #define ws2812_parallel_T3 3
70
71  static const uint16_t ws2812_parallel_program_instructions[] = {
72              //     .wrap_target
73      0x6020, //  0: out    x, 32
74      0xa10b, //  1: mov    pins, !null           [1]
75      0xa401, //  2: mov    pins, x               [4]
76      0xa103, //  3: mov    pins, null            [1]
77              //     .wrap
78  };
79
80  #if !PICO_NO_HARDWARE
81  static const struct pio_program ws2812_parallel_program = {
82      .instructions = ws2812_parallel_program_instructions,
83      .length = 4,
84      .origin = -1,
85  };
86
87  static inline pio_sm_config ws2812_parallel_program_get_default_config(uint offset) {
88      pio_sm_config c = pio_get_default_sm_config();
89      sm_config_set_wrap(&c, offset + ws2812_parallel_wrap_target, offset +
    ws2812_parallel_wrap);
90      return c;
91  }
92
93  #include "hardware/clocks.h"
94  static inline void ws2812_parallel_program_init(PIO pio, uint sm, uint offset, uint
    pin_base, uint pin_count, float freq) {
95      for(uint i=pin_base; i<pin_base+pin_count; i++) {
96          pio_gpio_init(pio, i);
97      }
98      pio_sm_set_consecutive_pindirs(pio, sm, pin_base, pin_count, true);
99      pio_sm_config c = ws2812_parallel_program_get_default_config(offset);
100     sm_config_set_out_shift(&c, true, true, 32);
101     sm_config_set_out_pins(&c, pin_base, pin_count);
102     sm_config_set_set_pins(&c, pin_base, pin_count);
103     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
104     int cycles_per_bit = ws2812_parallel_T1 + ws2812_parallel_T2 + ws2812_parallel_T3;
105     float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit);
106     sm_config_set_clkdiv(&c, div);
107     pio_sm_init(pio, sm, offset, &c);
108     pio_sm_set_enabled(pio, sm, true);
```

```
109 }
110
111 #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/tree/master/pio/ws2812/generated/ws2812.py Lines 1 - 46

```
 1 # ----------------------------------------------- #
 2 # This file is autogenerated by pioasm; do not edit! #
 3 # ----------------------------------------------- #
 4
 5 import rp2
 6 from machine import Pin
 7 # ------ #
 8 # ws2812 #
 9 # ------ #
10
11 ws2812_T1 = 2
12 ws2812_T2 = 5
13 ws2812_T3 = 3
14
15 @rp2.asm_pio(sideset_init=pico.PIO.OUT_HIGH, out_init=pico.PIO.OUT_HIGH, out_shiftdir=1)
16 def ws2812():
17     wrap_target()
18     label("0")
19     out(x, 1)               .side(0) [2]  # 0
20     jmp(not_x, "3")         .side(1) [1]  # 1
21     jmp("0")                .side(1) [4]  # 2
22     label("3")
23     nop()                   .side(0) [4]  # 3
24     wrap()
25
26
27
28 # --------------- #
29 # ws2812_parallel #
30 # --------------- #
31
32 ws2812_parallel_T1 = 2
33 ws2812_parallel_T2 = 5
34 ws2812_parallel_T3 = 3
35
36 @rp2.asm_pio()
```

```
37 def ws2812_parallel():
38     wrap_target()
39     out(x, 32)                          # 0
40     mov(pins, not null)             [1] # 1
41     mov(pins, x)                    [4] # 2
42     mov(pins, null)                 [1] # 3
43     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-bit hexadecimal number.

Given:

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/master/pio/squarewave/squarewave.pio Lines 1 - 13

```
 1 ;
 2 ; Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
 3 ;
 4 ; SPDX-License-Identifier: BSD-3-Clause
 5 ;
 6
 7 .program squarewave
 8     set pindirs, 1   ; Set pin to output
 9 again:
10     set pins, 1 [1]  ; Drive pin high and then delay for one cycle
11     set pins, 0      ; Drive pin low
12     jmp again        ; Set PC to label `again`
```

The *hex* output produces:

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/master/pio/squarewave/generated/squarewave.hex Lines 1 - 4

```
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 **RP2040 Datasheet**. You are encouraged to read the PIO chapter of the datasheet to get the full context for what these instructions do.

3.4.1. Summary

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

Table 9. 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				

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

3.4.2. JMP

3.4.2.1. 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				

3.4.2.2. 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, post-decrement
  - 011: `!Y`: scratch Y zero
  - 100: `Y--`: scratch Y non-zero, post-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`. 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.

3.4.2.3. 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`

3.4.3. WAIT

3.4.3.1. 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				

3.4.3.2. 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.
  - 10: `IRQ`: PIO IRQ flag selected by `Index`
  - 11: 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: if the MSB is set, 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.

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

3.4.3.3. 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

3.4.4. IN

3.4.4.1. 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				

3.4.4.2. 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 3, PINS` 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.

3.4.4.3. 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)

3.4.5. OUT

3.4.5.1. 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				

3.4.5.2. 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.

3.4.5.3. 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)

3.4.6. PUSH

3.4.6.1. 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

3.4.6.2. 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.

3.4.6.3. 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.

3.4.7. PULL

3.4.7.1. 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

3.4.7.2. 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.

3.4.7.3. 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.

3.4.8. MOV

3.4.8.1. 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		

3.4.8.2. 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: Reserved
  - 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 3.4.5), 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.

3.4.8.3. 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.

3.4.9. IRQ

3.4.9.1. 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	Index				

3.4.9.2. 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:
    - The 3 LSBs specify an IRQ index from 0-7. This IRQ flag will be set/cleared depending on the Clear bit.
    - If the MSB is set, 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 raise flag 3, and a flag value of 0x13 will raise flag 1.

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`.

The modulo addition bit 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.

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

3.4.9.3. 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

3.4.10. SET

3.4.10.1. 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				

3.4.10.2. 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.

3.4.10.3. 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

4.1. Hardware APIs

This group of libraries provides a thin and efficient C API / abstractions to access the RP2040 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.
hardware_clocks	Clock Management API.
hardware_divider	Low-level hardware-divider access.
hardware_dma	DMA Controller API.
channel_config	DMA channel configuration.
hardware_flash	Low level flash programming and erase API.
hardware_gpio	General Purpose Input/Output (GPIO) API.
hardware_i2c	I2C Controller API.
hardware_interp	Hardware Interpolator API.
interp_config	Interpolator configuration.
hardware_irq	Hardware interrupt handling.
hardware_pio	Programmable I/O (PIO) API.
sm_config	PIO state machine configuration.
hardware_pll	Phase Locked Loop control APIs.
hardware_pwm	Hardware Pulse Width Modulation (PWM) API.
hardware_resets	Hardware Reset API.
hardware_rtc	Hardware Real Time Clock API.
hardware_spi	Hardware SPI API.
hardware_sync	Low level hardware spin-lock, barrier and processor event API.
hardware_timer	Low-level hardware timer API.
hardware_uart	Hardware UART API.
hardware_vreg	Voltage Regulation API.
hardware_watchdog	Hardware Watchdog Timer API.
hardware_xosc	Crystal Oscillator (XOSC) API.

4.1.1. hardware_adc

Analog to Digital Converter (ADC) API.

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

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

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

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

Temperature sensor values can be approximated in centigrade as:

T = 27 - (ADC_Voltage - 0.706)/0.001721

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

Example

```
#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);
    }
}
```

4.1.1.1. Function List

- `void adc_init (void)`
- `static void adc_gpio_init (uint gpio)`
- `static void adc_select_input (uint input)`

- `static void adc_set_round_robin (uint input_mask)`
- `static void adc_set_temp_sensor_enabled (bool enable)`
- `static uint16_t adc_read (void)`
- `static void adc_run (bool run)`
- `static void adc_set_clkdiv (float clkdiv)`
- `static void adc_fifo_setup (bool en, bool dreq_en, uint16_t dreq_thresh, bool err_in_fifo, bool byte_shift)`
- `static bool adc_fifo_is_empty (void)`
- `static uint8_t adc_fifo_get_level (void)`
- `static uint16_t adc_fifo_get (void)`
- `static uint16_t adc_fifo_get_blocking (void)`
- `static void adc_fifo_drain (void)`
- `static void adc_irq_set_enabled (bool enabled)`

4.1.1.2. Function Documentation

4.1.1.2.1. adc_fifo_drain

`static void adc_fifo_drain (void)`

Drain the ADC FIFO.

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

4.1.1.2.2. adc_fifo_get

`static uint16_t adc_fifo_get (void)`

Get ADC result from FIFO.

Pops the latest result from the ADC FIFO.

4.1.1.2.3. adc_fifo_get_blocking

`static uint16_t adc_fifo_get_blocking (void)`

Wait for the ADC FIFO to have data.

Blocks until data is present in the FIFO

4.1.1.2.4. adc_fifo_get_level

`static uint8_t adc_fifo_get_level (void)`

Get number of entries in the ADC FIFO.

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

4.1.1.2.5. adc_fifo_is_empty

`static bool adc_fifo_is_empty (void)`

Check FIFO empty state.

Returns

- Returns true if the fifo is empty

4.1.1.2.6. adc_fifo_setup

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

Setup the ADC FIFO.

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

Parameters

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

4.1.1.2.7. adc_gpio_init

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

Initialise the gpio for use as an ADC pin.

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

Parameters

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

4.1.1.2.8. adc_init

```
void adc_init (void)
```

Initialise the ADC HW.

4.1.1.2.9. adc_irq_set_enabled

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

Enable/Disable ADC interrupts.

Parameters

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

4.1.1.2.10. adc_read

```
static uint16_t adc_read (void)
```

Perform a single conversion.

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

Returns

- Result of the conversion.

4.1.1.2.11. adc_run

```
static void adc_run (bool run)
```

Enable or disable free-running sampling mode.

Parameters

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

4.1.1.2.12. adc_select_input

```
static void adc_select_input (uint input)
```

ADC input select.

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

Parameters

- `input` Input to select.

4.1.1.2.13. adc_set_clkdiv

```
static void adc_set_clkdiv (float clkdiv)
```

Set the ADC Clock divisor.

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

Parameters

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

4.1.1.2.14. adc_set_round_robin

```
static void adc_set_round_robin (uint input_mask)
```

Round Robin sampling selector.

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

Parameters

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

4.1.1.2.15. adc_set_temp_sensor_enabled

```
static void adc_set_temp_sensor_enabled (bool enable)
```

Enable the onboard temperature sensor.

Parameters

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

4.1.2. hardware_base

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

hardware_base defines the low level types and access functions for memory mapped hardware registers. It is 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.

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

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

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

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

4.1.2.1. Function List

- `static void hw_set_bits (io_rw_32 *addr, uint32_t mask)`
- `static void hw_clear_bits (io_rw_32 *addr, uint32_t mask)`
- `static void hw_xor_bits (io_rw_32 *addr, uint32_t mask)`
- `static void hw_write_masked (io_rw_32 *addr, uint32_t values, uint32_t write_mask)`

4.1.2.2. Function Documentation

4.1.2.2.1. hw_clear_bits

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

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

Parameters

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

4.1.2.2.2. hw_set_bits

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

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

Parameters

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

4.1.2.2.3. hw_write_masked

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

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

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

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

Parameters

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

4.1.2.2.4. hw_xor_bits

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

Atomically flip the specified bits in a HW register.

Parameters

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

4.1.3. hardware_claim

Lightweight hardware resource management.

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

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

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

Facilities are providing for:

- Claiming resources (and asserting if they are already claimed)
- Freeing (unclaiming) resources
- Finding unused resources

4.1.3.1. Function List

- `void hw_claim_or_assert (uint8_t *bits, uint bit_index, const char *message)`
- `int hw_claim_unused_from_range (uint8_t *bits, bool required, uint bit_lsb, uint bit_msb, const char *message)`
- `bool hw_is_claimed (uint8_t *bits, uint bit_index)`
- `void hw_claim_clear (uint8_t *bits, uint bit_index)`
- `uint32_t hw_claim_lock ()`
- `void hw_claim_unlock (uint32_t token)`

4.1.3.2. Function Documentation

4.1.3.2.1. 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)

4.1.3.2.2. hw_claim_lock

```
uint32_t hw_claim_lock ()
```

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

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

Returns

- a token to pass to `hw_claim_unlock()`

4.1.3.2.3. 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

4.1.3.2.4. hw_claim_unlock

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

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

Parameters

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

4.1.3.2.5. 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

4.1.3.2.6. hw_is_claimed

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

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

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

Parameters

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

Returns

- true if the resource is claimed

4.1.4. hardware_clocks

Clock Management API.

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 datasheet for more details on the RP2040 clocks.

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

Main Clock Sources

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

Auxiliary Clock Sources

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

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

Example// hello_48MHz.c

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

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

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

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

int main() {
    stdio_init_all();

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

    measure_freqs();

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

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

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

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

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

    return 0;
}
```

4.1.4.1. Enumerations

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

4.1.4.2. Typedefs

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

4.1.4.3. Function List

- `void clocks_init ()`
- `bool clock_configure (enum clock_index clk_index, uint32_t src, uint32_t auxsrc, uint32_t src_freq, uint32_t freq)`
- `void clock_stop (enum clock_index clk_index)`
- `uint32_t clock_get_hz (enum clock_index clk_index)`
- `uint32_t frequency_count_khz (uint src)`
- `void clock_set_reported_hz (enum clock_index clk_index, uint hz)`
- `void clocks_enable_resus (resus_callback_t resus_callback)`
- `void clock_gpio_init (uint gpio, uint src, uint div)`
- `bool clock_configure_gpin (enum clock_index clk_index, uint gpio, uint32_t src_freq, uint32_t freq)`

4.1.4.4. Function Documentation

4.1.4.4.1. clock_configure

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

Configure the specified clock.

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

Parameters

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

4.1.4.4.2. clock_configure_gpin

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

```
    uint32_t freq)
```

Configure a clock to come from a gpio input.

Parameters

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

4.1.4.4.3. clock_get_hz

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

Get the current frequency of the specified clock.

Parameters

- `clk_index` Clock

Returns

- Clock frequency in Hz

4.1.4.4.4. clock_gpio_init

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

Output an optionally divided clock to the specified gpio pin.

Parameters

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

4.1.4.4.5. clock_set_reported_hz

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

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

See also

- clock_get_hz

4.1.4.4.6. clock_stop

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

Stop the specified clock.

Parameters

- `clk_index` The clock to stop

4.1.4.4.7. clocks_enable_resus

`void clocks_enable_resus (resus_callback_t resus_callback)`

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

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

Parameters

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

4.1.4.4.8. clocks_init

`void clocks_init ()`

Initialise the clock hardware.

Must be called before any other clock function.

4.1.4.4.9. frequency_count_khz

`uint32_t frequency_count_khz (uint src)`

Measure a clocks frequency using the Frequency counter.

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

4.1.5. hardware_divider

Low-level hardware-divider access.

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

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

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

See also

- pico_divider

Example

```
1 #include <stdio.h>
2 #include "pico/stdlib.h"
3 #include "hardware/divider.h"
4
5 // tag::hello_divider[]
6 int main() {
7     stdio_init_all();
8     printf("Hello, divider!\n");
9
```

```
10      // This is the basic hardware divider function
11      int32_t dividend = 123456;
12      int32_t divisor = -321;
13      divmod_result_t result = hw_divider_divmod_s32(dividend, divisor);
14
15      printf("%d/%d = %d remainder %d\n", dividend, divisor, to_quotient_s32(result),
    to_remainder_s32(result));
16
17      // Is it right?
18
19      printf("Working backwards! Result %d should equal %d!\n\n",
20             to_quotient_s32(result) * divisor + to_remainder_s32(result), dividend);
21
22      // This is the recommended unsigned fast divider for general use.
23      int32_t udividend = 123456;
24      int32_t udivisor = 321;
25      divmod_result_t uresult = hw_divider_divmod_u32(udividend, udivisor);
26
27      printf("%d/%d = %d remainder %d\n", udividend, udivisor, to_quotient_u32(uresult),
    to_remainder_u32(uresult));
28
29      // Is it right?
30
31      printf("Working backwards! Result %d should equal %d!\n\n",
32             to_quotient_u32(result) * divisor + to_remainder_u32(result), dividend);
33
34      // You can also do divides asynchronously. Divides will be complete after 8 cyles.
35
36      hw_divider_divmod_s32_start(dividend, divisor);
37
38      // Do something for 8 cycles!
39
40      // In this example, our results function will wait for completion.
41      // Use hw_divider_result_nowait() if you don't want to wait, but are sure you have delayed
    at least 8 cycles
42
43      result = hw_divider_result_wait();
44
45      printf("Async result %d/%d = %d remainder %d\n", dividend, divisor, to_quotient_s32
    (result),
46             to_remainder_s32(result));
47
48      // For a really fast divide, you can use the inlined versions... the / involves a function
    call as / always does
49      // when using the ARM AEABI, so if you really want the best performance use the inlined
    versions.
50      // Note that the / operator function DOES use the hardware divider by default, although
    you can change
51      // that behavior by calling pico_set_divider_implementation in the cmake build for your
    target.
52      printf("%d / %d = (by operator %d) (inlined %d)\n", dividend, divisor,
53             dividend / divisor, hw_divider_s32_quotient_inlined(dividend, divisor));
54
55      // Note however you must manually save/restore the divider state if you call the inlined
    methods from within an IRQ
56      // handler.
57      hw_divider_state_t state;
58      hw_divider_divmod_s32_start(dividend, divisor);
59      hw_divider_save_state(&state);
60
61      hw_divider_divmod_s32_start(123, 7);
62      printf("inner %d / %d = %d\n", 123, 7, hw_divider_s32_quotient_wait());
63
```

```
64     hw_divider_restore_state(&state);
65     int32_t tmp = hw_divider_s32_quotient_wait();
66     printf("outer divide %d / %d = %d\n", dividend, divisor, tmp);
67     return 0;
68 }
69 // end::hello_divider[]
```

4.1.5.1. Function List

- static void hw_divider_divmod_s32_start (int32_t a, int32_t b)
- static void hw_divider_divmod_u32_start (uint32_t a, uint32_t b)
- static void hw_divider_wait_ready ()
- static divmod_result_t hw_divider_result_nowait ()
- static divmod_result_t hw_divider_result_wait ()
- static uint32_t hw_divider_u32_quotient_wait ()
- static int32_t hw_divider_s32_quotient_wait ()
- static uint32_t hw_divider_u32_remainder_wait ()
- static int32_t hw_divider_s32_remainder_wait ()
- divmod_result_t hw_divider_divmod_s32 (int32_t a, int32_t b)
- divmod_result_t hw_divider_divmod_u32 (uint32_t a, uint32_t b)
- static uint32_t to_quotient_u32 (divmod_result_t r)
- static int32_t to_quotient_s32 (divmod_result_t r)
- static uint32_t to_remainder_u32 (divmod_result_t r)
- static int32_t to_remainder_s32 (divmod_result_t r)
- static uint32_t hw_divider_u32_quotient (uint32_t a, uint32_t b)
- static uint32_t hw_divider_u32_remainder (uint32_t a, uint32_t b)
- static int32_t hw_divider_quotient_s32 (int32_t a, int32_t b)
- static int32_t hw_divider_remainder_s32 (int32_t a, int32_t b)
- static void hw_divider_pause ()
- static uint32_t hw_divider_u32_quotient_inlined (uint32_t a, uint32_t b)
- static uint32_t hw_divider_u32_remainder_inlined (uint32_t a, uint32_t b)
- static int32_t hw_divider_s32_quotient_inlined (int32_t a, int32_t b)
- static int32_t hw_divider_s32_remainder_inlined (int32_t a, int32_t b)
- void hw_divider_save_state (hw_divider_state_t *dest)
- void hw_divider_restore_state (hw_divider_state_t *src)

4.1.5.2. Function Documentation

4.1.5.2.1. hw_divider_divmod_s32

divmod_result_t hw_divider_divmod_s32 (int32_t a,

```
int32_t b)
```

Do a signed HW divide and wait for result.

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

Parameters

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

Returns

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

4.1.5.2.2. hw_divider_divmod_s32_start

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

Start a signed asynchronous divide.

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

Parameters

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

4.1.5.2.3. hw_divider_divmod_u32

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

Do an unsigned HW divide and wait for result.

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

Parameters

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

Returns

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

4.1.5.2.4. hw_divider_divmod_u32_start

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

Start an unsigned asynchronous divide.

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

Parameters

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

4.1.5.2.5. hw_divider_pause

```
static void hw_divider_pause ()
```

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

4.1.5.2.6. hw_divider_quotient_s32

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

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

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

Parameters

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

Returns

- Quotient results of the divide

4.1.5.2.7. hw_divider_remainder_s32

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

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

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

Parameters

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

Returns

- Remainder results of the divide

4.1.5.2.8. hw_divider_restore_state

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

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

4.1.5.2.9. hw_divider_result_nowait

```
static divmod_result_t hw_divider_result_nowait ()
```

Return result of HW divide, nowait.

Returns

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

4.1.5.2.10. hw_divider_result_wait

```
static divmod_result_t hw_divider_result_wait ()
```

Return result of last asynchronous HW divide.

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

Returns

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

4.1.5.2.11. hw_divider_s32_quotient_inlined

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

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

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

Parameters

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

Returns

- Quotient result of the divide

4.1.5.2.12. hw_divider_s32_quotient_wait

```
static int32_t hw_divider_s32_quotient_wait ()
```

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

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

Returns

- Current signed quotient result.

4.1.5.2.13. hw_divider_s32_remainder_inlined

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

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

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

Parameters

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

Returns

- Remainder result of the divide

4.1.5.2.14. hw_divider_s32_remainder_wait

```
static int32_t hw_divider_s32_remainder_wait ()
```

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

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

Returns

- Current remainder results.

4.1.5.2.15. hw_divider_save_state

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

Save the calling cores hardware divider state.

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

Parameters

- `dest` the location to store the divider state

4.1.5.2.16. hw_divider_u32_quotient

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

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

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

Parameters

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

Returns

- Quotient results of the divide

4.1.5.2.17. hw_divider_u32_quotient_inlined

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

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

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

Parameters

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

Returns

- Quotient result of the divide

4.1.5.2.18. hw_divider_u32_quotient_wait

```
static uint32_t hw_divider_u32_quotient_wait ()
```

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

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

Returns

- Current unsigned quotient result.

4.1.5.2.19. hw_divider_u32_remainder

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

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

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

Parameters

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

Returns

- Remainder results of the divide

4.1.5.2.20. hw_divider_u32_remainder_inlined

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

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

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

Parameters

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

Returns

- Remainder result of the divide

4.1.5.2.21. hw_divider_u32_remainder_wait

```
static uint32_t hw_divider_u32_remainder_wait ()
```

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

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

Returns

- Current unsigned remainder result.

4.1.5.2.22. hw_divider_wait_ready

```
static void hw_divider_wait_ready ()
```

Wait for a divide to complete.

Wait for a divide to complete

4.1.5.2.23. to_quotient_s32

```
static int32_t to_quotient_s32 (divmod_result_t r)
```

Efficient extraction of signed quotient from 32p32 fixed point.

Parameters

- `r` 32p32 fixed point value.

Returns

- Unsigned quotient

4.1.5.2.24. to_quotient_u32

```
static uint32_t to_quotient_u32 (divmod_result_t r)
```

Efficient extraction of unsigned quotient from 32p32 fixed point.

Parameters

- `r` 32p32 fixed point value.

Returns

- Unsigned quotient

4.1.5.2.25. to_remainder_s32

```
static int32_t to_remainder_s32 (divmod_result_t r)
```

Efficient extraction of signed remainder from 32p32 fixed point.

Parameters

- `r` 32p32 fixed point value.

Returns

- Signed remainder

4.1.5.2.26. to_remainder_u32

```
static uint32_t to_remainder_u32 (divmod_result_t r)
```

Efficient extraction of unsigned remainder from 32p32 fixed point.

Parameters

- `r` 32p32 fixed point value.

Returns

- Unsigned remainder

4.1.6. hardware_dma

DMA Controller API.

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

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

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

4.1.6.1. Modules

- channel_config
  DMA channel configuration.

4.1.6.2. Enumerations

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

4.1.6.3. Function List

- void dma_channel_claim (uint channel)
- void dma_claim_mask (uint32_t channel_mask)
- void dma_channel_unclaim (uint channel)
- int dma_claim_unused_channel (bool required)
- static void dma_channel_set_config (uint channel, const dma_channel_config *config, bool trigger)
- static void dma_channel_set_read_addr (uint channel, const volatile void *read_addr, bool trigger)
- static void dma_channel_set_write_addr (uint channel, volatile void *write_addr, bool trigger)
- static void dma_channel_set_trans_count (uint channel, uint32_t trans_count, bool trigger)
- 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)
- static void dma_channel_transfer_from_buffer_now (uint channel, void *read_addr, uint32_t transfer_count)
- static void dma_channel_transfer_to_buffer_now (uint channel, void *write_addr, uint32_t transfer_count)
- static void dma_start_channel_mask (uint32_t chan_mask)
- static void dma_channel_start (uint channel)
- static void dma_channel_abort (uint channel)
- static void dma_channel_set_irq0_enabled (uint channel, bool enabled)
- static void dma_set_irq0_channel_mask_enabled (uint32_t channel_mask, bool enabled)
- static void dma_channel_set_irq1_enabled (uint channel, bool enabled)
- static void dma_set_irq1_channel_mask_enabled (uint32_t channel_mask, bool enabled)
- static bool dma_channel_is_busy (uint channel)
- static void dma_channel_wait_for_finish_blocking (uint channel)
- static void dma_sniffer_enable (uint channel, uint mode, bool force_channel_enable)
- static void dma_sniffer_set_byte_swap_enabled (bool swap)
- static void dma_sniffer_disable ()

4.1.6.4. Function Documentation

4.1.6.4.1. dma_channel_abort

static void dma_channel_abort (uint channel)

Stop a DMA transfer.

Function will only return once the DMA has stopped.

Parameters

- `channel` DMA channel

4.1.6.4.2. dma_channel_claim

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

Mark a dma channel as used.

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

Parameters

- `channel` the dma channel

4.1.6.4.3. dma_channel_configure

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

Configure all DMA parameters and optionally start transfer.

Parameters

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

4.1.6.4.4. dma_channel_is_busy

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

Check if DMA channel is busy.

Parameters

- `channel` DMA channel

Returns

- true if the channel is currently busy

4.1.6.4.5. dma_channel_set_config

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

Set a channel configuration.

Parameters

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

4.1.6.4.6. dma_channel_set_irq0_enabled

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

Enable single DMA channel interrupt 0.

Parameters

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

4.1.6.4.7. dma_channel_set_irq1_enabled

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

Enable single DMA channel interrupt 1.

Parameters

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

4.1.6.4.8. dma_channel_set_read_addr

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

Set the DMA initial read address.

Parameters

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

4.1.6.4.9. dma_channel_set_trans_count

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

Set the number of bus transfers the channel will do.

Parameters

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

4.1.6.4.10. dma_channel_set_write_addr

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

Set the DMA initial read address.

Parameters

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

4.1.6.4.11. dma_channel_start

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

Start a single DMA channel.

Parameters

- `channel` DMA channel

4.1.6.4.12. dma_channel_transfer_from_buffer_now

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

Start a DMA transfer from a buffer immediately.

Parameters

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

4.1.6.4.13. dma_channel_transfer_to_buffer_now

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

Start a DMA transfer to a buffer immediately.

Parameters

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

4.1.6.4.14. dma_channel_unclaim

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

Mark a dma channel as no longer used.

Method for cooperative claiming of hardware.

Parameters

- `channel` the dma channel to release

4.1.6.4.15. dma_channel_wait_for_finish_blocking

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

Wait for a DMA channel transfer to complete.

Parameters

- `channel` DMA channel

4.1.6.4.16. dma_claim_mask

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

Mark multiple dma channels as used.

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

Parameters

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

4.1.6.4.17. dma_claim_unused_channel

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

Claim a free dma channel.

Parameters

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

Returns

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

4.1.6.4.18. dma_set_irq0_channel_mask_enabled

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

Enable multiple DMA channels interrupt 0.

Parameters

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

4.1.6.4.19. dma_set_irq1_channel_mask_enabled

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

Enable multiple DMA channels interrupt 0.

Parameters

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

4.1.6.4.20. dma_sniffer_disable

```
static void dma_sniffer_disable ()
```

Disable the DMA sniffer.

4.1.6.4.21. dma_sniffer_enable

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

Enable the DMA sniffing targeting the specified channel.

The mode can be one of the following:

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

Parameters

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

4.1.6.4.22. dma_sniffer_set_byte_swap_enabled

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

Enable the Sniffer byte swap function.

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

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

Parameters

- `swap` Set true to enable byte swapping

4.1.6.4.23. dma_start_channel_mask

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

Start one or more channels simultaneously.

Parameters

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

4.1.7. channel_config

DMA channel configuration.

A DMA channel needs to be configured, these functions provide handy helpers to set up configuration structures. See dma_channel_config

4.1.7.1. Function List

- `static void channel_config_set_read_increment (dma_channel_config *c, bool incr)`
- `static void channel_config_set_write_increment (dma_channel_config *c, bool incr)`
- `static void channel_config_set_dreq (dma_channel_config *c, uint dreq)`
- `static void channel_config_set_chain_to (dma_channel_config *c, uint chain_to)`
- `static void channel_config_set_transfer_data_size (dma_channel_config *c, enum dma_channel_transfer_size size)`
- `static void channel_config_set_ring (dma_channel_config *c, bool write, uint size_bits)`
- `static void channel_config_set_bswap (dma_channel_config *c, bool bswap)`
- `static void channel_config_set_irq_quiet (dma_channel_config *c, bool irq_quiet)`
- `static void channel_config_set_enable (dma_channel_config *c, bool enable)`
- `static void channel_config_set_sniff_enable (dma_channel_config *c, bool sniff_enable)`
- `static dma_channel_config dma_channel_get_default_config (uint channel)`
- `static dma_channel_config dma_get_channel_config (uint channel)`
- `static uint32_t channel_config_get_ctrl_value (const dma_channel_config *config)`

4.1.7.2. Function Documentation

4.1.7.2.1. channel_config_get_ctrl_value

```
static uint32_t channel_config_get_ctrl_value (const dma_channel_config *config)
```

Get the raw configuration register from a channel configuration.

Parameters

- `config` Pointer to a config structure.

Returns

- Register content

4.1.7.2.2. channel_config_set_bswap

```
static void channel_config_set_bswap (dma_channel_config *c,
```

```
    bool bswap)
```

Set DMA byte swapping.

No effect for byte data, for halfword data, the two bytes of each halfword are swapped. For word data, the four bytes of each word are swapped to reverse their order.

Parameters

- `c` Pointer to channel configuration data
- `bswap` True to enable byte swapping

4.1.7.2.3. channel_config_set_chain_to

```
static void channel_config_set_chain_to (dma_channel_config *c,
    uint chain_to)
```

Set DMA channel completion channel.

When this channel completes, it will trigger the channel indicated by chain_to. Disable by setting chain_to to itself (the same channel)

Parameters

- `c` Pointer to channel configuration data
- `chain_to` Channel to trigger when this channel completes.

4.1.7.2.4. channel_config_set_dreq

```
static void channel_config_set_dreq (dma_channel_config *c,
    uint dreq)
```

Select a transfer request signal.

The channel uses the transfer request signal to pace its data transfer rate. Sources for TREQ signals are internal (TIMERS) or external (DREQ, a Data Request from the system). 0x0 to 0x3a → select DREQ n as TREQ 0x3b → Select Timer 0 as TREQ 0x3c → Select Timer 1 as TREQ 0x3d → Select Timer 2 as TREQ (Optional) 0x3e → Select Timer 3 as TREQ (Optional) 0x3f → Permanent request, for unpaced transfers.

Parameters

- `c` Pointer to channel configuration data
- `dreq` Source (see description)

4.1.7.2.5. channel_config_set_enable

```
static void channel_config_set_enable (dma_channel_config *c,
    bool enable)
```

Enable/Disable the DMA channel.

When false, the channel will ignore triggers, stop issuing transfers, and pause the current transfer sequence (i.e. BUSY will remain high if already high)

Parameters

- `c` Pointer to channel configuration data
- `enable` True to enable the DMA channel. When enabled, the channel will respond to triggering events, and start transferring data.

4.1.7.2.6. channel_config_set_irq_quiet

```
static void channel_config_set_irq_quiet (dma_channel_config *c,
      bool irq_quiet)
```

Set IRQ quiet mode.

In QUIET mode, the channel does not generate IRQs at the end of every transfer block. Instead, an IRQ is raised when NULL is written to a trigger register, indicating the end of a control block chain.

Parameters

- `c` Pointer to channel configuration data
- `irq_quiet` True to enable quiet mode, false to disable.

4.1.7.2.7. channel_config_set_read_increment

```
static void channel_config_set_read_increment (dma_channel_config *c,
      bool incr)
```

Set DMA channel read increment.

Parameters

- `c` Pointer to channel configuration data
- `incr` True to enable read address increments, if false, each read will be from the same address Usually disabled for peripheral to memory transfers

4.1.7.2.8. channel_config_set_ring

```
static void channel_config_set_ring (dma_channel_config *c,
      bool write,
      uint size_bits)
```

Set address wrapping parameters.

Size of address wrap region. If 0, don't wrap. For values n > 0, only the lower n bits of the address will change. This wraps the address on a (1 << n) byte boundary, facilitating access to naturally-aligned ring buffers. Ring sizes between 2 and 32768 bytes are possible (size_bits from 1 - 15)

0x0 → No wrapping.

Parameters

- `c` Pointer to channel configuration data
- `write` True to apply to write addresses, false to apply to read addresses
- `size_bits` 0 to disable wrapping. Otherwise the size in bits of the changing part of the address. Effectively wraps the address on a (1 << size_bits) byte boundary.

4.1.7.2.9. channel_config_set_sniff_enable

```
static void channel_config_set_sniff_enable (dma_channel_config *c,
      bool sniff_enable)
```

Enable access to channel by sniff hardware.

Sniff HW must be enabled and have this channel selected.

Parameters

- `c` Pointer to channel configuration data

- `sniff_enable` True to enable the Sniff HW access to this DMA channel.

4.1.7.2.10. channel_config_set_transfer_data_size

```
static void channel_config_set_transfer_data_size (dma_channel_config *c,
       enum dma_channel_transfer_size size)
```

Set the size of each DMA bus transfer.

Set the size of each bus transfer (byte/halfword/word). The read and write addresses advance by the specific amount (1/2/4 bytes) with each transfer.

Parameters

- `c` Pointer to channel configuration data
- `size` See enum for possible values.

4.1.7.2.11. channel_config_set_write_increment

```
static void channel_config_set_write_increment (dma_channel_config *c,
       bool incr)
```

Set DMA channel write increment.

Parameters

- `c` Pointer to channel configuration data
- `incr` True to enable write address increments, if false, each write will be to the same address Usually disabled for memory to peripheral transfers Usually disabled for memory to peripheral transfers

4.1.7.2.12. dma_channel_get_default_config

```
static dma_channel_config dma_channel_get_default_config (uint channel)
```

Get the default channel configuration for a given channel.

Setting	Default
Read Increment	true
Write Increment	false
DReq	DREQ_FORCE
Chain to	self
Data size	DMA_SIZE_32
Ring	write=false, size=0 (i.e. off)
Byte Swap	false
Quiet IRQs	false
Channel Enable	true
Sniff Enable	false

Parameters

- `channel` DMA channel

Returns

- the default configuration which can then be modified.

4.1.7.2.13. dma_get_channel_config

`static dma_channel_config dma_get_channel_config (uint channel)`

Get the current configuration for the specified channel.

Parameters

- `channel` DMA channel

Returns

- The current configuration as read from the HW register (not cached)

4.1.8. hardware_flash

Low level flash programming and erase API.

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

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

Example

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

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

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

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

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

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

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

    // Note that a whole number of sectors must be erased at a time.
    printf("\nErasing target region...\n");
```

```
34      flash_range_erase(FLASH_TARGET_OFFSET, FLASH_SECTOR_SIZE);
35      printf("Done. Read back target region:\n");
36      print_buf(flash_target_contents, FLASH_PAGE_SIZE);
37
38      printf("\nProgramming target region...\n");
39      flash_range_program(FLASH_TARGET_OFFSET, random_data, FLASH_PAGE_SIZE);
40      printf("Done. Read back target region:\n");
41      print_buf(flash_target_contents, FLASH_PAGE_SIZE);
42
43      bool mismatch = false;
44      for (int i = 0; i < FLASH_PAGE_SIZE; ++i) {
45          if (random_data[i] != flash_target_contents[i])
46              mismatch = true;
47      }
48      if (mismatch)
49          printf("Programming failed!\n");
50      else
51          printf("Programming successful!\n");
52  }
```

4.1.8.1. Function List

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

4.1.8.2. Function Documentation

4.1.8.2.1. flash_range_erase

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

Erase areas of flash.

Parameters

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

4.1.8.2.2. flash_range_program

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

Program flash.

Parameters

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

4.1.9. hardware_gpio

General Purpose Input/Output (GPIO) API.

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

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

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

Table 10. Function Select Table

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

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

4.1.9.1. Enumerations

- `enum gpio_function { GPIO_FUNC_XIP = 0, GPIO_FUNC_SPI = 1, GPIO_FUNC_UART = 2, GPIO_FUNC_I2C = 3, GPIO_FUNC_PWM = 4, GPIO_FUNC_SIO = 5, GPIO_FUNC_PIO0 = 6, GPIO_FUNC_PIO1 = 7, GPIO_FUNC_GPCK = 8, GPIO_FUNC_USB = 9, GPIO_FUNC_NULL = 0xf }`

GPIO function definitions for use with function select.

- 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.

4.1.9.2. Function List

- void gpio_set_function (uint gpio, enum gpio_function fn)
- void gpio_set_pulls (uint gpio, bool up, bool down)
- static void gpio_pull_up (uint gpio)
- static bool gpio_is_pulled_up (uint gpio)
- static void gpio_pull_down (uint gpio)
- static bool gpio_is_pulled_down (uint gpio)
- static void gpio_disable_pulls (uint gpio)
- void gpio_set_outover (uint gpio, uint value)
- void gpio_set_inover (uint gpio, uint value)
- void gpio_set_oeover (uint gpio, uint value)
- void gpio_set_input_enabled (uint gpio, bool enabled)
- void gpio_set_irq_enabled (uint gpio, uint32_t events, bool enabled)
- void gpio_set_irq_enabled_with_callback (uint gpio, uint32_t events, bool enabled, gpio_irq_callback_t callback)
- void gpio_set_dormant_irq_enabled (uint gpio, uint32_t events, bool enabled)
- void gpio_acknowledge_irq (uint gpio, uint32_t events)
- void gpio_init (uint gpio)
- void gpio_init_mask (uint gpio_mask)
- static bool gpio_get (uint gpio)
- static uint32_t gpio_get_all ()
- static void gpio_set_mask (uint32_t mask)
- static void gpio_clr_mask (uint32_t mask)
- static void gpio_xor_mask (uint32_t mask)
- static void gpio_put_masked (uint32_t mask, uint32_t value)
- static void gpio_put_all (uint32_t value)
- static void gpio_put (uint gpio, bool value)
- static void gpio_set_dir_out_masked (uint32_t mask)
- static void gpio_set_dir_in_masked (uint32_t mask)
- static void gpio_set_dir_masked (uint32_t mask, uint32_t value)
- static void gpio_set_dir_all_bits (uint32_t values)
- static void gpio_set_dir (uint gpio, bool out)
- static bool gpio_is_dir_out (uint gpio)
- static uint gpio_get_dir (uint gpio)

4.1.9.3. Function Documentation

4.1.9.3.1. gpio_acknowledge_irq

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

Acknowledge a GPIO interrupt.

Parameters

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

4.1.9.3.2. gpio_clr_mask

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

Drive low every GPIO appearing in mask.

Parameters

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

4.1.9.3.3. gpio_disable_pulls

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

Disable pulls on specified GPIO.

Parameters

- `gpio` GPIO number

4.1.9.3.4. gpio_get

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

Get state of a single specified GPIO.

Parameters

- `gpio` GPIO number

Returns

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

4.1.9.3.5. gpio_get_all

```
static uint32_t gpio_get_all ()
```

Get raw value of all GPIOs.

Returns

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

4.1.9.3.6. gpio_get_dir

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

Get a specific GPIO direction.

Parameters

- `gpio` GPIO number

Returns

- 1 for out, 0 for in

4.1.9.3.7. gpio_init

```
void gpio_init (uint gpio)
```

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

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

Parameters

- `gpio` GPIO number

4.1.9.3.8. gpio_init_mask

```
void gpio_init_mask (uint gpio_mask)
```

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

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

Parameters

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

4.1.9.3.9. gpio_is_dir_out

```
static bool gpio_is_dir_out (uint gpio)
```

Check if a specific GPIO direction is OUT.

Parameters

- `gpio` GPIO number

Returns

- true if the direction for the pin is OUT

4.1.9.3.10. gpio_is_pulled_down

```
static bool gpio_is_pulled_down (uint gpio)
```

Determine if the specified GPIO is pulled down.

Parameters

- `gpio` GPIO number

Returns

- true if the GPIO is pulled down

4.1.9.3.11. gpio_is_pulled_up

```
static bool gpio_is_pulled_up (uint gpio)
```

Determine if the specified GPIO is pulled up.

Parameters

- `gpio` GPIO number

Returns

- true if the GPIO is pulled up

4.1.9.3.12. gpio_pull_down

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

Set specified GPIO to be pulled down.

Parameters

- `gpio` GPIO number

4.1.9.3.13. gpio_pull_up

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

Set specified GPIO to be pulled up.

Parameters

- `gpio` GPIO number

4.1.9.3.14. gpio_put

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

Drive a single GPIO high/low.

Parameters

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

4.1.9.3.15. gpio_put_all

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

Drive all pins simultaneously.

Parameters

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

4.1.9.3.16. gpio_put_masked

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

Drive GPIO high/low depending on parameters.

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

Parameters

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

4.1.9.3.17. gpio_set_dir

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

Set a single GPIO direction.

Parameters

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

4.1.9.3.18. gpio_set_dir_all_bits

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

Set direction of all pins simultaneously.

Parameters

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

4.1.9.3.19. gpio_set_dir_in_masked

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

Set a number of GPIOs to input.

Parameters

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

4.1.9.3.20. gpio_set_dir_masked

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

Set multiple GPIO directions.

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

Parameters

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

4.1.9.3.21. gpio_set_dir_out_masked

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

Set a number of GPIOs to output.

Switch all GPIOs in "mask" to output

Parameters

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

4.1.9.3.22. gpio_set_dormant_irq_enabled

```
void gpio_set_dormant_irq_enabled (uint gpio,
      uint32_t events,
```

```
    bool enabled)
```

Enable dormant wake up interrupt for specified GPIO.

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

Parameters

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

4.1.9.3.23. gpio_set_function

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

Select GPIO function.

Parameters

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

4.1.9.3.24. gpio_set_inover

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

Select GPIO input override.

Parameters

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

4.1.9.3.25. gpio_set_input_enabled

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

Enable GPIO input.

Parameters

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

4.1.9.3.26. gpio_set_irq_enabled

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

Enable or disable interrupts for specified GPIO.

Events is a bitmask of the following:

bit	interrupt
0	Low level

bit	interrupt
1	High level
2	Edge low
3	Edge high

Parameters

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

4.1.9.3.27. gpio_set_irq_enabled_with_callback

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

Enable interrupts for specified GPIO.

Parameters

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

4.1.9.3.28. gpio_set_mask

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

Drive high every GPIO appearing in mask.

Parameters

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

4.1.9.3.29. gpio_set_oeover

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

Select GPIO output enable override.

Parameters

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

4.1.9.3.30. gpio_set_outover

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

Set GPIO output override.

Parameters

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

4.1.9.3.31. gpio_set_pulls

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

Select up and down pulls on specific GPIO.

Parameters

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

4.1.9.3.32. gpio_xor_mask

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

Toggle every GPIO appearing in mask.

Parameters

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

4.1.10. hardware_i2c

I2C Controller API.

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

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

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

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

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

Example

```
// Sweep through all 7-bit I2C addresses, to see if any slaves are present on
// the I2C bus. Print out a table that looks like this:
//
// I2C Bus Scan
//   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
// 0
// 1       @
// 2
// 3             @
// 4
// 5
```

```
// 6
// 7
//
// E.g. if slave addresses 0x12 and 0x34 were acknowledged.

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

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

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

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

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

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

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

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

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

4.1.10.1. Variables

- `i2c_inst_t i2c0_inst`

4.1.10.2. Function List

- `uint i2c_init (i2c_inst_t *i2c, uint baudrate)`
- `void i2c_deinit (i2c_inst_t *i2c)`
- `uint i2c_set_baudrate (i2c_inst_t *i2c, uint baudrate)`
- `void i2c_set_slave_mode (i2c_inst_t *i2c, bool slave, uint8_t addr)`
- `static uint i2c_hw_index (i2c_inst_t *i2c)`
- `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)`
- `int i2c_read_blocking_until (i2c_inst_t *i2c, uint8_t addr, uint8_t *dst, size_t len, bool nostop, absolute_time_t until)`
- `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)`
- `static int i2c_read_timeout_us (i2c_inst_t *i2c, uint8_t addr, uint8_t *dst, size_t len, bool nostop, uint timeout_us)`
- `int i2c_write_blocking (i2c_inst_t *i2c, uint8_t addr, const uint8_t *src, size_t len, bool nostop)`
- `int i2c_read_blocking (i2c_inst_t *i2c, uint8_t addr, uint8_t *dst, size_t len, bool nostop)`
- `static size_t i2c_get_write_available (i2c_inst_t *i2c)`
- `static size_t i2c_get_read_available (i2c_inst_t *i2c)`
- `static void i2c_write_raw_blocking (i2c_inst_t *i2c, const uint8_t *src, size_t len)`
- `static void i2c_read_raw_blocking (i2c_inst_t *i2c, uint8_t *dst, size_t len)`

4.1.10.3. Function Documentation

4.1.10.3.1. i2c_deinit

`void i2c_deinit (i2c_inst_t *i2c)`

Disable the I2C HW block.

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

Parameters

- `i2c` Either i2c0 or i2c1

4.1.10.3.2. i2c_get_read_available

`static size_t i2c_get_read_available (i2c_inst_t *i2c)`

Determine number of bytes received.

Parameters

- `i2c` Either i2c0 or i2c1

Returns

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

4.1.10.3.3. i2c_get_write_available

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

Determine non-blocking write space available.

Parameters

- `i2c` Either i2c0 or i2c1

Returns

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

4.1.10.3.4. i2c_hw_index

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

Convert I2c instance to hardware instance number.

Parameters

- `i2c` I2C instance

Returns

- Number of UART, 0 or 1.

4.1.10.3.5. i2c_init

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

Initialise the I2C HW block.

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

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

Parameters

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

Returns

- Actual set baudrate

4.1.10.3.6. i2c_read_blocking

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

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

Parameters

- `i2c` Either i2c0 or i2c1
- `addr` Address of device to read from

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

Returns

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

4.1.10.3.7. i2c_read_blocking_until

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

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

Parameters

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

Returns

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

4.1.10.3.8. i2c_read_raw_blocking

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

Write direct to TX FIFO.

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

Parameters

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

4.1.10.3.9. i2c_read_timeout_us

```
static int i2c_read_timeout_us (i2c_inst_t *i2c,
       uint8_t addr,
       uint8_t *dst,
       size_t len,
```

```
    bool nostop,
    uint timeout_us)
```

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

Parameters

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

Returns

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

4.1.10.3.10. i2c_set_baudrate

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

Set I2C baudrate.

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

Parameters

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

Returns

- Actual set baudrate

4.1.10.3.11. i2c_set_slave_mode

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

Set I2C port to slave mode.

Parameters

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

4.1.10.3.12. i2c_write_blocking

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

```
    bool nostop)
```

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

Parameters

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

Returns

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

4.1.10.3.13. i2c_write_blocking_until

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

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

Parameters

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

Returns

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

4.1.10.3.14. i2c_write_raw_blocking

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

Write direct to TX FIFO.

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

Parameters

- `i2c` Either i2c0 or i2c1

- `src` Data to send
- `len` Number of bytes to send

4.1.10.3.15. i2c_write_timeout_us

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

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

Parameters

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

Returns

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

4.1.11. hardware_interp

Hardware Interpolator API.

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

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

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

4.1.11.1. Modules

- `interp_config`
  Interpolator configuration.

4.1.11.2. Function List

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

- `void interp_claim_lane_mask (interp_hw_t *interp, uint lane_mask)`
- `void interp_unclaim_lane (interp_hw_t *interp, uint lane)`
- `static void interp_set_force_bits (interp_hw_t *interp, uint lane, uint bits)`
- `void interp_save (interp_hw_t *interp, interp_hw_save_t *saver)`
- `void interp_restore (interp_hw_t *interp, interp_hw_save_t *saver)`
- `static void interp_set_base (interp_hw_t *interp, uint lane, uint32_t val)`
- `static uint32_t interp_get_base (interp_hw_t *interp, uint lane)`
- `static void interp_set_base_both (interp_hw_t *interp, uint32_t val)`
- `static void interp_set_accumulator (interp_hw_t *interp, uint lane, uint32_t val)`
- `static uint32_t interp_get_accumulator (interp_hw_t *interp, uint lane)`
- `static uint32_t interp_pop_lane_result (interp_hw_t *interp, uint lane)`
- `static uint32_t interp_peek_lane_result (interp_hw_t *interp, uint lane)`
- `static uint32_t interp_pop_full_result (interp_hw_t *interp)`
- `static uint32_t interp_peek_full_result (interp_hw_t *interp)`
- `static void interp_add_accumulater (interp_hw_t *interp, uint lane, uint32_t val)`
- `static uint32_t interp_get_raw (interp_hw_t *interp, uint lane)`

4.1.11.3. Function Documentation

4.1.11.3.1. interp_add_accumulater

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

Add to accumulator.

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

Parameters

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

Returns

- The content of the FULL register

4.1.11.3.2. 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.

4.1.11.3.3. 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)

4.1.11.3.4. interp_get_accumulator

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

Gets the content of the interpolator accumulator register by lane.

Parameters

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

Returns

- The current content of the register

4.1.11.3.5. interp_get_base

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

Gets the content of interpolator base register by lane.

Parameters

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

Returns

- The current content of the lane base register

4.1.11.3.6. interp_get_raw

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

Get raw lane value.

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

Parameters

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

Returns

- The raw shift/mask value

4.1.11.3.7. interp_peek_full_result

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

Read lane result.

Parameters

- `interp` Interpolator instance, interp0 or interp1.

Returns

- The content of the FULL register

4.1.11.3.8. interp_peek_lane_result

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

Read lane result.

Parameters

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

Returns

- The content of the lane result register

4.1.11.3.9. interp_pop_full_result

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

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

Parameters

- `interp` Interpolator instance, interp0 or interp1.

Returns

- The content of the FULL register

4.1.11.3.10. interp_pop_lane_result

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

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

Parameters

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

Returns

- The content of the lane result register

4.1.11.3.11. 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

4.1.11.3.12. 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

4.1.11.3.13. interp_set_accumulator

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

Sets the interpolator accumulator register by lane.

Parameters

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

4.1.11.3.14. interp_set_base

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

Sets the interpolator base register by lane.

Parameters

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

4.1.11.3.15. interp_set_base_both

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

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

4.1.11.3.16. interp_set_force_bits

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

Directly set the force bits on a specified lane.

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

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

Parameters

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

4.1.11.3.17. interp_unclaim_lane

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

Release a previously claimed interpolator lane.

Parameters

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

4.1.12. interp_config

Interpolator configuration.

Each interpolator needs to be configured, these functions provide handy helpers to set up configuration structures.

4.1.12.1. Function List

- `static void interp_config_set_shift (interp_config *c, uint shift)`
- `static void interp_config_set_mask (interp_config *c, uint mask_lsb, uint mask_msb)`
- `static void interp_config_set_cross_input (interp_config *c, bool cross_input)`
- `static void interp_config_set_cross_result (interp_config *c, bool cross_result)`
- `static void interp_config_set_signed (interp_config *c, bool _signed)`
- `static void interp_config_set_add_raw (interp_config *c, bool add_raw)`

- `static void interp_config_set_blend (interp_config *c, bool blend)`
- `static void interp_config_set_clamp (interp_config *c, bool clamp)`
- `static void interp_config_set_force_bits (interp_config *c, uint bits)`
- `static interp_config interp_default_config ()`
- `static void interp_set_config (interp_hw_t *interp, uint lane, interp_config *config)`

4.1.12.2. Function Documentation

4.1.12.2.1. interp_config_set_add_raw

```
static void interp_config_set_add_raw (interp_config *c,
       bool add_raw)
```

Set raw add option.

When enabled, mask + shift is bypassed for LANE0 result. This does not affect the FULL result.

Parameters

- `c` Pointer to interpolation config
- `add_raw` If true, enable raw add option.

4.1.12.2.2. interp_config_set_blend

```
static void interp_config_set_blend (interp_config *c,
       bool blend)
```

Set blend mode.

If enabled, LANE1 result is a linear interpolation between BASE0 and BASE1, controlled by the 8 LSBs of lane 1 shift and mask value (a fractional number between 0 and 255/256ths)

LANE0 result does not have BASE0 added (yields only the 8 LSBs of lane 1 shift+mask value)

FULL result does not have lane 1 shift+mask value added (BASE2 + lane 0 shift+mask)

LANE1 SIGNED flag controls whether the interpolation is signed or unsig

Parameters

- `c` Pointer to interpolation config
- `blend` Set true to enable blend mode.

4.1.12.2.3. interp_config_set_clamp

```
static void interp_config_set_clamp (interp_config *c,
       bool clamp)
```

Set interpolator clamp mode (Interpolator 1 only)

Only present on INTERP1 on each core. If CLAMP mode is enabled:

Parameters

- `c` Pointer to interpolation config
- `clamp` Set true to enable clamp mode

4.1.12.2.4. interp_config_set_cross_input

```
static void interp_config_set_cross_input (interp_config *c,
       bool cross_input)
```

Enable cross input.

Allows feeding of the accumulator content from the other lane back in to this lanes shift+mask hardware. This will take effect even if the interp_config_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.

4.1.12.2.5. interp_config_set_cross_result

```
static void interp_config_set_cross_result (interp_config *c,
       bool cross_result)
```

Enable cross results.

Allows feeding of the other lane's result into this lane's accumulator on a POP operation.

Parameters

- `c` Pointer to interpolation config
- `cross_result` If true, enables the cross result

4.1.12.2.6. interp_config_set_force_bits

```
static void interp_config_set_force_bits (interp_config *c,
       uint bits)
```

Set interpolator Force bits.

ORed into bits 29:28 of the lane result presented to the processor on the bus.

No effect on the internal 32-bit datapath. Handy for using a lane to generate sequence of pointers into flash or SRAM

Parameters

- `c` Pointer to interpolation config
- `bits` Sets the force bits to that specified. Range 0-3 (two bits)

4.1.12.2.7. interp_config_set_mask

```
static void interp_config_set_mask (interp_config *c,
       uint mask_lsb,
       uint mask_msb)
```

Set the interpolator mask range.

Sets the range of bits (least to most) that are allowed to pass through the interpolator

Parameters

- `c` Pointer to interpolation config
- `mask_lsb` The least significant bit allowed to pass
- `mask_msb` The most significant bit allowed to pass

4.1.12.2.8. interp_config_set_shift

```
static void interp_config_set_shift (interp_config *c,
      uint shift)
```

Set the interpolator shift value.

Sets the number of bits the accumulator is shifted before masking, on each iteration.

Parameters

- `c` Pointer to an interpolator config
- `shift` Number of bits

4.1.12.2.9. interp_config_set_signed

```
static void interp_config_set_signed (interp_config *c,
      bool _signed)
```

Set sign extension.

Enables signed mode, where the shifted and masked accumulator value is sign-extended to 32 bits before adding to BASE1, and LANE1 PEEK/POP results appear extended to 32 bits when read by processor.

Parameters

- `c` Pointer to interpolation config
- `_signed` If true, enables sign extension

4.1.12.2.10. interp_default_config

```
static interp_config interp_default_config ()
```

Get a default configuration.

Returns

- A default interpolation configuration

4.1.12.2.11. interp_set_config

```
static void interp_set_config (interp_hw_t *interp,
      uint lane,
      interp_config *config)
```

Send configuration to a lane.

If an invalid configuration is specified (ie a lane specific item is set on wrong lane), depending on setup this function can panic.

Parameters

- `interp` Interpolator instance, interp0 or interp1.
- `lane` The lane to set
- `config` Pointer to interpolation config

4.1.13. hardware_irq

Hardware interrupt handling.

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

Interrupts are identified by a number from 0 to 31.

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

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

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

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

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

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

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

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

IRQ	Interrupt Source
0	TIMER_IRQ_0
1	TIMER_IRQ_1
2	TIMER_IRQ_2
3	TIMER_IRQ_3
4	PWM_IRQ_WRAP
5	USBCTRL_IRQ
6	XIP_IRQ
7	PIO0_IRQ_0
8	PIO0_IRQ_1
9	PIO1_IRQ_0
10	PIO1_IRQ_1
11	DMA_IRQ_0
12	DMA_IRQ_1
13	IO_IRQ_BANK0
14	IO_IRQ_QSPI
15	SIO_IRQ_PROC0
16	SIO_IRQ_PROC1
17	CLOCKS_IRQ
18	SPI0_IRQ

IRQ	Interrupt Source
19	SPI1_IRQ
20	UART0_IRQ
21	UART1_IRQ
22	ADC0_IRQ_FIFO
23	I2C0_IRQ
24	I2C1_IRQ
25	RTC_IRQ

4.1.13.1. Typedefs

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

4.1.13.2. Function List

- `void irq_set_priority (uint num, uint8_t hardware_priority)`
- `void irq_set_enabled (uint num, bool enabled)`
- `bool irq_is_enabled (uint num)`
- `void irq_set_mask_enabled (uint32_t mask, bool enabled)`
- `void irq_set_exclusive_handler (uint num, irq_handler_t handler)`
- `irq_handler_t irq_get_exclusive_handler (uint num)`
- `void irq_add_shared_handler (uint num, irq_handler_t handler, uint8_t order_priority)`
- `void irq_remove_handler (uint num, irq_handler_t handler)`
- `irq_handler_t irq_get_vtable_handler (uint num)`
- `static void irq_clear (uint int_num)`
- `void irq_set_pending (uint num)`

4.1.13.3. Function Documentation

4.1.13.3.1. irq_add_shared_handler

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

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

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

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

Parameters

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

See also

- irq_set_exclusive_handler

4.1.13.3.2. irq_clear

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

Clear a specific interrupt on the executing core.

Parameters

- `int_num` Interrupt number Interrupt Numbers

4.1.13.3.3. 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

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

See also

- irq_set_exclusive_handler

4.1.13.3.4. 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

4.1.13.3.5. 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

4.1.13.3.6. irq_remove_handler

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

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

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

Parameters

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

See also

- irq_set_exclusive_handler
- irq_add_shared_handler

4.1.13.3.7. 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

4.1.13.3.8. irq_set_exclusive_handler

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

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

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

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

Parameters

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

See also

- irq_add_shared_handler

4.1.13.3.9. irq_set_mask_enabled

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

Enable/disable multiple interrupts on the executing core.

Parameters

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

4.1.13.3.10. irq_set_pending

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

Force an interrupt to pending on the executing core.

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

Parameters

- `num` Interrupt number Interrupt Numbers

4.1.13.3.11. irq_set_priority

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

Set specified interrupts priority.

Parameters

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

4.1.14. hardware_pio

Programmable I/O (PIO) API.

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

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

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

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

4.1.14.1. Modules

- sm_config
  PIO state machine configuration.

4.1.14.2. Enumerations

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

4.1.14.3. Macros

- #define pio0 pio0_hw

4.1.14.4. Macros

- #define pio1 pio1_hw

4.1.14.5. Function List

- static void pio_sm_set_config (PIO pio, uint sm, const pio_sm_config *config)
- static uint pio_get_index (PIO pio)
- static void pio_gpio_init (PIO pio, uint pin)
- static uint pio_get_dreq (PIO pio, uint sm, bool is_tx)
- bool pio_can_add_program (PIO pio, const pio_program_t *program)
- bool pio_can_add_program_at_offset (PIO pio, const pio_program_t *program, uint offset)
- uint pio_add_program (PIO pio, const pio_program_t *program)
- void pio_add_program_at_offset (PIO pio, const pio_program_t *program, uint offset)
- void pio_remove_program (PIO pio, const pio_program_t *program, uint loaded_offset)
- void pio_clear_instruction_memory (PIO pio)
- void pio_sm_init (PIO pio, uint sm, uint initial_pc, const pio_sm_config *config)
- static void pio_sm_set_enabled (PIO pio, uint sm, bool enabled)
- static void pio_set_sm_mask_enabled (PIO pio, uint32_t mask, bool enabled)
- static void pio_sm_restart (PIO pio, uint sm)
- static void pio_restart_sm_mask (PIO pio, uint32_t mask)
- static void pio_sm_clkdiv_restart (PIO pio, uint sm)
- static void pio_clkdiv_restart_sm_mask (PIO pio, uint32_t mask)
- static void pio_enable_sm_mask_in_sync (PIO pio, uint32_t mask)
- static uint8_t pio_sm_get_pc (PIO pio, uint sm)
- static void pio_sm_exec (PIO pio, uint sm, uint instr)
- static bool pio_sm_is_exec_stalled (PIO pio, uint sm)

- `static void pio_sm_exec_wait_blocking (PIO pio, uint sm, uint instr)`
- `static void pio_sm_set_wrap (PIO pio, uint sm, uint wrap_target, uint wrap)`
- `static void pio_sm_put (PIO pio, uint sm, uint32_t data)`
- `static uint32_t pio_sm_get (PIO pio, uint sm)`
- `static bool pio_sm_is_rx_fifo_full (PIO pio, uint sm)`
- `static bool pio_sm_is_rx_fifo_empty (PIO pio, uint sm)`
- `static uint pio_sm_get_rx_fifo_level (PIO pio, uint sm)`
- `static bool pio_sm_is_tx_fifo_full (PIO pio, uint sm)`
- `static bool pio_sm_is_tx_fifo_empty (PIO pio, uint sm)`
- `static uint pio_sm_get_tx_fifo_level (PIO pio, uint sm)`
- `static void pio_sm_put_blocking (PIO pio, uint sm, uint32_t data)`
- `static uint32_t pio_sm_get_blocking (PIO pio, uint sm)`
- `void pio_sm_drain_tx_fifo (PIO pio, uint sm)`
- `static void pio_sm_set_clkdiv (PIO pio, uint sm, float div)`
- `static void pio_sm_set_clkdiv_int_frac (PIO pio, uint sm, uint16_t div_int, uint8_t div_frac)`
- `static void pio_sm_clear_fifos (PIO pio, uint sm)`
- `void pio_sm_set_pins (PIO pio, uint sm, uint32_t pin_values)`
- `void pio_sm_set_pins_with_mask (PIO pio, uint sm, uint32_t pin_values, uint32_t pin_mask)`
- `void pio_sm_set_pindirs_with_mask (PIO pio, uint sm, uint32_t pin_dirs, uint32_t pin_mask)`
- `void pio_sm_set_consecutive_pindirs (PIO pio, uint sm, uint pin_base, uint pin_count, bool is_out)`
- `void pio_sm_claim (PIO pio, uint sm)`
- `void pio_claim_sm_mask (PIO pio, uint sm_mask)`
- `void pio_sm_unclaim (PIO pio, uint sm)`
- `int pio_claim_unused_sm (PIO pio, bool required)`

4.1.14.6. Function Documentation

4.1.14.6.1. pio_add_program

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

Attempt to load the program, panicking if not possible.

Parameters

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

Returns

- the instruction memory offset the program is loaded at

See also

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

4.1.14.6.2. pio_add_program_at_offset

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

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

Parameters

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

Returns

- the instruction memory offset the program is loaded at

See also

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

4.1.14.6.3. pio_can_add_program

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

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

Parameters

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

Returns

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

4.1.14.6.4. pio_can_add_program_at_offset

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

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

Parameters

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

Returns

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

4.1.14.6.5. pio_claim_sm_mask

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

Mark multiple state machines as used.

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

Parameters

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

4.1.14.6.6. pio_claim_unused_sm

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

Claim a free state machine on a PIO instance.

Parameters

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

Returns

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

4.1.14.6.7. pio_clear_instruction_memory

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

Clears all of a PIO instance's instruction memory.

Parameters

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

4.1.14.6.8. pio_clkdiv_restart_sm_mask

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

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

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

Parameters

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

4.1.14.6.9. pio_enable_sm_mask_in_sync

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

Enable multiple PIO state machines synchronizing their clock dividers.

Parameters

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

4.1.14.6.10. pio_get_dreq

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

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

Parameters

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

4.1.14.6.11. pio_get_index

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

Return the instance number of a PIO instance.

Parameters

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

Returns

- the PIO instance number (either 0 or 1)

4.1.14.6.12. pio_gpio_init

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

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

Parameters

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

4.1.14.6.13. pio_remove_program

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

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

Parameters

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

4.1.14.6.14. pio_restart_sm_mask

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

Restart multiple state machine with a known state.

This method clears the ISR, shift counters, clock divider counter pin write flags, delay counter, latched EXEC instruction,

and IRQ wait condition.

Parameters

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

4.1.14.6.15. pio_set_sm_mask_enabled

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

Enable or disable multiple PIO state machines.

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

Parameters

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

See also

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

4.1.14.6.16. pio_sm_claim

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

Mark a state machine as used.

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

Parameters

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

4.1.14.6.17. pio_sm_clear_fifos

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

Clear a state machine's TX and RX FIFOFs.

Parameters

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

4.1.14.6.18. pio_sm_clkdiv_restart

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

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

Parameters

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

4.1.14.6.19. pio_sm_drain_tx_fifo

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

Empty out a state machine's TX FIFO.

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

Parameters

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

4.1.14.6.20. pio_sm_exec

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

Immediately execute an instruction on a state machine.

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

Parameters

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

See also

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

4.1.14.6.21. pio_sm_exec_wait_blocking

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

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

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

Parameters

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

See also

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

4.1.14.6.22. pio_sm_get

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

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

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

Parameters

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

4.1.14.6.23. pio_sm_get_blocking

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

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

Parameters

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

4.1.14.6.24. pio_sm_get_pc

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

Return the current program counter for a state machine.

Parameters

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

Returns

- the program counter

4.1.14.6.25. pio_sm_get_rx_fifo_level

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

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

Parameters

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

Returns

- the number of elements in the RX FIFO

4.1.14.6.26. pio_sm_get_tx_fifo_level

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

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

Parameters

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

Returns

- the number of elements in the TX FIFO

4.1.14.6.27. pio_sm_init

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

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

This method:

The state machine is disabled on return from this call

Parameters

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

4.1.14.6.28. pio_sm_is_exec_stalled

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

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

Parameters

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

Returns

- true if the executed instruction is still running (stalled)

4.1.14.6.29. pio_sm_is_rx_fifo_empty

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

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

Parameters

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

- `sm` State machine index (0..3)

Returns

- true if the RX FIFO is empty

4.1.14.6.30. pio_sm_is_rx_fifo_full

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

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

Parameters

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

Returns

- true if the RX FIFO is full

4.1.14.6.31. pio_sm_is_tx_fifo_empty

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

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

Parameters

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

Returns

- true if the TX FIFO is empty

4.1.14.6.32. pio_sm_is_tx_fifo_full

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

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

Parameters

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

Returns

- true if the TX FIFO is full

4.1.14.6.33. pio_sm_put

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

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

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

Parameters

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

4.1.14.6.34. pio_sm_put_blocking

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

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

Parameters

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

4.1.14.6.35. pio_sm_restart

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

Restart a state machine with a known state.

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

Parameters

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

4.1.14.6.36. pio_sm_set_clkdiv

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

set the current clock divider for a state machine

Parameters

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

4.1.14.6.37. pio_sm_set_clkdiv_int_frac

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

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

Parameters

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

4.1.14.6.38. pio_sm_set_config

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

Apply a state machine configuration to a state machine.

Parameters

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

4.1.14.6.39. pio_sm_set_consecutive_pindirs

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

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

This method repeatedly reconfigures the target state machine's pin configuration and executes 'set' instructions to set the pin direction on consecutive pins, before restoring the state machine's pin configuration to what it was.

This method is provided as a convenience to set initial pin directions, and should not be used against a state machine that is enabled.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3) to use
- `pin_base` the first pin to set a direction for
- `pin_count` the count of consecutive pins to set the direction for
- `is_out` the direction to set; true = out, false = in

4.1.14.6.40. pio_sm_set_enabled

```
static void pio_sm_set_enabled (PIO pio,
       uint sm,
       bool enabled)
```

Enable or disable a PIO state machine.

Parameters

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

- `enabled` true to enable the state machine; false to disable

4.1.14.6.41. pio_sm_set_pindirs_with_mask

```
void pio_sm_set_pindirs_with_mask (PIO pio,
       uint sm,
       uint32_t pin_dirs,
       uint32_t pin_mask)
```

Use a state machine to set the pin directions for multiple pins for the PIO instance.

This method repeatedly reconfigures the target state machine's pin configuration and executes 'set' instructions to set pin directions on up to 32 pins, before restoring the state machine's pin configuration to what it was.

This method is provided as a convenience to set initial pin directions, and should not be used against a state machine that is enabled.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3) to use
- `pin_dirs` the pin directions to set - 1 = out, 0 = in (if the corresponding bit in pin_mask is set)
- `pin_mask` a bit for each pin to indicate whether the corresponding pin_value for that pin should be applied.

4.1.14.6.42. pio_sm_set_pins

```
void pio_sm_set_pins (PIO pio,
       uint sm,
       uint32_t pin_values)
```

Use a state machine to set a value on all pins for the PIO instance.

This method repeatedly reconfigures the target state machine's pin configuration and executes 'set' instructions to set values on all 32 pins, before restoring the state machine's pin configuration to what it was.

This method is provided as a convenience to set initial pin states, and should not be used against a state machine that is enabled.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3) to use
- `pin_values` the pin values to set

4.1.14.6.43. pio_sm_set_pins_with_mask

```
void pio_sm_set_pins_with_mask (PIO pio,
       uint sm,
       uint32_t pin_values,
       uint32_t pin_mask)
```

Use a state machine to set a value on multiple pins for the PIO instance.

This method repeatedly reconfigures the target state machine's pin configuration and executes 'set' instructions to set values on up to 32 pins, before restoring the state machine's pin configuration to what it was.

This method is provided as a convenience to set initial pin states, and should not be used against a state machine that is enabled.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3) to use
- `pin_values` the pin values to set (if the corresponding bit in pin_mask is set)
- `pin_mask` a bit for each pin to indicate whether the corresponding pin_value for that pin should be applied.

4.1.14.6.44. pio_sm_set_wrap

```
static void pio_sm_set_wrap (PIO pio,
       uint sm,
       uint wrap_target,
       uint wrap)
```

Set the current wrap configuration for a state machine.

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)
- `wrap_target` the instruction memory address to wrap to
- `wrap` the instruction memory address after which to set the program counter to wrap_target if the instruction does not itself update the program_counter

4.1.14.6.45. pio_sm_unclaim

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

Mark a state machine as no longer used.

Method for cooperative claiming of hardware.

Parameters

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

4.1.15. sm_config

PIO state machine configuration.

A PIO block needs to be configured, these functions provide helpers to set up configuration structures. See pio_sm_set_config

4.1.15.1. Data Structures

- `struct pio_sm_config`
  PIO Configuration structure.

4.1.15.2. Function List

- `static void sm_config_set_out_pins (pio_sm_config *c, uint out_base, uint out_count)`
- `static void sm_config_set_set_pins (pio_sm_config *c, uint set_base, uint set_count)`
- `static void sm_config_set_in_pins (pio_sm_config *c, uint in_base)`

- static void sm_config_set_sideset_pins (pio_sm_config *c, uint sideset_base)
- static void sm_config_set_sideset (pio_sm_config *c, uint bit_count, bool optional, bool pindirs)
- static void sm_config_set_clkdiv (pio_sm_config *c, float div)
- static void sm_config_set_clkdiv_int_frac (pio_sm_config *c, uint16_t div_int, uint8_t div_frac)
- static void sm_config_set_wrap (pio_sm_config *c, uint wrap_target, uint wrap)
- static void sm_config_set_jmp_pin (pio_sm_config *c, uint pin)
- static void sm_config_set_in_shift (pio_sm_config *c, bool shift_right, bool autopush, uint push_threshold)
- static void sm_config_set_out_shift (pio_sm_config *c, bool shift_right, bool autopull, uint pull_threshold)
- static void sm_config_set_fifo_join (pio_sm_config *c, enum pio_fifo_join join)
- static void sm_config_set_out_special (pio_sm_config *c, bool sticky, bool has_enable_pin, int enable_pin_index)
- static void sm_config_set_mov_status (pio_sm_config *c, enum pio_mov_status_type status_sel, uint status_n)
- static pio_sm_config pio_get_default_sm_config ()
- static void pio_sm_set_out_pins (PIO pio, uint sm, uint out_base, uint out_count)
- static void pio_sm_set_set_pins (PIO pio, uint sm, uint set_base, uint set_count)
- static void pio_sm_set_in_pins (PIO pio, uint sm, uint in_base)
- static void pio_sm_set_sideset_pins (PIO pio, uint sm, uint sideset_base)

4.1.15.3. Function Documentation

4.1.15.3.1. pio_get_default_sm_config

static pio_sm_config pio_get_default_sm_config ()

Get the default state machine configuration.

Setting	Default
Out Pins	32 starting at 0
Set Pins	0 starting at 0
In Pins (base)	0
Side Set Pins (base)	0
Side Set	disabled
Wrap	wrap=31, wrap_to=0
In Shift	shift_direction=right, autopush=false, push_thrshold=32
Out Shift	shift_direction=right, autopull=false, pull_thrshold=32
Jmp Pin	0
Out Special	sticky=false, has_enable_pin=false, enable_pin_index=0
Mov Status	status_sel=STATUS_TX_LESSTHAN, n=0

Returns

- the default state machine configuration which can then be modified.

4.1.15.3.2. pio_sm_set_in_pins

```
static void pio_sm_set_in_pins (PIO pio,
       uint sm,
       uint in_base)
```

Set the current 'in' pins for a state machine.

Can overlap with the 'out', "set' and 'sideset' pins

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)
- `in_base` 0-31 First pin to set as input

4.1.15.3.3. pio_sm_set_out_pins

```
static void pio_sm_set_out_pins (PIO pio,
       uint sm,
       uint out_base,
       uint out_count)
```

Set the current 'out' pins for a state machine.

Can overlap with the 'in', 'set' and 'sideset' pins

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)
- `out_base` 0-31 First pin to set as output
- `out_count` 0-32 Number of pins to set.

4.1.15.3.4. pio_sm_set_set_pins

```
static void pio_sm_set_set_pins (PIO pio,
       uint sm,
       uint set_base,
       uint set_count)
```

Set the current 'set' pins for a state machine.

Can overlap with the 'in', 'out' and 'sideset' pins

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)
- `set_base` 0-31 First pin to set as
- `set_count` 0-5 Number of pins to set.

4.1.15.3.5. pio_sm_set_sideset_pins

```
static void pio_sm_set_sideset_pins (PIO pio,
       uint sm,
       uint sideset_base)
```

Set the current 'sideset' pins for a state machine.

Can overlap with the 'in', 'out' and 'set' pins

Parameters

- `pio` The PIO instance; either pio0 or pio1
- `sm` State machine index (0..3)
- `sideset_base` base pin for 'side set'

4.1.15.3.6. sm_config_set_clkdiv

```
static void sm_config_set_clkdiv (pio_sm_config *c,
       float div)
```

Set the state machine clock divider (from a floating point value) in a state machine configuration.

The clock divider acts on the system clock to provide a clock for the state machine. See the datasheet for more details.

Parameters

- `c` Pointer to the configuration structure to modify
- `div` The fractional divisor to be set. 1 for full speed. An integer clock divisor of n will cause the state machine to run 1 cycle in every n. Note that for small n, the jitter introduced by a fractional divider (e.g. 2.5) may be unacceptable although it will depend on the use case.

4.1.15.3.7. sm_config_set_clkdiv_int_frac

```
static void sm_config_set_clkdiv_int_frac (pio_sm_config *c,
       uint16_t div_int,
       uint8_t div_frac)
```

Set the state machine clock divider (from integer and fractional parts - 16:8) in a state machine configuration.

The clock divider acts on the system clock to provide a clock for the state machine. See the datasheet for more details.

Parameters

- `c` Pointer to the configuration structure to modify
- `div_int` Integer part of the divisor
- `div_frac` Fractional part in 1/256ths

See also

- sm_config_set_clkdiv

4.1.15.3.8. sm_config_set_fifo_join

```
static void sm_config_set_fifo_join (pio_sm_config *c,
       enum pio_fifo_join join)
```

Setup the FIFO joining in a state machine configuration.

Parameters

- `c` Pointer to the configuration structure to modify
- `join` Specifies the join type.

See also

- enum pio_fifo_join

4.1.15.3.9. sm_config_set_in_pins

```
static void sm_config_set_in_pins (pio_sm_config *c,
       uint in_base)
```

Set the 'in' pins in a state machine configuration.

Can overlap with the 'out', "set' and 'sideset' pins

Parameters

- `c` Pointer to the configuration structure to modify
- `in_base` 0-31 First pin to set as input

4.1.15.3.10. sm_config_set_in_shift

```
static void sm_config_set_in_shift (pio_sm_config *c,
       bool shift_right,
       bool autopush,
       uint push_threshold)
```

Setup 'in' shifting parameters in a state machine configuration.

Parameters

- `c` Pointer to the configuration structure to modify
- `shift_right` true to shift ISR to right, false to shift ISR to left
- `autopush` whether autopush is enabled
- `push_threshold` threshold in bits to shift in before auto/conditional re-pushing of the ISR

4.1.15.3.11. sm_config_set_jmp_pin

```
static void sm_config_set_jmp_pin (pio_sm_config *c,
       uint pin)
```

Set the 'jmp' pin in a state machine configuration.

Parameters

- `c` Pointer to the configuration structure to modify
- `pin` The raw GPIO pin number to use as the source for a jmp pin instruction

4.1.15.3.12. sm_config_set_mov_status

```
static void sm_config_set_mov_status (pio_sm_config *c,
       enum pio_mov_status_type status_sel,
       uint status_n)
```

Set source for 'mov status' in a state machine configuration.

Parameters

- `c` Pointer to the configuration structure to modify
- `status_sel` the status operation selector
- `status_n` parameter for the mov status operation (currently a bit count)

4.1.15.3.13. sm_config_set_out_pins

```
static void sm_config_set_out_pins (pio_sm_config *c,
       uint out_base,
       uint out_count)
```

Set the 'out' pins in a state machine configuration.

Can overlap with the 'in', 'set' and 'sideset' pins

Parameters

- `c` Pointer to the configuration structure to modify
- `out_base` 0-31 First pin to set as output
- `out_count` 0-32 Number of pins to set.

4.1.15.3.14. sm_config_set_out_shift

```
static void sm_config_set_out_shift (pio_sm_config *c,
       bool shift_right,
       bool autopull,
       uint pull_threshold)
```

Setup 'out' shifting parameters in a state machine configuration.

Parameters

- `c` Pointer to the configuration structure to modify
- `shift_right` true to shift OSR to right, false to shift OSR to left
- `autopull` whether autopull is enabled
- `pull_threshold` threshold in bits to shift out before auto/conditional re-pulling of the OSR

4.1.15.3.15. sm_config_set_out_special

```
static void sm_config_set_out_special (pio_sm_config *c,
       bool sticky,
       bool has_enable_pin,
       int enable_pin_index)
```

Set special 'out' operations in a state machine configuration.

Parameters

- `c` Pointer to the configuration structure to modify
- `sticky` to enable 'sticky' output (i.e. re-asserting most recent OUT/SET pin values on subsequent cycles)
- `has_enable_pin` true to enable auxiliary OUT enable pin
- `enable_pin_index` pin index for auxiliary OUT enable

4.1.15.3.16. sm_config_set_set_pins

```
static void sm_config_set_set_pins (pio_sm_config *c,
       uint set_base,
       uint set_count)
```

Set the 'set' pins in a state machine configuration.

Can overlap with the 'in', 'out' and 'sideset' pins

Parameters

- `c` Pointer to the configuration structure to modify
- `set_base` 0-31 First pin to set as
- `set_count` 0-5 Number of pins to set.

4.1.15.3.17. sm_config_set_sideset

```
static void sm_config_set_sideset (pio_sm_config *c,
       uint bit_count,
       bool optional,
       bool pindirs)
```

Set the 'sideset' options in a state machine configuration.

Parameters

- `c` Pointer to the configuration structure to modify
- `bit_count` Number of bits to steal from delay field in the instruction for use of side set
- `optional` True if the topmost side set bit is used as a flag for whether to apply side set on that instruction
- `pindirs` True if the side set affects pin directions rather than values

4.1.15.3.18. sm_config_set_sideset_pins

```
static void sm_config_set_sideset_pins (pio_sm_config *c,
       uint sideset_base)
```

Set the 'sideset' pins in a state machine configuration.

Can overlap with the 'in', 'out' and 'set' pins

Parameters

- `c` Pointer to the configuration structure to modify
- `sideset_base` base pin for 'side set'

4.1.15.3.19. sm_config_set_wrap

```
static void sm_config_set_wrap (pio_sm_config *c,
       uint wrap_target,
       uint wrap)
```

Set the wrap addresses in a state machine configuration.

Parameters

- `c` Pointer to the configuration structure to modify
- `wrap_target` the instruction memory address to wrap to
- `wrap` the instruction memory address after which to set the program counter to wrap_target if the instruction does not itself update the program_counter

4.1.16. hardware_pll

Phase Locked Loop control APIs.

There are two PLLs in RP2040. They are:

- pll_sys - Used to generate up to a 133MHz system clock

- pll_usb - Used to generate a 48MHz USB reference clock

For details on how the PLL's are calculated, please refer to the RP2040 datasheet.

4.1.16.1. Function List

- `void pll_init (PLL pll, uint32_t ref_div, uint32_t vco_freq, uint32_t post_div1, uint8_t post_div2)`
- `void pll_deinit (PLL pll)`

4.1.16.2. Function Documentation

4.1.16.2.1. 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

4.1.16.2.2. pll_init

```
void pll_init (PLL pll,
       uint32_t ref_div,
       uint32_t vco_freq,
       uint32_t post_div1,
       uint8_t post_div2)
```

Initialise specified PLL.

Parameters

- `pll` pll_sys or pll_usb
- `ref_div` Input clock divider.
- `vco_freq` Requested output from the VCO (voltage controlled oscillator)
- `post_div1` Post Divider 1 - range 1-7. Must be >= post_div2
- `post_div2` Post Divider 2 - range 1-7

4.1.17. hardware_pwm

Hardware Pulse Width Modulation (PWM) API.

The RP2040 PWM block has 8 identical slices. Each slice can drive two PWM output signals, or measure the frequency or duty cycle of an input signal. This gives a total of up to 16 controllable PWM outputs. All 30 GPIOs can be driven by the PWM block

The PWM hardware functions by continuously comparing the input value to a free-running counter. This produces a toggling output where the amount of time spent at the high output level is proportional to the input value. The fraction of time spent at the high signal level is known as the duty cycle of the signal.

The default behaviour of a PWM slice is to count upward until the wrap value (pwm_config_set_wrap) is reached, and then immediately wrap to 0. PWM slices also offer a phase-correct mode, where the counter starts to count downward after

reaching TOP, until it reaches 0 again.

Example

```
// Output PWM signals on pins 0 and 1

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

int main() {

    // Tell GPIO 0 and 1 they are allocated to the PWM
    gpio_set_function(0, GPIO_FUNC_PWM);
    gpio_set_function(1, GPIO_FUNC_PWM);

    // Find out which PWM slice is connected to GPIO 0 (it's slice 0)
    uint slice_num = pwm_gpio_to_slice_num(0);

    // Set period of 4 cycles (0 to 3 inclusive)
    pwm_set_wrap(slice_num, 3);
    // Set channel A output high for one cycle before dropping
    pwm_set_chan_level(slice_num, PWM_CHAN_A, 1);
    // Set initial B output high for three cycles before dropping
    pwm_set_chan_level(slice_num, PWM_CHAN_B, 3);
    // Set the PWM running
    pwm_set_enabled(slice_num, true);

    // Note we could also use pwm_set_gpio_level(gpio, x) which looks up the
    // correct slice and channel for a given GPIO.
}
```

4.1.17.1. Enumerations

- enum **pwm_clkdiv_mode** { PWM_DIV_FREE_RUNNING, PWM_DIV_B_HIGH, PWM_DIV_B_RISING, PWM_DIV_B_FALLING }
  PWM Divider mode settings.

4.1.17.2. Function List

- static uint pwm_gpio_to_slice_num (uint gpio)
- static uint pwm_gpio_to_channel (uint gpio)
- static void pwm_config_set_phase_correct (pwm_config *c, bool phase_correct)
- static void pwm_config_set_clkdiv (pwm_config *c, float div)
- static void pwm_config_set_clkdiv_int (pwm_config *c, uint div)
- static void pwm_config_set_clkdiv_mode (pwm_config *c, enum pwm_clkdiv_mode mode)
- static void pwm_config_set_output_polarity (pwm_config *c, bool a, bool b)
- static void pwm_config_set_wrap (pwm_config *c, uint16_t wrap)
- static void pwm_init (uint slice_num, pwm_config *c, bool start)
- static pwm_config pwm_get_default_config ()
- static void pwm_set_wrap (uint slice_num, uint16_t wrap)
- static void pwm_set_chan_level (uint slice_num, uint chan, uint16_t level)

- `static void pwm_set_both_levels (uint slice_num, uint16_t level_a, uint16_t level_b)`
- `static void pwm_set_gpio_level (uint gpio, uint16_t level)`
- `static int16_t pwm_get_counter (uint slice_num)`
- `static void pwm_set_counter (uint slice_num, uint16_t c)`
- `static void pwm_advance_count (uint slice_num)`
- `static void pwm_retard_count (uint slice_num)`
- `static void pwm_set_clkdiv_int_frac (uint slice_num, uint8_t integer, uint8_t fract)`
- `static void pwm_set_clkdiv (uint slice_num, float divider)`
- `static void pwm_set_output_polarity (uint slice_num, bool a, bool b)`
- `static void pwm_set_clkdiv_mode (uint slice_num, enum pwm_clkdiv_mode mode)`
- `static void pwm_set_phase_correct (uint slice_num, bool phase_correct)`
- `static void pwm_set_enabled (uint slice_num, bool enabled)`
- `static void pwm_set_mask_enabled (uint32_t mask)`
- `static void pwm_set_irq_enabled (uint slice_num, bool enabled)`
- `static void pwm_set_irq_mask_enabled (uint32_t slice_mask, bool enabled)`
- `static void pwm_clear_irq (uint slice_num)`
- `static int32_t pwm_get_irq_status_mask ()`
- `static void pwm_force_irq (uint slice_num)`

4.1.17.3. Function Documentation

4.1.17.3.1. pwm_advance_count

```
static void pwm_advance_count (uint slice_num)
```

Advance PWM count.

Advance the phase of a running the counter by 1 count.

This function will return once the increment is complete.

Parameters

- `slice_num` PWM slice number

4.1.17.3.2. pwm_clear_irq

```
static void pwm_clear_irq (uint slice_num)
```

Clear single PWM channel interrupt.

Parameters

- `slice_num` PWM slice number

4.1.17.3.3. pwm_config_set_clkdiv

```
static void pwm_config_set_clkdiv (pwm_config *c,
       float div)
```

Set clock divider in a PWM configuration.

If the divide mode is free-running, the PWM counter runs at clk_sys / div. Otherwise, the divider reduces the rate of events seen on the B pin input (level or edge) before passing them on to the PWM counter.

Parameters

- `c` PWM configuration struct to modify
- `div` Value to divide counting rate by. Must be greater than or equal to 1.

4.1.17.3.4. pwm_config_set_clkdiv_int

```
static void pwm_config_set_clkdiv_int (pwm_config *c,
       uint div)
```

Set PWM clock divider in a PWM configuration.

If the divide mode is free-running, the PWM counter runs at clk_sys / div. Otherwise, the divider reduces the rate of events seen on the B pin input (level or edge) before passing them on to the PWM counter.

Parameters

- `c` PWM configuration struct to modify
- `div` integer value to reduce counting rate by. Must be greater than or equal to 1.

4.1.17.3.5. pwm_config_set_clkdiv_mode

```
static void pwm_config_set_clkdiv_mode (pwm_config *c,
       enum pwm_clkdiv_mode mode)
```

Set PWM counting mode in a PWM configuration.

Configure which event gates the operation of the fractional divider. The default is always-on (free-running PWM). Can also be configured to count on high level, rising edge or falling edge of the B pin input.

Parameters

- `c` PWM configuration struct to modify
- `mode` PWM divide/count mode

4.1.17.3.6. pwm_config_set_output_polarity

```
static void pwm_config_set_output_polarity (pwm_config *c,
       bool a,
       bool b)
```

Set output polarity in a PWM configuration.

Parameters

- `c` PWM configuration struct to modify
- `a` true to invert output A
- `b` true to invert output B

4.1.17.3.7. pwm_config_set_phase_correct

```
static void pwm_config_set_phase_correct (pwm_config *c,
       bool phase_correct)
```

Set phase correction in a PWM configuration.

Setting phase control to true means that instead of wrapping back to zero when the wrap point is reached, the PWM starts counting back down. The output frequency is halved when phase-correct mode is enabled.

Parameters

- `c` PWM configuration struct to modify
- `phase_correct` true to set phase correct modulation, false to set trailing edge

4.1.17.3.8. pwm_config_set_wrap

```
static void pwm_config_set_wrap (pwm_config *c,
       uint16_t wrap)
```

Set PWM counter wrap value in a PWM configuration.

Set the highest value the counter will reach before returning to 0. Also known as TOP.

Parameters

- `c` PWM configuration struct to modify
- `wrap` Value to set wrap to

4.1.17.3.9. pwm_force_irq

```
static void pwm_force_irq (uint slice_num)
```

Force PWM interrupt.

Parameters

- `slice_num` PWM slice number

4.1.17.3.10. pwm_get_counter

```
static int16_t pwm_get_counter (uint slice_num)
```

Get PWM counter.

Get current value of PWM counter

Parameters

- `slice_num` PWM slice number

Returns

- Current value of PWM counter

4.1.17.3.11. pwm_get_default_config

```
static pwm_config pwm_get_default_config ()
```

Get a set of default values for PWM configuration.

PWM config is free running at system clock speed, no phase correction, wrapping at 0xffff, with standard polarities for channels A and B.

Returns

- Set of default values.

4.1.17.3.12. pwm_get_irq_status_mask

```
static int32_t pwm_get_irq_status_mask ()
```

Get PWM interrupt status, raw.

Returns

- Bitmask of all PWM interrupts currently set

4.1.17.3.13. pwm_gpio_to_channel

```
static uint pwm_gpio_to_channel (uint gpio)
```

Determine the PWM channel that is attached to the specified GPIO.

Each slice 0 to 7 has two channels, A and B.

Returns

- The PWM channel that controls the specified GPIO.

4.1.17.3.14. pwm_gpio_to_slice_num

```
static uint pwm_gpio_to_slice_num (uint gpio)
```

Determine the PWM slice that is attached to the specified GPIO.

Returns

- The PWM slice number that controls the specified GPIO.

4.1.17.3.15. pwm_init

```
static void pwm_init (uint slice_num,
      pwm_config *c,
      bool start)
```

Initialise a PWM with settings from a configuration object.

Use the `pwm_get_default_config()` function to initialise a config structure, make changes as needed using the pwm_config_* functions, then call this function to set up the PWM.

Parameters

- `slice_num` PWM slice number
- `c` The configuration to use
- `start` If true the PWM will be started running once configured. If false you will need to start manually using pwm_set_enabled() or pwm_set_mask_enabled()

4.1.17.3.16. pwm_retard_count

```
static void pwm_retard_count (uint slice_num)
```

Retard PWM count.

Retard the phase of a running counter by 1 count

This function will return once the retardation is complete.

Parameters

- `slice_num` PWM slice number

4.1.17.3.17. pwm_set_both_levels

```
static void pwm_set_both_levels (uint slice_num,
      uint16_t level_a,
      uint16_t level_b)
```

Set PWM counter compare values.

Set the value of the PWM counter compare values, A and B

Parameters

- `slice_num` PWM slice number
- `level_a` Value to set compare A to. When the counter reaches this value the A output is deasserted
- `level_b` Value to set compare B to. When the counter reaches this value the B output is deasserted

4.1.17.3.18. pwm_set_chan_level

```
static void pwm_set_chan_level (uint slice_num,
       uint chan,
       uint16_t level)
```

Set the current PWM counter compare value for one channel.

Set the value of the PWM counter compare value, for either channel A or channel B

Parameters

- `slice_num` PWM slice number
- `chan` Which channel to update. 0 for A, 1 for B.
- `level` new level for the selected output

4.1.17.3.19. pwm_set_clkdiv

```
static void pwm_set_clkdiv (uint slice_num,
       float divider)
```

Set PWM clock divider.

Set the clock divider. Counter increment will be on sysclock divided by this value, taking in to account the gating.

Parameters

- `slice_num` PWM slice number
- `divider` Floating point clock divider, 1.f ⇐ value < 256.f

4.1.17.3.20. pwm_set_clkdiv_int_frac

```
static void pwm_set_clkdiv_int_frac (uint slice_num,
       uint8_t integer,
       uint8_t fract)
```

Set PWM clock divider using an 8:4 fractional value.

Set the clock divider. Counter increment will be on sysclock divided by this value, taking in to account the gating.

Parameters

- `slice_num` PWM slice number
- `integer` 8 bit integer part of the clock divider
- `fract` 4 bit fractional part of the clock divider

4.1.17.3.21. pwm_set_clkdiv_mode

```
static void pwm_set_clkdiv_mode (uint slice_num,
       enum pwm_clkdiv_mode mode)
```

Set PWM divider mode.

Parameters

- `slice_num` PWM slice number
- `mode` Required divider mode

4.1.17.3.22. pwm_set_counter

```
static void pwm_set_counter (uint slice_num,
       uint16_t c)
```

Set PWM counter.

Set the value of the PWM counter

Parameters

- `slice_num` PWM slice number
- `c` Value to set the PWM counter to

4.1.17.3.23. pwm_set_enabled

```
static void pwm_set_enabled (uint slice_num,
       bool enabled)
```

Enable/Disable PWM.

Parameters

- `slice_num` PWM slice number
- `enabled` true to enable the specified PWM, false to disable

4.1.17.3.24. pwm_set_gpio_level

```
static void pwm_set_gpio_level (uint gpio,
       uint16_t level)
```

Helper function to set the PWM level for the slice and channel associated with a GPIO.

Look up the correct slice (0 to 7) and channel (A or B) for a given GPIO, and update the corresponding counter-compare field.

This PWM slice should already have been configured and set running. Also be careful of multiple GPIOs mapping to the same slice and channel (if GPIOs have a difference of 16).

Parameters

- `gpio` GPIO to set level of
- `level` PWM level for this GPIO

4.1.17.3.25. pwm_set_irq_enabled

```
static void pwm_set_irq_enabled (uint slice_num,
       bool enabled)
```

Enable PWM instance interrupt.

Used to enable a single PWM instance interrupt

Parameters

- `slice_num` PWM block to enable/disable
- `enabled` true to enable, false to disable

4.1.17.3.26. pwm_set_irq_mask_enabled

```
static void pwm_set_irq_mask_enabled (uint32_t slice_mask,
       bool enabled)
```

Enable multiple PWM instance interrupts.

Use this to enable multiple PWM interrupts at once.

Parameters

- `slice_mask` Bitmask of all the blocks to enable/disable. Channel 0 = bit 0, channel 1 = bit 1 etc.
- `enabled` true to enable, false to disable

4.1.17.3.27. pwm_set_mask_enabled

```
static void pwm_set_mask_enabled (uint32_t mask)
```

Enable/Disable multiple PWM slices simultaneously.

Parameters

- `mask` Bitmap of PWMs to enable/disable. Bits 0 to 7 enable slices 0-7 respectively

4.1.17.3.28. pwm_set_output_polarity

```
static void pwm_set_output_polarity (uint slice_num,
       bool a,
       bool b)
```

Set PWM output polarity.

Parameters

- `slice_num` PWM slice number
- `a` true to invert output A
- `b` true to invert output B

4.1.17.3.29. pwm_set_phase_correct

```
static void pwm_set_phase_correct (uint slice_num,
       bool phase_correct)
```

Set PWM phase correct on/off.

Setting phase control to true means that instead of wrapping back to zero when the wrap point is reached, the PWM starts counting back down. The output frequency is halved when phase-correct mode is enabled.

Parameters

- `slice_num` PWM slice number
- `phase_correct` true to set phase correct modulation, false to set trailing edge

4.1.17.3.30. pwm_set_wrap

```
static void pwm_set_wrap (uint slice_num,
       uint16_t wrap)
```

Set the current PWM counter wrap value.

Set the highest value the counter will reach before returning to 0. Also known as TOP.

Parameters

- `slice_num` PWM slice number
- `wrap` Value to set wrap to

4.1.18. hardware_resets

Hardware Reset API.

The reset controller allows software control of the resets to all of the peripherals that are not critical to boot the processor in the RP2040.

reset_bitmask

Multiple blocks are referred to using a bitmask as follows:

Block to reset	Bit
USB	24
UART 1	23
UART 0	22
Timer	21
TB Manager	20
SysInfo	19
System Config	18
SPI 1	17
SPI 0	16
RTC	15
PWM	14
PLL USB	13
PLL System	12
PIO 1	11
PIO 0	10
Pads - QSPI	9
Pads - bank 0	8
JTAG	7
IO Bank 1	6
IO Bank 0	5
I2C 1	4
I2C 0	3
DMA	2
Bus Control	1

Block to reset	Bit
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(RESETS_RESET_PWM_BITS);

    // And bring it out
    unreset_block_wait(RESETS_RESET_PWM_BITS);

    // Put the PWM and RTC block into reset
    reset_block(RESETS_RESET_PWM_BITS | RESETS_RESET_RTC_BITS);

    // Wait for both to come out of reset
    unreset_block_wait(RESETS_RESET_PWM_BITS | RESETS_RESET_RTC_BITS);

    return 0;
}
```

4.1.18.1. Function List

- `static void reset_block (uint32_t bits)`
- `static void unreset_block (uint32_t bits)`
- `static void unreset_block_wait (uint32_t bits)`

4.1.18.2. Function Documentation

4.1.18.2.1. reset_block

`static void reset_block (uint32_t bits)`

Reset the specified HW blocks.

Parameters

- `bits` Bit pattern indicating blocks to reset. See reset_bitmask

4.1.18.2.2. unreset_block

`static void unreset_block (uint32_t bits)`

bring specified HW blocks out of reset

Parameters

- `bits` Bit pattern indicating blocks to unreset. See reset_bitmask

4.1.18.2.3. unreset_block_wait

```
static void unreset_block_wait (uint32_t bits)
```

Bring specified HW blocks out of reset and wait for completion.

Parameters

- `bits` Bit pattern indicating blocks to unreset. See reset_bitmask

4.1.19. hardware_rtc

Hardware Real Time Clock API.

The RTC keeps track of time in human readable format and generates events when the time is equal to a preset value. Think of a digital clock, not epoch time used by most computers. There are seven fields, one each for year (12 bit), month (4 bit), day (5 bit), day of the week (3 bit), hour (5 bit) minute (6 bit) and second (6 bit), storing the data in binary format.

See also

- datetime_t

Example

```c
#include <stdio.h>
#include "hardware/rtc.h"
#include "pico/stdlib.h"
#include "pico/util/datetime.h"

int main() {
    stdio_init_all();
    printf("Hello RTC!\n");

    char datetime_buf[256];
    char *datetime_str = &datetime_buf[0];

    // Start on Friday 5th of June 2020 15:45:00
    datetime_t t = {
            .year  = 2020,
            .month = 06,
            .day   = 05,
            .dotw  = 5, // 0 is Sunday, so 5 is Friday
            .hour  = 15,
            .min   = 45,
            .sec   = 00
    };

    // Start the RTC
    rtc_init();
    rtc_set_datetime(&t);

    // Print the time
    while (true) {
        rtc_get_datetime(&t);
        datetime_to_str(datetime_str, sizeof(datetime_buf), &t);
        printf("\r%s      ", datetime_str);
        sleep_ms(100);
    }

    return 0;
```

```
37 }
```

4.1.19.1. Function List

- `void rtc_init (void)`
- `bool rtc_set_datetime (datetime_t *t)`
- `bool rtc_get_datetime (datetime_t *t)`
- `bool rtc_running (void)`
- `void rtc_set_alarm (datetime_t *t, rtc_callback_t user_callback)`
- `void rtc_disable_alarm (void)`

4.1.19.2. Function Documentation

4.1.19.2.1. rtc_disable_alarm

```
void rtc_disable_alarm (void)
```

Disable the RTC alarm (if active)

4.1.19.2.2. 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.

4.1.19.2.3. rtc_init

```
void rtc_init (void)
```

Initialise the RTC system.

4.1.19.2.4. rtc_running

```
bool rtc_running (void)
```

Is the RTC running?

4.1.19.2.5. rtc_set_alarm

```
void rtc_set_alarm (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

4.1.19.2.6. rtc_set_datetime

```
bool rtc_set_datetime (datetime_t *t)
```

Set the RTC to the specified time.

Parameters

- `t` Pointer to a datetime_t structure contains time to set

Returns

- true if set, false if the passed in datetime was invalid.

4.1.20. hardware_spi

Hardware SPI API.

RP2040 has 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.

4.1.20.1. Macros

- `#define spi0 ((spi_inst_t * const)spi0_hw)`
- `#define spi1 ((spi_inst_t * const)spi1_hw)`

4.1.20.2. Function List

- `void spi_init (spi_inst_t *spi, uint baudrate)`
- `void spi_deinit (spi_inst_t *spi)`
- `uint spi_set_baudrate (spi_inst_t *spi, uint baudrate)`
- `static uint spi_get_index (spi_inst_t *spi)`
- `static void spi_set_format (spi_inst_t *spi, uint data_bits, spi_cpol_t cpol, spi_cpha_t cpha, spi_order_t order)`
- `static void spi_set_slave (spi_inst_t *spi, bool slave)`
- `static size_t spi_is_writable (spi_inst_t *spi)`
- `static size_t spi_is_readable (spi_inst_t *spi)`
- `int spi_write_read_blocking (spi_inst_t *spi, const uint8_t *src, uint8_t *dst, size_t len)`
- `int spi_write_blocking (spi_inst_t *spi, const uint8_t *src, size_t len)`
- `int spi_read_blocking (spi_inst_t *spi, uint8_t repeated_tx_data, uint8_t *dst, size_t len)`
- `int spi_write16_read16_blocking (spi_inst_t *spi, const uint16_t *src, uint16_t *dst, size_t len)`

- `int spi_write16_blocking (spi_inst_t *spi, const uint16_t *src, size_t len)`
- `int spi_read16_blocking (spi_inst_t *spi, uint16_t repeated_tx_data, uint16_t *dst, size_t len)`

4.1.20.3. Function Documentation

4.1.20.3.1. spi_deinit

```
void spi_deinit (spi_inst_t *spi)
```

Deinitialise SPI instances.

Puts the SPI into a disabled state. Init will need to be called to reenable the device functions.

Parameters

- `spi` SPI instance specifier, either spi0 or spi1

4.1.20.3.2. spi_get_index

```
static uint spi_get_index (spi_inst_t *spi)
```

Convert I2c instance to hardware instance number.

Parameters

- `spi` SPI instance

Returns

- Number of SPI, 0 or 1.

4.1.20.3.3. spi_init

```
void spi_init (spi_inst_t *spi,
       uint baudrate)
```

Initialise SPI instances.

Puts the SPI into a known state, and enable it. Must be called before other functions.

Parameters

- `spi` SPI instance specifier, either spi0 or spi1
- `baudrate` Baudrate required in Hz

4.1.20.3.4. spi_is_readable

```
static size_t spi_is_readable (spi_inst_t *spi)
```

Check whether a read can be done on SPI device.

Parameters

- `spi` SPI instance specifier, either spi0 or spi1

Returns

- Non-zero if a read is possible i.e. data is present

4.1.20.3.5. spi_is_writable

```
static size_t spi_is_writable (spi_inst_t *spi)
```

Check whether a write can be done on SPI device.

Parameters

- `spi` SPI instance specifier, either spi0 or spi1

Returns

- 0 if no space is available to write. Non-zero if a write is possible

4.1.20.3.6. spi_read16_blocking

```
int spi_read16_blocking (spi_inst_t *spi,
       uint16_t repeated_tx_data,
       uint16_t *dst,
       size_t len)
```

Read from an SPI device.

Read `len` halfwords from SPI to `dst`. Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate. `repeated_tx_data` is output repeatedly on 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 in halfwords

Returns

- Number of bytes written/read

4.1.20.3.7. spi_read_blocking

```
int spi_read_blocking (spi_inst_t *spi,
       uint8_t repeated_tx_data,
       uint8_t *dst,
       size_t len)
```

Read from an SPI device.

Read `len` bytes from SPI to `dst`. Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate. `repeated_tx_data` is output repeatedly on 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

4.1.20.3.8. spi_set_baudrate

```
uint spi_set_baudrate (spi_inst_t *spi,
      uint baudrate)
```

Set SPI baudrate.

Set SPI frequency as close as possible to baudrate, and return the actual achieved rate.

Parameters

- `spi` SPI instance specifier, either spi0 or spi1
- `baudrate` Baudrate required in Hz, should be capable of a bitrate of at least 2Mbps, or higher, depending on system clock settings.

Returns

- The actual baudrate set

4.1.20.3.9. spi_set_format

```
static void spi_set_format (spi_inst_t *spi,
      uint data_bits,
      spi_cpol_t cpol,
      spi_cpha_t cpha,
      spi_order_t order)
```

Configure SPI.

Configure how the SPI serialises and deserialises data on the wire

Parameters

- `spi` SPI instance specifier, either spi0 or spi1
- `data_bits` Number of data bits per transfer. Valid values 4..16.
- `cpol` SSPCLKOUT polarity, applicable to Motorola SPI frame format only.
- `cpha` SSPCLKOUT phase, applicable to Motorola SPI frame format only
- `order` Must be SPI_MSB_FIRST, no other values supported on the PL022

4.1.20.3.10. spi_set_slave

```
static void spi_set_slave (spi_inst_t *spi,
      bool slave)
```

Set SPI master/slave.

Configure the SPI for master- or slave-mode operation. By default, `spi_init()` sets master-mode.

Parameters

- `spi` SPI instance specifier, either spi0 or spi1
- `slave` true to set SPI device as a slave device, false for master.

4.1.20.3.11. spi_write16_blocking

```
int spi_write16_blocking (spi_inst_t *spi,
      const uint16_t *src,
      size_t len)
```

Write to an SPI device.

Write `len` halfwords from `src` to SPI. Discard any data received back. Blocks until all data is transferred. No timeout, as SPI

hardware always transfers at a known data rate.

Parameters

- `spi` SPI instance specifier, either spi0 or spi1
- `src` Buffer of data to write
- `len` Length of buffers

Returns

- Number of bytes written/read

4.1.20.3.12. spi_write16_read16_blocking

```
int spi_write16_read16_blocking (spi_inst_t *spi,
       const uint16_t *src,
       uint16_t *dst,
       size_t len)
```

Write/Read half words to/from an SPI device.

Write `len` halfwords from `src` to SPI. Simultaneously read `len` halfwords from SPI to `dst`. Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate.

Parameters

- `spi` SPI instance specifier, either spi0 or spi1
- `src` Buffer of data to write
- `dst` Buffer for read data
- `len` Length of BOTH buffers in halfwords

Returns

- Number of bytes written/read

4.1.20.3.13. spi_write_blocking

```
int spi_write_blocking (spi_inst_t *spi,
       const uint8_t *src,
       size_t len)
```

Write to an SPI device, blocking.

Write `len` bytes from `src` to SPI, and discard any data received back Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate.

Parameters

- `spi` SPI instance specifier, either spi0 or spi1
- `src` Buffer of data to write
- `len` Length of src

Returns

- Number of bytes written/read

4.1.20.3.14. spi_write_read_blocking

```
int spi_write_read_blocking (spi_inst_t *spi,
       const uint8_t *src,
       uint8_t *dst,
```

size_t len)

Write/Read to/from an SPI device.

Write `len` bytes from `src` to SPI. Simultaneously read `len` bytes from SPI to `dst`. Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate.

Parameters

- `spi` SPI instance specifier, either spi0 or spi1
- `src` Buffer of data to write
- `dst` Buffer for read data
- `len` Length of BOTH buffers

Returns

- Number of bytes written/read

4.1.21. hardware_sync

Low level hardware spin-lock, barrier and processor event API.

Functions for synchronisation between core's, HW, etc

The RP2040 provides 32 hardware spin locks, which can be used to manage mutually-exclusive access to shared software resources.

spin locks 0-15 are currently reserved for fixed uses by the SDK - i.e. if you use them other functionality may break or not function optimally

4.1.21.1. Typedefs

- `typedef uint32_t spin_lock_t`
  A spin lock identifier.

4.1.21.2. Function List

- `static void __sev ()`
- `static void __wfe ()`
- `static void __wfi ()`
- `static void __dmb ()`
- `static void __isb ()`
- `static void __mem_fence_acquire ()`
- `static void __mem_fence_release ()`
- `static uint32_t save_and_disable_interrupts ()`
- `static void restore_interrupts (uint32_t status)`
- `static spin_lock_t * spin_lock_instance (uint lock_num)`
- `static uint spin_lock_get_num (spin_lock_t *lock)`
- `static void spin_lock_unsafe_blocking (spin_lock_t *lock)`
- `static void spin_unlock_unsafe (spin_lock_t *lock)`
- `static uint32_t spin_lock_blocking (spin_lock_t *lock)`

- static bool is_spin_locked (const spin_lock_t *lock)
- static void spin_unlock (spin_lock_t *lock, uint32_t saved_irq)
- static uint get_core_num ()
- spin_lock_t * spin_lock_init (uint lock_num)
- void spin_locks_reset (void)
- void spin_lock_claim (uint lock_num)
- void spin_lock_claim_mask (uint32_t lock_num_mask)
- void spin_lock_unclaim (uint lock_num)
- int spin_lock_claim_unused (bool required)

4.1.21.3. Function Documentation

4.1.21.3.1. __dmb

```
static void __dmb ()
```

Insert a DMB instruction in to the code path.

The DMB (data memory barrier) acts as a memory barrier, all memory accesses prior to this instruction will be observed before any explicit access after the instruction.

4.1.21.3.2. __isb

```
static void __isb ()
```

Insert a ISB instruction in to the code path.

ISB acts as an instruction synchronization barrier. It flushes the pipeline of the processor, so that all instructions following the ISB are fetched from cache or memory again, after the ISB instruction has been completed.

4.1.21.3.3. __mem_fence_acquire

```
static void __mem_fence_acquire ()
```

Acquire a memory fence.

4.1.21.3.4. __mem_fence_release

```
static void __mem_fence_release ()
```

Release a memory fence.

4.1.21.3.5. __sev

```
static void __sev ()
```

Insert a SEV instruction in to the code path.

The SEV (send event) instruction sends an event to both cores.

4.1.21.3.6. __wfe

```
static void __wfe ()
```

Insert a WFE instruction in to the code path.

The WFE (wait for event) instruction waits until one of a number of events occurs, including events signalled by the SEV instruction on either core.

4.1.21.3.7. __wfi

```
static void __wfi ()
```

Insert a WFI instruction in to the code path.

The WFI (wait for interrupt) instruction waits for a interrupt to wake up the core.

4.1.21.3.8. get_core_num

```
static uint get_core_num ()
```

Get the current core number.

Returns

- The core number the call was made from

4.1.21.3.9. is_spin_locked

```
static bool is_spin_locked (const spin_lock_t *lock)
```

Check to see if a spinlock is currently acquired elsewhere.

Parameters

- `lock` Spinlock instance

4.1.21.3.10. restore_interrupts

```
static void restore_interrupts (uint32_t status)
```

Restore interrupts to a specified state.

Parameters

- `status` Previous interrupt status from save_and_disable_interrupts()

4.1.21.3.11. save_and_disable_interrupts

```
static uint32_t save_and_disable_interrupts ()
```

Save and disable interrupts.

Returns

- The prior interrupt enable status for restoration later via `restore_interrupts()`

4.1.21.3.12. spin_lock_blocking

```
static uint32_t spin_lock_blocking (spin_lock_t *lock)
```

Acquire a spin lock safely.

This function will disable interrupts prior to acquiring the spinlock

Parameters

- `lock` Spinlock instance

Returns

- interrupt status to be used when unlocking, to restore to original state

4.1.21.3.13. spin_lock_claim

```
void spin_lock_claim (uint lock_num)
```

Mark a spin lock as used.

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

Parameters

- `lock_num` the spin lock number

4.1.21.3.14. spin_lock_claim_mask

```
void spin_lock_claim_mask (uint32_t lock_num_mask)
```

Mark multiple spin locks as used.

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

Parameters

- `lock_num_mask` Bitfield of all required spin locks to claim (bit 0 == spin lock 0, bit 1 == spin lock 1 etc)

4.1.21.3.15. spin_lock_claim_unused

```
int spin_lock_claim_unused (bool required)
```

Claim a free spin lock.

Parameters

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

Returns

- the spin lock number or -1 if required was false, and none were free

4.1.21.3.16. spin_lock_get_num

```
static uint spin_lock_get_num (spin_lock_t *lock)
```

Get HW Spinlock number from instance.

Parameters

- `lock` The Spinlock instance

Returns

- The Spinlock ID

4.1.21.3.17. spin_lock_init

```
spin_lock_t* spin_lock_init (uint lock_num)
```

Initialise a spin lock.

The spin lock is initially unlocked

Parameters

- `lock_num` The spin lock number

Returns

- The spin lock instance

4.1.21.3.18. spin_lock_instance

```
static spin_lock_t* spin_lock_instance (uint lock_num)
```

Get HW Spinlock instance from number.

Parameters

- `lock_num` Spinlock ID

Returns

- The spinlock instance

4.1.21.3.19. spin_lock_unclaim

```
void spin_lock_unclaim (uint lock_num)
```

Mark a spin lock as no longer used.

Method for cooperative claiming of hardware.

Parameters

- `lock_num` the spin lock number to release

4.1.21.3.20. spin_lock_unsafe_blocking

```
static void spin_lock_unsafe_blocking (spin_lock_t *lock)
```

Acquire a spin lock without disabling interrupts (hence unsafe)

Parameters

- `lock` Spinlock instance

4.1.21.3.21. spin_locks_reset

```
void spin_locks_reset (void)
```

Release all spin locks.

4.1.21.3.22. spin_unlock

```
static void spin_unlock (spin_lock_t *lock,
       uint32_t saved_irq)
```

Release a spin lock safely.

This function will re-enable interrupts according to the parameters.

Parameters

- `lock` Spinlock instance
- `saved_irq` Return value from the spin_lock_blocking() function.

Returns

- interrupt status to be used when unlocking, to restore to original state

See also

- spin_lock_blocking()

4.1.21.3.23. spin_unlock_unsafe

`static void spin_unlock_unsafe (spin_lock_t *lock)`

Release a spin lock without re-enabling interrupts.

Parameters

- `lock` Spinlock instance

4.1.22. hardware_timer

Low-level hardware timer API.

This API provides medium level access to the timer HW. See also pico_time which provides higher levels functionality using the hardware timer.

The timer peripheral on RP2040 supports the following features:

- single 64-bit counter, incrementing once per microsecond
- Latching two-stage read of counter, for race-free read over 32 bit bus
- Four alarms: match on the lower 32 bits of counter, IRQ on match.

By default the timer uses a one microsecond reference that is generated in the Watchdog (see Section 4.8.2) which is derived from the clk_ref.

The timer has 4 alarms, and can output a separate interrupt for each alarm. The alarms match on the lower 32 bits of the 64 bit counter which means they can be fired a maximum of 2^32 microseconds into the future. This is equivalent to:

- 2^32 ÷ 10^6: ~4295 seconds
- 4295 ÷ 60: ~72 minutes

The timer is expected to be used for short sleeps, if you want a longer alarm see the hardware_rtc functions.

Example

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

volatile bool timer_fired = false;

int64_t alarm_callback(alarm_id_t id, void *user_data) {
    printf("Timer %d fired!\n", (int) id);
    timer_fired = true;
    // Can return a value here in us to fire in the future
    return 0;
}

bool repeating_timer_callback(struct repeating_timer *t) {
    printf("Repeat at %lld\n", time_us_64());
    return true;
}

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

```
21
22     // Call alarm_callback in 2 seconds
23     add_alarm_in_ms(2000, alarm_callback, NULL, false);
24
25     // Wait for alarm callback to set timer_fired
26     while (!timer_fired) {
27         tight_loop_contents();
28     }
29
30     // Create a repeating timer that calls repeating_timer_callback.
31     // If the delay is > 0 then this is the delay between the previous callback ending and the
   next starting.
32     // If the delay is negative (see below) then the next call to the callback will be exactly
   500ms after the
33     // start of the call to the last callback
34     struct repeating_timer timer;
35     add_repeating_timer_ms(500, repeating_timer_callback, NULL, &timer);
36     sleep_ms(3000);
37     bool cancelled = cancel_repeating_timer(&timer);
38     printf("cancelled... %d\n", cancelled);
39     sleep_ms(2000);
40
41     // Negative delay so means we will call repeating_timer_callback, and call it again
42     // 500ms later regardless of how long the callback took to execute
43     add_repeating_timer_ms(-500, repeating_timer_callback, NULL, &timer);
44     sleep_ms(3000);
45     cancelled = cancel_repeating_timer(&timer);
46     printf("cancelled... %d\n", cancelled);
47     sleep_ms(2000);
48     printf("Done\n");
49     return 0;
50 }
```

See also

- pico_time

4.1.22.1. Typedefs

- `typedef void(* hardware_alarm_callback_t )(uint alarm_num)`

4.1.22.2. Function List

- `static uint32_t time_us_32 ()`
- `uint64_t time_us_64 ()`
- `void busy_wait_us_32 (uint32_t delay_us)`
- `void busy_wait_us (uint64_t delay_us)`
- `void busy_wait_until (absolute_time_t t)`
- `static bool time_reached (absolute_time_t t)`
- `void hardware_alarm_claim (uint alarm_num)`
- `void hardware_alarm_unclaim (uint alarm_num)`
- `void hardware_alarm_set_callback (uint alarm_num, hardware_alarm_callback_t callback)`

4.1.22.3. Function Documentation

4.1.22.3.1. busy_wait_until

```
void busy_wait_until (absolute_time_t t)
```

Busy wait wasting cycles until after the specified timestamp.

Parameters

- `t` Absolute time to wait until

4.1.22.3.2. busy_wait_us

```
void busy_wait_us (uint64_t delay_us)
```

Busy wait wasting cycles for the given (64 bit) number of microseconds.

Parameters

- `delay_us` delay amount

4.1.22.3.3. busy_wait_us_32

```
void busy_wait_us_32 (uint32_t delay_us)
```

Busy wait wasting cycles for the given (32 bit) number of microseconds.

Busy wait wasting cycles for the given (32 bit) number of microseconds.

Parameters

- `delay_us` delay amount

4.1.22.3.4. hardware_alarm_claim

```
void hardware_alarm_claim (uint alarm_num)
```

cooperatively claim the use of this hardware alarm_num

This method hard asserts if the hardware alarm is currently claimed.

Parameters

- `alarm_num` the hardware alarm to claim

See also

- hardware_claiming

4.1.22.3.5. hardware_alarm_set_callback

```
void hardware_alarm_set_callback (uint alarm_num,
      hardware_alarm_callback_t callback)
```

Enable/Disable a callback for a hardware timer on this core.

This method enables/disables the alarm IRQ for the specified hardware alarm on the calling core, and set the specified callback to be associated with that alarm.

This callback will be used for the timeout set via hardware_alarm_set_target

Parameters

- `alarm_num` the hardware alarm number

- `callback` the callback to install, or NULL to unset

See also

- hardware_alarm_set_target

4.1.22.3.6. hardware_alarm_unclaim

```
void hardware_alarm_unclaim (uint alarm_num)
```

cooperatively release the claim on use of this hardware alarm_num

Parameters

- `alarm_num` the hardware alarm to unclaim

See also

- hardware_claiming

4.1.22.3.7. time_reached

```
static bool time_reached (absolute_time_t t)
```

Check if the specified timestamp has been reached.

Parameters

- `t` Absolute time to compare against current time

Returns

- true if it is now after the specified timestamp

4.1.22.3.8. time_us_32

```
static uint32_t time_us_32 ()
```

Return a 32 bit timestamp value in microseconds.

Returns the low 32 bits of the hardware timer.

Returns

- the 32 bit timestamp

4.1.22.3.9. time_us_64

```
uint64_t time_us_64 ()
```

Return the current 64 bit timestamp value in microseconds.

Returns the full 64 bits of the hardware timer. The `pico_time` and other functions rely on the fact that this value monotonically increases from power up. As such it is expected that this value counts upwards and never wraps (we apologize for introducing a potential year 5851444 bug).

Return the current 64 bit timestamp value in microseconds.

Returns

- the 64 bit timestamp

4.1.23. hardware_uart

Hardware UART API.

RP2040 has 2 identical instances of a UART peripheral, based on the ARM PL011. Each UART can be connected to a number of GPIO pins as defined in the GPIO muxing.

Only the TX, RX, RTS, and CTS signals are connected, meaning that the modem mode and IrDA mode of the PL011 are not supported.

Example

```
1  int main() {
2
3      // Initialise UART 0
4      uart_init(uart0, 115200);
5
6      // Set the GPIO pin mux to the UART - 0 is TX, 1 is RX
7      gpio_set_function(0, GPIO_FUNC_UART);
8      gpio_set_function(1, GPIO_FUNC_UART);
9
10     uart_puts(uart0, "Hello world!");
11 }
```

4.1.23.1. Enumerations

- `enum uart_parity_t { UART_PARITY_NONE, UART_PARITY_EVEN, UART_PARITY_ODD }`
  UART Parity enumeration.

4.1.23.2. Macros

- `#define uart0 ((uart_inst_t * const)uart0_hw)`
  Identifier for UART instance 0.
- `#define uart1 ((uart_inst_t * const)uart1_hw)`
  Identifier for UART instance 1.

4.1.23.3. Function List

- `static uint uart_get_index (uart_inst_t *uart)`
- `uint uart_init (uart_inst_t *uart, uint baudrate)`
- `void uart_deinit (uart_inst_t *uart)`
- `uint uart_set_baudrate (uart_inst_t *uart, uint baudrate)`
- `static void uart_set_hw_flow (uart_inst_t *uart, bool cts, bool rts)`
- `static void uart_set_format (uart_inst_t *uart, uint data_bits, uint stop_bits, uart_parity_t parity)`
- `static void uart_set_irq_enables (uart_inst_t *uart, bool rx_has_data, bool tx_needs_data)`
- `static bool uart_is_enabled (uart_inst_t *uart)`
- `static void uart_set_fifo_enabled (uart_inst_t *uart, bool enabled)`
- `static bool uart_is_writable (uart_inst_t *uart)`
- `static void uart_tx_wait_blocking (uart_inst_t *uart)`

- `static bool uart_is_readable (uart_inst_t *uart)`
- `static void uart_write_blocking (uart_inst_t *uart, const uint8_t *src, size_t len)`
- `static void uart_read_blocking (uart_inst_t *uart, uint8_t *dst, size_t len)`
- `static void uart_putc_raw (uart_inst_t *uart, char c)`
- `static void uart_putc (uart_inst_t *uart, char c)`
- `static void uart_puts (uart_inst_t *uart, const char *s)`
- `static char uart_getc (uart_inst_t *uart)`
- `static void uart_set_break (uart_inst_t *uart, bool en)`
- `void uart_set_translate_crlf (uart_inst_t *uart, bool translate)`
- `static void uart_default_tx_wait_blocking ()`
- `bool uart_is_readable_within_us (uart_inst_t *uart, uint32_t us)`

4.1.23.4. Function Documentation

4.1.23.4.1. uart_default_tx_wait_blocking

`static void uart_default_tx_wait_blocking ()`

Wait for the default UART'S TX fifo to be drained.

4.1.23.4.2. 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

4.1.23.4.3. uart_get_index

`static uint uart_get_index (uart_inst_t *uart)`

Convert UART instance to hardware instance number.

Parameters

- `uart` UART instance

Returns

- Number of UART, 0 or 1.

4.1.23.4.4. uart_getc

`static char uart_getc (uart_inst_t *uart)`

Read a single character to UART.

This function will block until the character has been read

Parameters

- `uart` UART instance. uart0 or uart1

Returns

- The character read.

4.1.23.4.5. 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.

Parameters

- `uart` UART instance. uart0 or uart1
- `baudrate` Baudrate of UART in Hz

Returns

- Actual set baudrate

4.1.23.4.6. uart_is_enabled

```
static bool uart_is_enabled (uart_inst_t *uart)
```

Test if specific UART is enabled.

Parameters

- `uart` UART instance. uart0 or uart1

Returns

- true if the UART is enabled

4.1.23.4.7. uart_is_readable

```
static bool uart_is_readable (uart_inst_t *uart)
```

Determine whether data is waiting in the RX FIFO.

Parameters

- `uart` UART instance. uart0 or uart1

Returns

- 0 if no data available, otherwise the number of bytes, at least, that can be read

4.1.23.4.8. 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

4.1.23.4.9. uart_is_writable

```
static bool uart_is_writable (uart_inst_t *uart)
```

Determine if space is available in the TX FIFO.

Parameters

- `uart` UART instance. uart0 or uart1

Returns

- false if no space available, true otherwise

4.1.23.4.10. uart_putc

```
static void uart_putc (uart_inst_t *uart,
       char c)
```

Write single character to UART for transmission, with optional CR/LF conversions.

This function will block until the character has been sent

Parameters

- `uart` UART instance. uart0 or uart1
- `c` The character to send

4.1.23.4.11. uart_putc_raw

```
static void uart_putc_raw (uart_inst_t *uart,
       char c)
```

Write single character to UART for transmission.

This function will block until all the character has been sent

Parameters

- `uart` UART instance. uart0 or uart1
- `c` The character to send

4.1.23.4.12. uart_puts

```
static void uart_puts (uart_inst_t *uart,
       const char *s)
```

Write string to UART for transmission, doing any CR/LF conversions.

This function will block until the entire string has been sent

Parameters

- `uart` UART instance. uart0 or uart1
- `s` The null terminated string to send

4.1.23.4.13. uart_read_blocking

```
static void uart_read_blocking (uart_inst_t *uart,
       uint8_t *dst,
```

```
    size_t len)
```

Read from the UART.

This function will block until all the data has been received from the UART

Parameters

- `uart` UART instance. uart0 or uart1
- `dst` Buffer to accept received bytes
- `len` The number of bytes to receive.

4.1.23.4.14. uart_set_baudrate

```
uint uart_set_baudrate (uart_inst_t *uart,
    uint baudrate)
```

Set UART baud rate.

Set baud rate as close as possible to requested, and return actual rate selected.

Parameters

- `uart` UART instance. uart0 or uart1
- `baudrate` Baudrate in Hz

4.1.23.4.15. uart_set_break

```
static void uart_set_break (uart_inst_t *uart,
    bool en)
```

Assert a break condition on the UART transmission.

Parameters

- `uart` UART instance. uart0 or uart1
- `en` Assert break condition (TX held low) if true. Clear break condition if false.

4.1.23.4.16. uart_set_fifo_enabled

```
static void uart_set_fifo_enabled (uart_inst_t *uart,
    bool enabled)
```

Enable/Disable the FIFOs on specified UART.

Parameters

- `uart` UART instance. uart0 or uart1
- `enabled` true to enable FIFO (default), false to disable

4.1.23.4.17. uart_set_format

```
static void uart_set_format (uart_inst_t *uart,
    uint data_bits,
    uint stop_bits,
    uart_parity_t parity)
```

Set UART data format.

Configure the data format (bits etc() for the UART

Parameters

- `uart` UART instance. uart0 or uart1
- `data_bits` Number of bits of data. 5..8
- `stop_bits` Number of stop bits 1..2
- `parity` Parity option.

4.1.23.4.18. uart_set_hw_flow

```
static void uart_set_hw_flow (uart_inst_t *uart,
       bool cts,
       bool rts)
```

Set UART flow control CTS/RTS.

Parameters

- `uart` UART instance. uart0 or uart1
- `cts` If true enable flow control of TX by clear-to-send input
- `rts` If true enable assertion of request-to-send output by RX flow control

4.1.23.4.19. uart_set_irq_enables

```
static void uart_set_irq_enables (uart_inst_t *uart,
       bool rx_has_data,
       bool tx_needs_data)
```

Setup UART interrupts.

Enable the UART's interrupt output. An interrupt handler will need to be installed prior to calling this function.

Parameters

- `uart` UART instance. uart0 or uart1
- `rx_has_data` If true an interrupt will be fired when the RX FIFO contain data.
- `tx_needs_data` If true an interrupt will be fired when the TX FIFO needs data.

4.1.23.4.20. 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

4.1.23.4.21. uart_tx_wait_blocking

```
static void uart_tx_wait_blocking (uart_inst_t *uart)
```

Wait for the UART TX fifo to be drained.

Parameters

- `uart` UART instance. uart0 or uart1

4.1.23.4.22. uart_write_blocking

```
static void uart_write_blocking (uart_inst_t *uart,
       const uint8_t *src,
       size_t len)
```

Write to the UART for transmission.

This function will block until all the data has been sent to the UART

Parameters

- `uart` UART instance. uart0 or uart1
- `src` The bytes to send
- `len` The number of bytes to send

4.1.24. hardware_vreg

Voltage Regulation API.

4.1.24.1. Function List

- `void vreg_set_voltage (enum vreg_voltage voltage)`

4.1.24.2. Function Documentation

4.1.24.2.1. vreg_set_voltage

```
void vreg_set_voltage (enum vreg_voltage voltage)
```

Set voltage.

Parameters

- `voltage` The voltage (from enumeration vreg_voltage) to apply to the voltage regulator

4.1.25. hardware_watchdog

Hardware Watchdog Timer API.

Supporting functions for the Pico hardware watchdog timer.

The RP2040 has a built in HW watchdog Timer. This is a countdown timer that can restart parts of the chip if it reaches zero. For example, this can be used to restart the processor if the software running on it gets stuck in an infinite loop or similar. The programmer has to periodically write a value to the watchdog to stop it reaching zero.

Example

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

int main() {
    stdio_init_all();

    if (watchdog_caused_reboot()) {
        printf("Rebooted by Watchdog!\n");
        return 0;
```

```
11      } else {
12          printf("Clean boot\n");
13      }
14
15      // Enable the watchdog, requiring the watchdog to be updated every 100ms or the chip will
    reboot
16      // second arg is pause on debug which means the watchdog will pause when stepping through
    code
17      watchdog_enable(100, 1);
18
19      for (uint i = 0; i < 5; i++) {
20          printf("Updating watchdog %d\n", i);
21          watchdog_update();
22      }
23
24      // Wait in an infinite loop and don't update the watchdog so it reboots us
25      printf("Waiting to be rebooted by watchdog\n");
26      while(1);
27  }
```

4.1.25.1. Function List

- `void watchdog_reboot (uint32_t pc, uint32_t sp, uint32_t delay_ms)`
- `void watchdog_start_tick (uint cycles)`
- `void watchdog_update (void)`
- `void watchdog_enable (uint32_t delay_ms, bool pause_on_debug)`
- `bool watchdog_caused_reboot (void)`
- `uint32_t watchdog_get_count (void)`

4.1.25.2. Function Documentation

4.1.25.2.1. watchdog_caused_reboot

```
bool watchdog_caused_reboot (void)
```

Did the watchdog cause the last reboot?

Returns

- true if the watchdog timer or a watchdog force caused the last reboot
- false there has been no watchdog reboot since run has been

4.1.25.2.2. watchdog_enable

```
void watchdog_enable (uint32_t delay_ms,
      bool pause_on_debug)
```

Enable the watchdog.

By default the SDK assumes a 12MHz XOSC and sets the `watchdog_start_tick` appropriately.

Parameters

- `delay_ms` Number of milliseconds before watchdog will reboot without watchdog_update being called. Maximum of 0x7fffff, which is approximately 8.3 seconds

- `pause_on_debug` If the watchdog should be paused when the debugger is stepping through code

4.1.25.2.3. watchdog_get_count

```
uint32_t watchdog_get_count (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.

4.1.25.2.4. watchdog_reboot

```
void watchdog_reboot (uint32_t pc,
      uint32_t sp,
      uint32_t delay_ms)
```

Define actions to perform at watchdog timeout.

By default the SDK assumes a 12MHz XOSC and sets the `watchdog_start_tick` appropriately.

Parameters

- `pc` If Zero, a standard boot will be performed, if non-zero this is the program counter to jump to on reset.
- `sp` If pc is non-zero, this will be the stack pointer used.
- `delay_ms` Initial load value. Maximum value 0x7fffff, approximately 8.3s.

4.1.25.2.5. watchdog_start_tick

```
void watchdog_start_tick (uint cycles)
```

Start the watchdog tick.

Parameters

- `cycles` This needs to be a divider that when applied to the XOSC input, produces a 1MHz clock. So if the XOSC is 12MHz, this will need to be 12.

4.1.25.2.6. watchdog_update

```
void watchdog_update (void)
```

Reload the watchdog counter with the amount of time set in watchdog_enable.

4.1.26. hardware_xosc

Crystal Oscillator (XOSC) API.

4.1.26.1. Function List

- `void xosc_init (void)`
- `void xosc_disable (void)`
- `void xosc_dormant (void)`

4.1.26.2. Function Documentation

4.1.26.2.1. xosc_disable

```
void xosc_disable (void)
```

Disable the Crystal oscillator.

Turns off the crystal oscillator source, and waits for it to become unstable

4.1.26.2.2. 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.

4.1.26.2.3. 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_multicore	Adds support for running code on the second processor core (core1)
fifo	Functions for inter-core FIFO.
pico_stdlib	Aggregation of a core subset of Pico SDK libraries used by most executables along with some additional utility methods. Including pico_stdlib gives you everything you need to get a basic program running which prints to stdout or flashes a LED.
pico_sync	Synchronization primitives and mutual exclusion.
critical_section	Critical Section API for short-lived mutual exclusion safe for IRQ and multi-core.
mutex	Mutex API for non IRQ mutual exclusion between cores.
sem	Semaphore API for restricting access to a 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_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_multicore

Adds support for running code on the second processor core (core1)

Example

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

#define FLAG_VALUE 123

void core1_entry() {

    multicore_fifo_push_blocking(FLAG_VALUE);

    uint32_t g = multicore_fifo_pop_blocking();

    if (g != FLAG_VALUE)
        printf("Hmm, that's not right on core 1!\n");
    else
        printf("Its all gone well on core 1!");

    while (1)
        tight_loop_contents();
}

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


    multicore_launch_core1(core1_entry);

    // Wait for it to start up

    uint32_t g = multicore_fifo_pop_blocking();

    if (g != FLAG_VALUE)
        printf("Hmm, that's not right on core 0!\n");
    else {
        multicore_fifo_push_blocking(FLAG_VALUE);
        printf("It's all gone well on core 0!");
    }

}
```

4.2.1.1. Modules

- fifo
  Functions for inter-core FIFO.

4.2.1.2. Function List

- `void multicore_reset_core1 ()`
- `void multicore_launch_core1 (void(*entry)(void))`
- `void multicore_launch_core1_with_stack (void(*entry)(void), uint32_t *stack_bottom, size_t stack_size_bytes)`
- `void multicore_sleep_core1 ()`
- `void multicore_launch_core1_raw (void(*entry)(void), uint32_t *sp, uint32_t vector_table)`

4.2.1.3. Function Documentation

4.2.1.3.1. multicore_launch_core1

```
void multicore_launch_core1 (void(*entry)(void))
```

Run code on core 1.

Reset core1 and enter the given function on core 1 using the default core 1 stack (below core 0 stack)

Parameters

- `entry` Function entry point, this function should not return.

4.2.1.3.2. multicore_launch_core1_raw

```
void multicore_launch_core1_raw (void(*entry)(void),
      uint32_t *sp,
      uint32_t vector_table)
```

Launch code on core 1 with no stack protection.

Reset core1 and enter the given function using the passed sp as the initial stack pointer. This is a bare bones functions that does not provide a stack guard even if USE_STACK_GUARDS is defined

4.2.1.3.3. multicore_launch_core1_with_stack

```
void multicore_launch_core1_with_stack (void(*entry)(void),
      uint32_t *stack_bottom,
      size_t stack_size_bytes)
```

Launch code on core 1 with stack.

Reset core1 and enter the given function on core 1 using the passed stack for core 1

4.2.1.3.4. multicore_reset_core1

```
void multicore_reset_core1 ()
```

Reset Core 1.

4.2.1.3.5. multicore_sleep_core1

```
void multicore_sleep_core1 ()
```

Send core 1 to sleep.

4.2.2. fifo

Functions for inter-core FIFO.

The RP2040 contains two FIFOs for passing data, messages or ordered events between the two cores. Each FIFO is 32 bits wide, and 8 entries deep. One of the FIFOs can only be written by core 0, and read by core 1. The other can only be written by core 1, and read by core 0.

4.2.2.1. Function List

- `static bool multicore_fifo_rvalid ()`
- `static bool multicore_fifo_wready ()`
- `void multicore_fifo_push_blocking (uint32_t data)`
- `uint32_t multicore_fifo_pop_blocking ()`
- `static void multicore_fifo_drain ()`
- `static void multicore_fifo_clear_irq ()`
- `static int32_t multicore_fifo_get_status ()`

4.2.2.2. Function Documentation

4.2.2.2.1. multicore_fifo_clear_irq

`static void multicore_fifo_clear_irq ()`

Clear FIFO interrupt.

4.2.2.2.2. multicore_fifo_drain

`static void multicore_fifo_drain ()`

Flush any data in the outgoing FIFO.

4.2.2.2.3. multicore_fifo_get_status

`static int32_t multicore_fifo_get_status ()`

Get FIFO status.

Bit	Description
3	Sticky flag indicating the RX FIFO was read when empty. This read was ignored by the FIFO.
2	Sticky flag indicating the TX FIFO was written when full. This write was ignored by the FIFO.
1	Value is 1 if this core's TX FIFO is not full (i.e. if FIFO_WR is ready for more data)
0	Value is 1 if this core's RX FIFO is not empty (i.e. if FIFO_RD is valid)

Returns

- The status as a bitfield

4.2.2.2.4. multicore_fifo_pop_blocking

`uint32_t multicore_fifo_pop_blocking ()`

Pop data from the FIFO.

This function will block until there is data ready to be read Use `multicore_fifo_rvalid()` to check if data is ready to be read if you don't want to block.

Returns

- 32 bit unsigned data from the FIFO.

4.2.2.2.5. multicore_fifo_push_blocking

`void multicore_fifo_push_blocking (uint32_t data)`

Push data on to the FIFO.

This function will block until there is space for the data to be sent. Use `multicore_fifo_wready()` to check if it is possible to write to the FIFO if you don't want to block.

Parameters

- `data` A 32 bit value to push on to the FIFO

4.2.2.2.6. multicore_fifo_rvalid

`static bool multicore_fifo_rvalid ()`

Check the read FIFO to see if there is data waiting.

Returns

- true if the FIFO has data in it, false otherwise

4.2.2.2.7. multicore_fifo_wready

`static bool multicore_fifo_wready ()`

Check the FIFO to see if the write FIFO is full.

Returns

- true if the FIFO is full, false otherwise

4.2.3. pico_stdlib

Aggregation of a core subset of Pico SDK libraries used by most executables along with some additional utility methods. Including pico_stdlib gives you everything you need to get a basic program running which prints to stdout or flashes a LED.

This library aggregates:

- hardware_uart
- hardware_gpio
- pico_binary_info
- pico_runtime
- pico_platform
- pico_printf

- pico_stdio
- pico_standard_link
- pico_util

There are some basic default values used by these functions that will default to usable values, however, they can be customised in a board definition header via config.h or similar

4.2.3.1. Function List

- `void setup_default_uart ()`
- `void set_sys_clock_48mhz ()`
- `void set_sys_clock_pll (uint32_t vco_freq, uint post_div1, uint post_div2)`
- `bool check_sys_clock_khz (uint32_t freq_khz, uint *vco_freq_out, uint *post_div1_out, uint *post_div2_out)`
- `static bool set_sys_clock_khz (uint32_t freq_khz, bool required)`

4.2.3.2. Function Documentation

4.2.3.2.1. check_sys_clock_khz

```
bool check_sys_clock_khz (uint32_t freq_khz,
       uint *vco_freq_out,
       uint *post_div1_out,
       uint *post_div2_out)
```

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

Parameters

- `freq_khz` Requested frequency
- `vco_freq_out` On success, the voltage controller oscillator frequeucny to be used by the SYS PLL
- `post_div1_out` On success, The first post divider for the SYS PLL
- `post_div2_out` On success, The second post divider for the SYS PLL.

Returns

- true if the frequency is possible and the output parameters have been written.

4.2.3.2.2. set_sys_clock_48mhz

```
void set_sys_clock_48mhz ()
```

Initialise the system clock to 48MHz.

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

4.2.3.2.3. set_sys_clock_khz

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

Attempt to set a system clock frequency in khz.

Note that not all clock frequencies are possible; it is preferred that you use src/rp2_common/hardware_clocks/scripts/vcocalc.py to calculate the parameters for use with 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

4.2.3.2.4. set_sys_clock_pll

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

Initialise the system clock.

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

Parameters

- `vco_freq` The voltage controller oscillator 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.

4.2.3.2.5. setup_default_uart

```
void setup_default_uart ()
```

Set up the default UART and assign it to the default GPIO's.

By default this will use UART 0, with TX to pin GPIO 0, RX to pin GPIO 1, and the baudrate to 115200

Calling this method also 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

4.2.4. pico_sync

Synchronization primitives and mutual exclusion.

4.2.4.1. Modules

- `critical_section`
  Critical Section API for short-lived mutual exclusion safe for IRQ and multi-core.
- `mutex`
  Mutex API for non IRQ mutual exclusion between cores.
- `sem`
  Semaphore API for restricting access to a resource.

4.2.5. critical_section

Critical Section API for short-lived mutual exclusion safe for IRQ and multi-core.

A critical section is non-reentrant, and provides mutual exclusion using a spin-lock to prevent access from the other core, and from (higher priority) interrupts on the same core. It does the former using a spin lock and the latter by disabling

interrupts on the calling core.

Because interrupts are disabled by this function, uses of the critical_section should be as short as possible.

4.2.5.1. Function List

- `void critical_section_init (critical_section_t *critsec)`
- `void critical_section_init_with_lock_num (critical_section_t *critsec, uint lock_num)`
- `static void critical_section_enter_blocking (critical_section_t *critsec)`
- `static void critical_section_exit (critical_section_t *critsec)`

4.2.5.2. Function Documentation

4.2.5.2.1. critical_section_enter_blocking

```
static void critical_section_enter_blocking (critical_section_t *critsec)
```

Enter a critical_section.

If the spin lock associated with this critical section is in use, then this method will block until it is released.

Parameters

- `critsec` Pointer to critical_section structure

4.2.5.2.2. critical_section_exit

```
static void critical_section_exit (critical_section_t *critsec)
```

Release a critical_section.

Parameters

- `critsec` Pointer to critical_section structure

4.2.5.2.3. critical_section_init

```
void critical_section_init (critical_section_t *critsec)
```

Initialise a critical_section structure allowing the system to assign a spin lock number.

The critical section is initialized ready for use, and will use a (possibly shared) spin lock number assigned by the system. Note that in general it is unlikely that you would be nesting critical sections, however if you do so you use `critical_section_init_with_lock_num` to ensure that the spin lock's used are different.

Parameters

- `critsec` Pointer to critical_section structure

4.2.5.2.4. critical_section_init_with_lock_num

```
void critical_section_init_with_lock_num (critical_section_t *critsec,
      uint lock_num)
```

Initialise a critical_section structure assigning a specific spin lock number.

Parameters

- `critsec` Pointer to critical_section structure

- `lock_num` the specific spin lock number to use

4.2.6. mutex

Mutex API for non IRQ mutual exclusion between cores.

Mutexes are application level locks usually used protecting data structures that might be used by multiple cores. Unlike critical sections, the mutex protected code is not necessarily required/expected to complete quickly, as no other sytemwide locks are held on account of a locked mutex.

Because they are not re-entrant on the same core, blocking on a mutex should never be done in an IRQ handler. It is valid to call mutex_try_enter from within an IRQ handler, if the operation that would be conducted under lock can be skipped if the mutex is locked (at least by the same core).

See critical_section.h for protecting access between multiple cores AND IRQ handlers

4.2.6.1. Macros

- `#define auto_init_mutex(name) static __attribute__((section(".mutex_array"))) mutex_t name`
  Helper macro for static definition of mutexes.

4.2.6.2. Function List

- `void mutex_init (mutex_t *mtx)`
- `void mutex_enter_blocking (mutex_t *mtx)`
- `bool mutex_try_enter (mutex_t *mtx, uint32_t *owner_out)`
- `bool mutex_enter_timeout_ms (mutex_t *mtx, uint32_t timeout_ms)`
- `bool mutex_enter_block_until (mutex_t *mtx, absolute_time_t until)`
- `void mutex_exit (mutex_t *mtx)`
- `static bool mutex_is_initialzed (mutex_t *mtx)`

4.2.6.3. Function Documentation

4.2.6.3.1. mutex_enter_block_until

```
bool mutex_enter_block_until (mutex_t *mtx,
      absolute_time_t until)
```

Wait for mutex until a specific time.

Wait until the specific time to take ownership of the mutex. If the calling core can take ownership of the mutex before the timeout expires, then true will be returned and the calling core will own the mutex, otherwise false will be returned and the calling core will own the mutex.

Parameters

- `mtx` Pointer to mutex structure
- `until` The time after which to return if the core cannot take owner ship of the mutex

Returns

- true if mutex now owned, false if timeout occurred before mutex became available

4.2.6.3.2. mutex_enter_blocking

```
void mutex_enter_blocking (mutex_t *mtx)
```

Take ownership of a mutex.

This function will block until the calling core can claim ownership of the mutex. On return the caller core owns the mutex

Parameters

- `mtx` Pointer to mutex structure

4.2.6.3.3. mutex_enter_timeout_ms

```
bool mutex_enter_timeout_ms (mutex_t *mtx,
      uint32_t timeout_ms)
```

Wait for mutex with timeout.

Wait for up to the specific time to take ownership of the mutex. If the calling core can take ownership of the mutex before the timeout expires, then true will be returned and the calling core will own the mutex, otherwise false will be returned and the calling core will own the mutex.

Parameters

- `mtx` Pointer to mutex structure
- `timeout_ms` The timeout in milliseconds.

Returns

- true if mutex now owned, false if timeout occurred before mutex became available

4.2.6.3.4. mutex_exit

```
void mutex_exit (mutex_t *mtx)
```

Release ownership of a mutex.

Parameters

- `mtx` Pointer to mutex structure

4.2.6.3.5. mutex_init

```
void mutex_init (mutex_t *mtx)
```

Initialise a mutex structure.

Parameters

- `mtx` Pointer to mutex structure

4.2.6.3.6. mutex_is_initialzed

```
static bool mutex_is_initialzed (mutex_t *mtx)
```

Test for mutex initialised state.

Parameters

- `mtx` Pointer to mutex structure

Returns

- true if the mutex is initialised, false otherwise

4.2.6.3.7. mutex_try_enter

```
bool mutex_try_enter (mutex_t *mtx,
      uint32_t *owner_out)
```

Check to see if a mutex is available.

Will return true if the mutex is unowned, false otherwise

Parameters

- `mtx` Pointer to mutex structure
- `owner_out` If mutex is owned, and this pointer is non-zero, it will be filled in with the core number of the current owner of the mutex

4.2.7. sem

Semaphore API for restricting access to a resource.

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)

4.2.7.1. Function List

- `void sem_init (semaphore_t *sem, int16_t initial_permits, int16_t max_permits)`
- `int sem_available (semaphore_t *sem)`
- `bool sem_release (semaphore_t *sem)`
- `void sem_reset (semaphore_t *sem, int16_t permits)`
- `void sem_acquire_blocking (semaphore_t *sem)`
- `bool sem_acquire_timeout_ms (semaphore_t *sem, uint32_t timeout_ms)`

4.2.7.2. Function Documentation

4.2.7.2.1. sem_acquire_blocking

```
void sem_acquire_blocking (semaphore_t *sem)
```

Acquire a permit from the semaphore.

This function will block and wait if no permits are available.

Parameters

- `sem` Pointer to semaphore structure

4.2.7.2.2. sem_acquire_timeout_ms

```
bool sem_acquire_timeout_ms (semaphore_t *sem,
      uint32_t timeout_ms)
```

Acquire a permit from a semaphore, with timeout.

This function will block and wait if no permits are available, until the defined timeout has been reached. If the timeout is reached the function will return false, otherwise it will return true.

Parameters

- `sem` Pointer to semaphore structure
- `timeout_ms` Time to wait to acquire the semaphore, in ms.

Returns

- false if timeout reached, true if permit was acquired.

4.2.7.2.3. sem_available

```
int sem_available (semaphore_t *sem)
```

Return number of available permits on the semaphore.

Parameters

- `sem` Pointer to semaphore structure

Returns

- The number of permits available on the semaphore.

4.2.7.2.4. sem_init

```
void sem_init (semaphore_t *sem,
       int16_t initial_permits,
       int16_t max_permits)
```

Initialise a semaphore structure.

Parameters

- `sem` Pointer to semaphore structure
- `initial_permits` How many permits are initially acquired
- `max_permits` Total number of permits allowed for this semaphore

4.2.7.2.5. sem_release

```
bool sem_release (semaphore_t *sem)
```

Release a permit on a semaphore.

Increases the number of permits by one (unless the number of permits is already at the maximum). A blocked `sem_acquire` will be released if the number of permits is increased.

Parameters

- `sem` Pointer to semaphore structure

Returns

- true if the number of permits available was increased.

4.2.7.2.6. sem_reset

```
void sem_reset (semaphore_t *sem,
       int16_t permits)
```

Reset semaphore to a specific number of available permits.

Reset value should be from 0 to the max_permits specified in the init function

Parameters

- `sem` Pointer to semaphore structure
- `permits` the new number of available permits

4.2.8. pico_time

API for accurate timestamps, sleeping, and time based callbacks.

The functions defined here provide a much more powerful and user friendly wrapping around the low level hardware timer functionality. For these functions (and any other Pico SDK functionality e.g. timeouts, that relies on them) to work correctly, the hardware timer should not be modified. i.e. it is expected to be monotonically increasing once per microsecond. Fortunately there is no need to modify the hardware timer as any functionality you can think of that isn't already covered here can easily be modelled by adding or subtracting a constant value from the unmodified hardware timer.

- hardware_timer

4.2.8.1. Modules

- `timestamp`
  Timestamp functions relating to points in time (including the current time)
- `sleep`
  Sleep functions for delaying execution in a lower power state.
- `alarm`
  Alarm functions for scheduling future execution.
- `repeating_timer`
  Repeating Timer functions for simple scheduling of repeated execution.

4.2.9. timestamp

Timestamp functions relating to points in time (including the current time)

These are functions for dealing with timestamps (i.e. instants in time) represented by the type absolute_time_t. This opaque type is provided to help prevent accidental mixing of timestamps and relative time values.

4.2.9.1. Function List

- `static absolute_time_t get_absolute_time ()`
- `static uint32_t to_ms_since_boot (absolute_time_t t)`
- `static absolute_time_t delayed_by_us (const absolute_time_t t, uint64_t us)`
- `static absolute_time_t delayed_by_ms (const absolute_time_t t, uint32_t ms)`
- `static absolute_time_t make_timeout_time_us (uint64_t us)`
- `static absolute_time_t make_timeout_time_ms (uint32_t ms)`
- `static int64_t absolute_time_diff_us (absolute_time_t from, absolute_time_t to)`
- `static bool is_nil_time (absolute_time_t t)`

4.2.9.2. Function Documentation

4.2.9.2.1. absolute_time_diff_us

```
static int64_t absolute_time_diff_us (absolute_time_t from,
       absolute_time_t to)
```

Return the difference in microseconds between two timestamps.

Parameters

- `from` the first timestamp
- `to` the second timestamp

Returns

- the number of microseconds between the two timestamps (positive if `to` is after `from`)

4.2.9.2.2. delayed_by_ms

```
static absolute_time_t delayed_by_ms (const absolute_time_t t,
       uint32_t ms)
```

Return a timestamp value obtained by adding a number of milliseconds to another timestamp.

Parameters

- `t` the base timestamp
- `ms` the number of milliseconds to add

Returns

- the timestamp representing the resulting time

4.2.9.2.3. delayed_by_us

```
static absolute_time_t delayed_by_us (const absolute_time_t t,
       uint64_t us)
```

Return a timestamp value obtained by adding a number of microseconds to another timestamp.

Parameters

- `t` the base timestamp
- `us` the number of microseconds to add

Returns

- the timestamp representing the resulting time

4.2.9.2.4. get_absolute_time

```
static absolute_time_t get_absolute_time ()
```

Return a representation of the current time.

Returns an opaque high fidelity representation of the current time sampled during the call.

Returns

- the absolute time (now) of the hardware timer

See also

- absolute_time_t
- sleep_until()
- time_us_64()

4.2.9.2.5. is_nil_time

```
static bool is_nil_time (absolute_time_t t)
```

Determine if the given timestamp is nil.

Parameters

- `t` the timestamp

Returns

- true if the timestamp is nil

See also

- nil_time()

4.2.9.2.6. make_timeout_time_ms

```
static absolute_time_t make_timeout_time_ms (uint32_t ms)
```

Convenience method to get the timestamp a number of milliseconds from the current time.

Parameters

- `ms` the number of milliseconds to add to the current timestamp

Returns

- the future timestamp

4.2.9.2.7. make_timeout_time_us

```
static absolute_time_t make_timeout_time_us (uint64_t us)
```

Convenience method to get the timestamp a number of microseconds from the current time.

Parameters

- `us` the number of microseconds to add to the current timestamp

Returns

- the future timestamp

4.2.9.2.8. to_ms_since_boot

```
static uint32_t to_ms_since_boot (absolute_time_t t)
```

Convert a timestamp into a number of milliseconds since boot.

fn to_ms_since_boot

Parameters

- `t` an absolute_time_t value to convert

Returns

- the number of microseconds since boot represented by t

See also

- to_us_since_boot

4.2.10. sleep

Sleep functions for delaying execution in a lower power state.

These functions allow the calling core to sleep. This is a lower powered sleep; waking and re-checking time on every processor event (WFE)

These functions should not be called from an IRQ handler.

Lower powered sleep requires use of the default alarm pool which may be disabled by the PICO_TIME_DEFAULT_ALARM_POOL_DISABLED define or currently full in which case these functions become busy waits instead.

Whilst *sleep_* functions are preferable to *busy_wait* functions from a power perspective, the *busy_wait* equivalent function may return slightly sooner after the target is reached.

- busy_wait_until()
- busy_wait_us()
- busy_wait_us_32()

4.2.10.1. Function List

- `void sleep_until (absolute_time_t target)`
- `void sleep_us (uint64_t us)`
- `void sleep_ms (uint32_t ms)`
- `bool best_effort_wfe_or_timeout (absolute_time_t timeout_timestamp)`

4.2.10.2. Function Documentation

4.2.10.2.1. best_effort_wfe_or_timeout

`bool best_effort_wfe_or_timeout (absolute_time_t timeout_timestamp)`

Helper method for blocking on a timeout.

This method will return in response to a an event (as per __wfe) or when the target time is reached, or at any point before.

This method can be used to implement a lower power polling loop waiting on some condition signalled by an event (`__sev()`).

This is called because under certain circumstances (notably the default timer pool being disabled or full) the best effort is simply to return immediately without a __wfe, thus turning the calling code into a busy wait.

Example usage:

Parameters

- `timeout_timestamp` the timeout time

Returns

- true if the target time is reached, false otherwise

4.2.10.2.2. sleep_ms

```
void sleep_ms (uint32_t ms)
```

Wait for the given number of milliseconds before returning.

Parameters

- `ms` the number of milliseconds to sleep

4.2.10.2.3. sleep_until

```
void sleep_until (absolute_time_t target)
```

Wait until after the given timestamp to return.

Parameters

- `target` the time after which to return

See also

- sleep_us()
- busy_wait_until()

4.2.10.2.4. sleep_us

```
void sleep_us (uint64_t us)
```

Wait for the given number of microseconds before returning.

Parameters

- `us` the number of microseconds to sleep

See also

- busy_wait_us()

4.2.11. alarm

Alarm functions for scheduling future execution.

Alarms are added to alarm pools, which may hold a certain fixed number of active alarms. Each alarm pool utilizes one of four underlying hardware alarms, thus you may have up to four alarm pools. An alarm pool calls (except when the callback would happen before or during being set) the callback on the core from which the alarm pool was created. Callbacks are called from the hardware alarm IRQ handler, so care must be taken in their implementation.

A default pool is created 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

4.2.11.1. Macros

- `#define PICO_TIME_DEFAULT_ALARM_POOL_DISABLED 0`
  If 1 then the default alarm pool is disabled (so no hardware alarm is claimed for the pool)
- `#define PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM 3`
  Selects which hardware alarm is used for the default alarm pool.

- `#define PICO_TIME_DEFAULT_ALARM_POOL_MAX_TIMERS 16`
  Selects the maximum number of concurrent timers in the default alarm pool.

4.2.11.2. 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.

4.2.11.3. Function List

- `void alarm_pool_init_default ()`
- `alarm_pool_t * alarm_pool_get_default ()`
- `alarm_pool_t * alarm_pool_create (uint hardware_alarm_num, uint max_timers)`
- `uint alarm_pool_hardware_alarm_num (alarm_pool_t *pool)`
- `void alarm_pool_destroy (alarm_pool_t *pool)`
- `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)`
- `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)`
- `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)`
- `bool alarm_pool_cancel_alarm (alarm_pool_t *pool, alarm_id_t alarm_id)`
- `static alarm_id_t add_alarm_at (absolute_time_t time, alarm_callback_t callback, void *user_data, bool fire_if_past)`
- `static alarm_id_t add_alarm_in_us (uint64_t us, alarm_callback_t callback, void *user_data, bool fire_if_past)`
- `static alarm_id_t add_alarm_in_ms (uint32_t ms, alarm_callback_t callback, void *user_data, bool fire_if_past)`
- `static bool cancel_alarm (alarm_id_t alarm_id)`

4.2.11.4. Function Documentation

4.2.11.4.1. add_alarm_at

```
static alarm_id_t add_alarm_at (absolute_time_t time,
       alarm_callback_t callback,
       void *user_data,
       bool fire_if_past)
```

Add an alarm callback to be called at a specific time.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core of the default alarm pool (generally core 0). If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- `time` the timestamp when (after which) the callback should fire
- `callback` the callback function

- `user_data` user data to pass to the callback function
- `fire_if_past` if true, this method will call the callback itself before returning 0 if the timestamp happens before or during this method call

Returns

- >0 the alarm id
- 0 the target timestamp was during or before this method call (whether the callback was called depends on fire_if_past)
- -1 if there were no alarm slots available

4.2.11.4.2. add_alarm_in_ms

```
static alarm_id_t add_alarm_in_ms (uint32_t ms,
       alarm_callback_t callback,
       void *user_data,
       bool fire_if_past)
```

Add an alarm callback to be called after a delay specified in milliseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core of the default alarm pool (generally core 0). If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- `ms` the delay (from now) in milliseconds when (after which) the callback should fire
- `callback` the callback function
- `user_data` user data to pass to the callback function
- `fire_if_past` if true, this method will call the callback itself before returning 0 if the timestamp happens before or during this method call

Returns

- >0 the alarm id
- 0 the target timestamp was during or before this method call (whether the callback was called depends on fire_if_past)
- -1 if there were no alarm slots available

4.2.11.4.3. add_alarm_in_us

```
static alarm_id_t add_alarm_in_us (uint64_t us,
       alarm_callback_t callback,
       void *user_data,
       bool fire_if_past)
```

Add an alarm callback to be called after a delay specified in microseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core of the default alarm pool (generally core 0). If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- `us` the delay (from now) in microseconds when (after which) the callback should fire
- `callback` the callback function
- `user_data` user data to pass to the callback function

- `fire_if_past` if true, this method will call the callback itself before returning 0 if the timestamp happens before or during this method call

Returns

- >0 the alarm id
- 0 the target timestamp was during or before this method call (whether the callback was called depends on fire_if_past)
- -1 if there were no alarm slots available

4.2.11.4.4. alarm_pool_add_alarm_at

```
alarm_id_t alarm_pool_add_alarm_at (alarm_pool_t *pool,
        absolute_time_t time,
        alarm_callback_t callback,
        void *user_data,
        bool fire_if_past)
```

Add an alarm callback to be called at a specific time.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core the alarm pool was created on. If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- `pool` the alarm pool to use for scheduling the callback (this determines which hardware alarm is used, and which core calls the callback)
- `time` the timestamp when (after which) the callback should fire
- `callback` the callback function
- `user_data` user data to pass to the callback function
- `fire_if_past` if true, this method will call the callback itself before returning 0 if the timestamp happens before or during this method call

Returns

- >0 the alarm id
- 0 the target timestamp was during or before this method call (whether the callback was called depends on fire_if_past)
- -1 if there were no alarm slots available

4.2.11.4.5. alarm_pool_add_alarm_in_ms

```
static alarm_id_t alarm_pool_add_alarm_in_ms (alarm_pool_t *pool,
        uint32_t ms,
        alarm_callback_t callback,
        void *user_data,
        bool fire_if_past)
```

Add an alarm callback to be called after a delay specified in milliseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core the alarm pool was created on. If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- `pool` the alarm pool to use for scheduling the callback (this determines which hardware alarm is used, and which

core calls the callback)

- `ms` the delay (from now) in milliseconds when (after which) the callback should fire
- `callback` the callback function
- `user_data` user data to pass to the callback function
- `fire_if_past` if true, this method will call the callback itself before returning 0 if the timestamp happens before or during this method call

Returns

- >0 the alarm id
- 0 the target timestamp was during or before this method call (whether the callback was called depends on fire_if_past)
- -1 if there were no alarm slots available

4.2.11.4.6. alarm_pool_add_alarm_in_us

```
static alarm_id_t alarm_pool_add_alarm_in_us (alarm_pool_t *pool,
       uint64_t us,
       alarm_callback_t callback,
       void *user_data,
       bool fire_if_past)
```

Add an alarm callback to be called after a delay specified in microseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core the alarm pool was created on. If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- `pool` the alarm pool to use for scheduling the callback (this determines which hardware alarm is used, and which core calls the callback)
- `us` the delay (from now) in microseconds when (after which) the callback should fire
- `callback` the callback function
- `user_data` user data to pass to the callback function
- `fire_if_past` if true, this method will call the callback itself before returning 0 if the timestamp happens before or during this method call

Returns

- >0 the alarm id
- 0 the target timestamp was during or before this method call (whether the callback was called depends on fire_if_past)
- -1 if there were no alarm slots available

4.2.11.4.7. alarm_pool_cancel_alarm

```
bool alarm_pool_cancel_alarm (alarm_pool_t *pool,
       alarm_id_t alarm_id)
```

Cancel an alarm.

Parameters

- `pool` the alarm_pool containing the alarm

- `alarm_id` the alarm

Returns

- true if the alarm was cancelled, false if it didn't exist

See also

- alarm_id_t for a note on reuse of IDs

4.2.11.4.8. alarm_pool_create

```
alarm_pool_t* alarm_pool_create (uint hardware_alarm_num,
      uint max_timers)
```

Create an alarm pool.

The alarm pool will call callbacks from an alarm IRQ Handler on the core of this function is called from.

In many situations there is never any need for anything other than the default alarm pool, however you might want to create another if you want alarm callbacks on core 1 or require alarm pools of different priority (IRQ priority based preemption of callbacks)

Parameters

- `hardware_alarm_num` the hardware alarm to use to back this pool
- `max_timers` the maximum number of timers

See also

- alarm_pool_get_default()
- hardware_claiming

4.2.11.4.9. alarm_pool_destroy

```
void alarm_pool_destroy (alarm_pool_t *pool)
```

Destroy the alarm pool, cancelling all alarms and freeing up the underlying hardware alarm.

Parameters

- `pool` the pool

Returns

- the hardware alarm used by the pool

4.2.11.4.10. alarm_pool_get_default

```
alarm_pool_t* alarm_pool_get_default ()
```

The default alarm pool used when alarms are added without specifying an alarm pool, and also used by the Pico SDK to support lower power sleeps and timeouts.

See also

- PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM

4.2.11.4.11. alarm_pool_hardware_alarm_num

```
uint alarm_pool_hardware_alarm_num (alarm_pool_t *pool)
```

Return the hardware alarm used by an alarm pool.

Parameters

- `pool` the pool

Returns

- the hardware alarm used by the pool

4.2.11.4.12. alarm_pool_init_default

```
void alarm_pool_init_default ()
```

Create the default alarm pool (if not already created or disabled)

4.2.11.4.13. cancel_alarm

```
static bool cancel_alarm (alarm_id_t alarm_id)
```

Cancel an alarm from the default alarm pool.

Parameters

- `alarm_id` the alarm

Returns

- true if the alarm was cancelled, false if it didn't exist

See also

- alarm_id_t for a note on reuse of IDs

4.2.12. repeating_timer

Repeating Timer functions for simple scheduling of repeated execution.

The regular *alarm_* functionality can be used to make repeating alarms (by return non zero from the callback), however these methods abstract that further (at the cost of a user structure to store the repeat delay in (which the alarm framework does not have space for).

4.2.12.1. Data Structures

- `struct repeating_timer`
  Information about a repeating timer.

4.2.12.2. Typedefs

- `typedef bool(* repeating_timer_callback_t )(repeating_timer_t *rt)`
  Callback for a repeating timer.

4.2.12.3. Function List

- `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)`
- `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)`
- `static bool add_repeating_timer_us (int64_t delay_us, repeating_timer_callback_t callback, void *user_data, repeating_timer_t *out)`
- `static bool add_repeating_timer_ms (int32_t delay_ms, repeating_timer_callback_t callback, void *user_data, repeating_timer_t *out)`

- `bool cancel_repeating_timer (repeating_timer_t *timer)`

4.2.12.4. Function Documentation

4.2.12.4.1. add_repeating_timer_ms

```
static bool add_repeating_timer_ms (int32_t delay_ms,
       repeating_timer_callback_t callback,
       void *user_data,
       repeating_timer_t *out)
```

Add a repeating timer that is called repeatedly at the specified interval in milliseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core of the default alarm pool (generally core 0). If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- `delay_ms` the repeat delay in milliseconds; if >0 then this is the delay between one callback ending and the next starting; if <0 then this is the negative of the time between the starts of the callbacks. The value of 0 is treated as 1 microsecond
- `callback` the repeating timer callback function
- `user_data` user data to pass to store in the repeating_timer structure for use by the callback.
- `out` the pointer to the user owned structure to store the repeating timer info in. BEWARE this storage location must outlive the repeating timer, so be careful of using stack space

Returns

- false if there were no alarm slots available to create the timer, true otherwise.

4.2.12.4.2. add_repeating_timer_us

```
static bool add_repeating_timer_us (int64_t delay_us,
       repeating_timer_callback_t callback,
       void *user_data,
       repeating_timer_t *out)
```

Add a repeating timer that is called repeatedly at the specified interval in microseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core of the default alarm pool (generally core 0). If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- `delay_us` the repeat delay in microseconds; if >0 then this is the delay between one callback ending and the next starting; if <0 then this is the negative of the time between the starts of the callbacks. The value of 0 is treated as 1
- `callback` the repeating timer callback function
- `user_data` user data to pass to store in the repeating_timer structure for use by the callback.
- `out` the pointer to the user owned structure to store the repeating timer info in. BEWARE this storage location must outlive the repeating timer, so be careful of using stack space

Returns

- false if there were no alarm slots available to create the timer, true otherwise.

4.2.12.4.3. alarm_pool_add_repeating_timer_ms

```
static bool alarm_pool_add_repeating_timer_ms (alarm_pool_t *pool,
       int32_t delay_ms,
       repeating_timer_callback_t callback,
       void *user_data,
       repeating_timer_t *out)
```

Add a repeating timer that is called repeatedly at the specified interval in milliseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core the alarm pool was created on. If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- `pool` the alarm pool to use for scheduling the repeating timer (this determines which hardware alarm is used, and which core calls the callback)
- `delay_ms` the repeat delay in milliseconds; if >0 then this is the delay between one callback ending and the next starting; if <0 then this is the negative of the time between the starts of the callbacks. The value of 0 is treated as 1 microsecond
- `callback` the repeating timer callback function
- `user_data` user data to pass to store in the repeating_timer structure for use by the callback.
- `out` the pointer to the user owned structure to store the repeating timer info in. BEWARE this storage location must outlive the repeating timer, so be careful of using stack space

Returns

- false if there were no alarm slots available to create the timer, true otherwise.

4.2.12.4.4. alarm_pool_add_repeating_timer_us

```
bool alarm_pool_add_repeating_timer_us (alarm_pool_t *pool,
       int64_t delay_us,
       repeating_timer_callback_t callback,
       void *user_data,
       repeating_timer_t *out)
```

Add a repeating timer that is called repeatedly at the specified interval in microseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core the alarm pool was created on. If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

Parameters

- `pool` the alarm pool to use for scheduling the repeating timer (this determines which hardware alarm is used, and which core calls the callback)
- `delay_us` the repeat delay in microseconds; if >0 then this is the delay between one callback ending and the next starting; if <0 then this is the negative of the time between the starts of the callbacks. The value of 0 is treated as 1
- `callback` the repeating timer callback function
- `user_data` user data to pass to store in the repeating_timer structure for use by the callback.
- `out` the pointer to the user owned structure to store the repeating timer info in. BEWARE this storage location must outlive the repeating timer, so be careful of using stack space

Returns

- false if there were no alarm slots available to create the timer, true otherwise.

4.2.12.4.5. cancel_repeating_timer

```
bool cancel_repeating_timer (repeating_timer_t *timer)
```

Cancel a repeating timer.

Parameters

- `timer` the repeating timer to cancel

Returns

- true if the repeating timer was cancelled, false if it didn't exist

See also

- alarm_id_t for a note on reuse of IDs

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.

4.2.14. datetime

Date/Time formatting.

4.2.14.1. Data Structures

- `struct datetime_t`
  Structure containing date and time information.

4.2.14.2. Function List

- `void datetime_to_str (char *buf, uint buf_size, const datetime_t *t)`

4.2.14.3. Function Documentation

4.2.14.3.1. datetime_to_str

```
void datetime_to_str (char *buf,
       uint buf_size,
       const datetime_t *t)
```

Convert a datetime_t structure to a string.

Parameters

- `buf` character buffer to accept generated string
- `buf_size` The size of the passed in buffer
- `t` The datetime to be converted.

4.2.15. pheap

Pairing Heap Implementation.

pheap defines a simple pairing heap. the implementation simply tracks array indexes, it is up to the user to provide storage for heap entries and a comparison function.

NOTE

this class is not safe for concurrent usage. It should be externally protected. Furthermore if used concurrently, the caller needs to protect around their use of the returned id. for example, ph_remove_head returns the id of an element that is no longer in the heap.

The user can still use this to look at the data in their companion array, however obviously further operations on the heap may cause them to overwrite that data as the id may be reused on subsequent operations

4.2.16. queue

Multi-core and IRQ safe queue implementation.

Note that this queue stores values of a specified size, and pushed values are copied into the queue

4.2.16.1. Function List

- `void queue_init_with_spinlock (queue_t *q, uint element_size, uint element_count, uint spinlock_num)`
- `static void queue_init (queue_t *q, uint element_size, uint element_count)`
- `void queue_free (queue_t *q)`
- `static uint queue_get_level_unsafe (queue_t *q)`
- `static uint queue_get_level (queue_t *q)`
- `static bool queue_is_empty (queue_t *q)`
- `static bool queue_is_full (queue_t *q)`
- `bool queue_try_add (queue_t *q, void *data)`
- `bool queue_try_remove (queue_t *q, void *data)`
- `bool queue_try_peek (queue_t *q, void *data)`
- `void queue_add_blocking (queue_t *q, void *data)`
- `void queue_remove_blocking (queue_t *q, void *data)`
- `void queue_peek_blocking (queue_t *q, void *data)`

4.2.16.2. Function Documentation

4.2.16.2.1. queue_add_blocking

```
void queue_add_blocking (queue_t *q,
      void *data)
```

Blocking add of value to queue.

If the queue is full this function will block, until a removal happens on the queue

Parameters

- `q` Pointer to a queue_t structure, used as a handle
- `data` Pointer to value to be copied into the queue

4.2.16.2.2. queue_free

```
void queue_free (queue_t *q)
```

Destroy the specified queue.

Does not deallocate the `queue_t` structure itself.

Parameters

- `q` Pointer to a queue_t structure, used as a handle

4.2.16.2.3. queue_get_level

```
static uint queue_get_level (queue_t *q)
```

Check of level of the specified queue.

Parameters

- `q` Pointer to a queue_t structure, used as a handle

Returns

- Number of entries in the queue

4.2.16.2.4. queue_get_level_unsafe

```
static uint queue_get_level_unsafe (queue_t *q)
```

Unsafe check of level of the specified queue.

This does not use the spinlock, so may return incorrect results if the spin lock is not externally locked

Parameters

- `q` Pointer to a queue_t structure, used as a handle

Returns

- Number of entries in the queue

4.2.16.2.5. queue_init

```
static void queue_init (queue_t *q,
      uint element_size,
      uint element_count)
```

Initialise a queue, allocating a (possibly shared) spinlock.

Parameters

- `q` Pointer to a queue_t structure, used as a handle
- `element_size` Size of each value in the queue
- `element_count` Maximum number of entries in the queue

4.2.16.2.6. queue_init_with_spinlock

```
void queue_init_with_spinlock (queue_t *q,
       uint element_size,
       uint element_count,
       uint spinlock_num)
```

Initialise a queue with a specific spinlock for concurrency protection.

Parameters

- `q` Pointer to a queue_t structure, used as a handle
- `element_size` Size of each value in the queue
- `element_count` Maximum number of entries in the queue
- `spinlock_num` The spin ID used to protect the queue

4.2.16.2.7. queue_is_empty

```
static bool queue_is_empty (queue_t *q)
```

Check if queue is empty.

This function is interrupt and multicore safe.

Parameters

- `q` Pointer to a queue_t structure, used as a handle

Returns

- true if queue is empty, false otherwise

4.2.16.2.8. queue_is_full

```
static bool queue_is_full (queue_t *q)
```

Check if queue is full.

This function is interrupt and multicore safe.

Parameters

- `q` Pointer to a queue_t structure, used as a handle

Returns

- true if queue is full, false otherwise

4.2.16.2.9. queue_peek_blocking

```
void queue_peek_blocking (queue_t *q,
       void *data)
```

Blocking peek at next value to be removed from queue.

If the queue is empty function will block until a value is added

Parameters

- `q` Pointer to a queue_t structure, used as a handle
- `data` Pointer to the location to receive the peeked value

4.2.16.2.10. queue_remove_blocking

```
void queue_remove_blocking (queue_t *q,
      void *data)
```

Blocking remove entry from queue.

If the queue is empty this function will block until a value is added.

Parameters

- `q` Pointer to a queue_t structure, used as a handle
- `data` Pointer to the location to receive the removed value

4.2.16.2.11. queue_try_add

```
bool queue_try_add (queue_t *q,
      void *data)
```

Non-blocking add value queue if not full.

If the queue is full this function will return immediately with false, otherwise the data is copied into a new value added to the queue, and this function will return true.

Parameters

- `q` Pointer to a queue_t structure, used as a handle
- `data` Pointer to value to be copied into the queue

Returns

- true if the value was added

4.2.16.2.12. queue_try_peek

```
bool queue_try_peek (queue_t *q,
      void *data)
```

Non-blocking peek at the next item to be removed from the queue.

If the queue is not empty this function will return immediately with true with the peeked entry copied into the location specified by the data parameter, otherwise the function will return false.

Parameters

- `q` Pointer to a queue_t structure, used as a handle
- `data` Pointer to the location to receive the peeked value

Returns

- true if there was a value to peek

4.2.16.2.13. queue_try_remove

```
bool queue_try_remove (queue_t *q,
      void *data)
```

Non-blocking removal of entry from the queue if non empty.

If the queue is not empty function will copy the removed value into the location provided and return immediately with true,

otherwise the function will return immediately with false.

Parameters

- `q` Pointer to a queue_t structure, used as a handle
- `data` Pointer to the location to receive the removed value

Returns

- true if a value was removed

4.3. Third-party Libraries

Third party libraries for implementing high level functionality.

tinyusb_device	TinyUSB Device-mode support for the RP2040
tinyusb_host	TinyUSB Host-mode support for the RP2040

4.3.1. tinyusb_device

TinyUSB Device-mode support for the RP2040

4.3.2. tinyusb_host

TinyUSB Host-mode support for the RP2040

4.4. 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_base	Core types and macros for the Pico SDK. This header is intended to be included by all source code.
pico_bit_ops	Optimized bit manipulation functions. Additionally provides replacement implementations of the compiler built-ins *builtin_popcount,* builtin_clz and __bulitin_ctz.
pico_bootrom	Access to functions and data in the RP2040 bootrom.
pico_cxx_options	non-code library controlling C++ related compile options
pico_divider	Optimized 32 and 64 bit division functions accelerated by the RP2040 hardware divider. Additionally provides integration with the C / and % operators.
pico_double	Optimized double-precision floating point functions.
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	Compiler definitions for the selected PICO_PLATFORM.

pico_printf	Compact replacement for printf by Marco Paland (info@paland.com)
pico_runtime	Aggregate runtime support including pico_bit_ops, pico_divider, pico_double, pico_int64_ops, pico_float, pico_malloc, pico_mem_ops and pico_standard_link.
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_usb	Support for stdin/stdout over USB serial (CDC)
pico_standard_link	Standard link step providing the basics for creating a runnable binary.

4.4.1. boot_stage2

Second stage boot loaders responsible for setting up external flash.

4.4.2. pico_base

Core types and macros for the Pico SDK. This header is intended to be included by all source code.

4.4.3. pico_bit_ops

Optimized bit manipulation functions. Additionally provides replacement implementations of the compiler built-ins *builtin_popcount,* builtin_clz and __bulitin_ctz.

4.4.3.1. Function List

- `uint32_t __rev (uint32_t bits)`
- `uint64_t __revll (uint64_t bits)`

4.4.3.2. Function Documentation

4.4.3.2.1. __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

4.4.3.2.2. __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

4.4.4. pico_bootrom

Access to functions and data in the RP2040 bootrom.

4.4.4.1. Function List

- `static uint32_t rom_table_code (char c1, char c2)`
- `void * rom_func_lookup (uint32_t code)`
- `void * rom_data_lookup (uint32_t code)`
- `bool rom_funcs_lookup (uint32_t *table, unsigned int count)`
- `static void reset_usb_boot (uint32_t usb_activity_gpio_pin_mask, uint32_t disable_interface_mask)`

4.4.4.2. Function Documentation

4.4.4.2.1. reset_usb_boot

```
static void 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

4.4.4.2.2. rom_data_lookup

```
void* rom_data_lookup (uint32_t code)
```

Lookup a bootrom address by code.

Parameters

- `code` the code

Returns

- a pointer to the data, or NULL if the code does not match any bootrom function

4.4.4.2.3. rom_func_lookup

```
void* rom_func_lookup (uint32_t code)
```

Lookup a bootrom function by code.

Parameters

- `code` the code

Returns

- a pointer to the function, or NULL if the code does not match any bootrom function

4.4.4.2.4. 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

4.4.4.2.5. rom_table_code

```
static uint32_t rom_table_code (char c1,
       char c2)
```

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()`

4.4.5. pico_cxx_options

non-code library controlling C++ related compile options

4.4.6. pico_divider

Optimized 32 and 64 bit division functions accelerated by the RP2040 hardware divider. Additionally provides integration with the C / and % operators.

4.4.6.1. Function List

- int32_t div_s32s32 (int32_t a, int32_t b)
- static int32_t divmod_s32s32_rem (int32_t a, int32_t b, int32_t *rem)
- divmod_result_t divmod_s32s32 (int32_t a, int32_t b)
- uint32_t div_u32u32 (uint32_t a, uint32_t b)
- static uint32_t divmod_u32u32_rem (uint32_t a, uint32_t b, uint32_t *rem)
- divmod_result_t divmod_u32u32 (uint32_t a, uint32_t b)
- int64_t div_s64s64 (int64_t a, int64_t b)
- int64_t divmod_s64s64_rem (int64_t a, int64_t b, int64_t *rem)
- int64_t divmod_s64s64 (int64_t a, int64_t b)
- uint64_t div_u64u64 (uint64_t a, uint64_t b)
- uint64_t divmod_u64u64_rem (uint64_t a, uint64_t b, uint64_t *rem)
- uint64_t divmod_u64u64 (uint64_t a, uint64_t b)
- int32_t div_s32s32_unsafe (int32_t a, int32_t b)
- int32_t divmod_s32s32_rem_unsafe (int32_t a, int32_t b, int32_t *rem)
- int64_t divmod_s32s32_unsafe (int32_t a, int32_t b)
- uint32_t div_u32u32_unsafe (uint32_t a, uint32_t b)
- uint32_t divmod_u32u32_rem_unsafe (uint32_t a, uint32_t b, uint32_t *rem)
- uint64_t divmod_u32u32_unsafe (uint32_t a, uint32_t b)
- int64_t div_s64s64_unsafe (int64_t a, int64_t b)
- int64_t divmod_s64s64_rem_unsafe (int64_t a, int64_t b, int64_t *rem)
- int64_t divmod_s64s64_unsafe (int64_t a, int64_t b)
- uint64_t div_u64u64_unsafe (uint64_t a, uint64_t b)
- uint64_t divmod_u64u64_rem_unsafe (uint64_t a, uint64_t b, uint64_t *rem)
- uint64_t divmod_u64u64_unsafe (uint64_t a, uint64_t b)

4.4.6.2. Function Documentation

4.4.6.2.1. 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

4.4.6.2.2. div_s32s32_unsafe

```
int32_t div_s32s32_unsafe (int32_t a,
      int32_t b)
```

Unsafe integer divide of two signed 32-bit values.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor

Returns

- quotient

4.4.6.2.3. 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

4.4.6.2.4. div_s64s64_unsafe

```
int64_t div_s64s64_unsafe (int64_t a,
      int64_t b)
```

Unsafe integer divide of two signed 64-bit values.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor

Returns

- Quotient

4.4.6.2.5. 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

4.4.6.2.6. div_u32u32_unsafe

```
uint32_t div_u32u32_unsafe (uint32_t a,
      uint32_t b)
```

Unsafe integer divide of two unsigned 32-bit values.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor

Returns

- Quotient

4.4.6.2.7. 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

4.4.6.2.8. div_u64u64_unsafe

```
uint64_t div_u64u64_unsafe (uint64_t a,
      uint64_t b)
```

Unsafe integer divide of two unsigned 64-bit values.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor

Returns

- Quotient

4.4.6.2.9. 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

4.4.6.2.10. divmod_s32s32_rem

```
static int32_t divmod_s32s32_rem (int32_t a,
       int32_t b,
       int32_t *rem)
```

Integer divide of two signed 32-bit values, with remainder.

Parameters

- `a` Dividend
- `b` Divisor
- `rem` The remainder of dividend/divisor

Returns

- Quotient result of dividend/divisor

4.4.6.2.11. divmod_s32s32_rem_unsafe

```
int32_t divmod_s32s32_rem_unsafe (int32_t a,
       int32_t b,
       int32_t *rem)
```

Unsafe integer divide of two signed 32-bit values, with remainder.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor
- `rem` The remainder of dividend/divisor

Returns

- Quotient result of dividend/divisor

4.4.6.2.12. divmod_s32s32_unsafe

```
int64_t divmod_s32s32_unsafe (int32_t a,
       int32_t b)
```

Unsafe integer divide of two unsigned 32-bit values.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor

Returns

- quotient in low word/r0, remainder in high word/r1

4.4.6.2.13. 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)

4.4.6.2.14. 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

4.4.6.2.15. divmod_s64s64_rem_unsafe

```
int64_t divmod_s64s64_rem_unsafe (int64_t a,
        int64_t b,
        int64_t *rem)
```

Unsafe integer divide of two signed 64-bit values, with remainder.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor
- `rem` The remainder of dividend/divisor

Returns

- Quotient result of dividend/divisor

4.4.6.2.16. divmod_s64s64_unsafe

```
int64_t divmod_s64s64_unsafe (int64_t a,
        int64_t b)
```

Unsafe integer divide of two signed 64-bit values.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor

Returns

- quotient in result (r0,r1), remainder in regs (r2, r3)

4.4.6.2.17. 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

4.4.6.2.18. divmod_u32u32_rem

```
static uint32_t divmod_u32u32_rem (uint32_t a,
      uint32_t b,
      uint32_t *rem)
```

Integer divide of two unsigned 32-bit values, with remainder.

Parameters

- `a` Dividend
- `b` Divisor
- `rem` The remainder of dividend/divisor

Returns

- Quotient result of dividend/divisor

4.4.6.2.19. divmod_u32u32_rem_unsafe

```
uint32_t divmod_u32u32_rem_unsafe (uint32_t a,
      uint32_t b,
      uint32_t *rem)
```

Unsafe integer divide of two unsigned 32-bit values, with remainder.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor
- `rem` The remainder of dividend/divisor

Returns

- Quotient result of dividend/divisor

4.4.6.2.20. divmod_u32u32_unsafe

```
uint64_t divmod_u32u32_unsafe (uint32_t a,
      uint32_t b)
```

Unsafe integer divide of two unsigned 32-bit values.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor

Returns

- quotient in low word/r0, remainder in high word/r1

4.4.6.2.21. 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)

4.4.6.2.22. 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

4.4.6.2.23. divmod_u64u64_rem_unsafe

```
uint64_t divmod_u64u64_rem_unsafe (uint64_t a,
      uint64_t b,
      uint64_t *rem)
```

Unsafe integer divide of two unsigned 64-bit values, with remainder.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor
- `rem` The remainder of dividend/divisor

Returns

- Quotient result of dividend/divisor

4.4.6.2.24. divmod_u64u64_unsafe

```
uint64_t divmod_u64u64_unsafe (uint64_t a,
      uint64_t b)
```

Unsafe integer divide of two signed 64-bit values.

Do not use in interrupts

Parameters

- `a` Dividend
- `b` Divisor

Returns

- quotient in result (r0,r1), remainder in regs (r2, r3)

4.4.7. pico_double

Optimized double-precision floating point functions.

(Replacement) optimized implementations are provided of the following compiler built-ins and math library functions:

- *aeabi_dadd,* aeabi_ddiv, *aeabi_dmul,* aeabi_drsub, *aeabi_dsub,* aeabi_cdcmpeq, *aeabi_cdrcmple,* aeabi_cdcmple, *aeabi_dcmpeq,* aeabi_dcmplt, *aeabi_dcmple,* aeabi_dcmpge, *aeabi_dcmpgt,* aeabi_dcmpun, *aeabi_i2d,* aeabi_l2d, *aeabi_ui2d,* aeabi_ul2d, *aeabi_d2iz,* aeabi_d2lz, *aeabi_d2uiz,* aeabi_d2ulz, __aeabi_d2f
- sqrt, cos, sin, tan, atan2, exp, log, 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
- powint, sincos (GNU extensions)

The following additional optimized functions are also provided:

- fix2double, ufix2double, fix642double, ufix642double, double2fix, double2ufix, double2fix64, double2ufix64, double2int, double2int64, double2int_z, double2int64_z

4.4.8. pico_float

Optimized single-precision floating point functions.

(Replacement) optimized implementations are provided of the following compiler built-ins and math library functions:

- *aeabi_fadd,* aeabi_fdiv, *aeabi_fmul,* aeabi_frsub, *aeabi_fsub,* aeabi_cfcmpeq, *aeabi_cfrcmple,* aeabi_cfcmple, *aeabi_fcmpeq,* aeabi_fcmplt, *aeabi_fcmple,* aeabi_fcmpge, *aeabi_fcmpgt,* aeabi_fcmpun, *aeabi_i2f,* aeabi_l2f, *aeabi_ui2f,* aeabi_ul2f, *aeabi_f2iz,* aeabi_f2lz, *aeabi_f2uiz,* aeabi_f2ulz, __aeabi_f2d, sqrtf, cosf, sinf, tanf, atan2f, expf, logf
- 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
- powintf, sincosf (GNU extensions)

The following additional optimized functions are also provided:

- fix2float, ufix2float, fix642float, ufix642float, float2fix, float2ufix, float2fix64, float2ufix64, float2int, float2int64, float2int_z, float2int64_z

4.4.9. pico_int64_ops

Optimized replacement implementations of the compiler built-in 64 bit multiplication.

This library does not provide any additional functions

4.4.10. pico_malloc

Multi-core safety for malloc, calloc and free.

This library does not provide any additional functions

4.4.11. pico_mem_ops

Provides optimized replacement implementations of the compiler built-in memcpy, memset and related functions:

- memset, memcpy
- *aeabi_memset,* aeabi_memset4, *aeabi_memset8,* aeabi_memcpy, *aeabi_memcpy4,* aeabi_memcpy8

This library does not provide any additional functions

4.4.12. pico_platform

Compiler definitions for the selected PICO_PLATFORM.

4.4.13. pico_printf

Compact replacement for printf by Marco Paland (info@paland.com)

4.4.14. pico_runtime

Aggregate runtime support including pico_bit_ops, pico_divider, pico_double, pico_int64_ops, pico_float, pico_malloc, pico_mem_ops and pico_standard_link.

4.4.15. pico_stdio

Customized stdio support allowing for input and output from UART, USB, semi-hosting etc.

Note the API for adding additional input output devices is not yet considered stable

4.4.15.1. Modules

- `pico_stdio_semihosting`
  Experimental support for stdout using RAM semihosting.
- `pico_stdio_uart`
  Support for stdin/stdout using UART.
- `pico_stdio_usb`
  Support for stdin/stdout over USB serial (CDC)

4.4.15.2. Function List

- `void stdio_init_all ()`
- `void stdio_flush ()`
- `int getchar_timeout_us (uint32_t timeout_us)`
- `void stdio_set_driver_enabled (stdio_driver_t *driver, bool enabled)`
- `void stdio_filter_driver (stdio_driver_t *driver)`
- `void stdio_set_translate_crlf (stdio_driver_t *driver, bool translate)`

4.4.15.3. Function Documentation

4.4.15.3.1. getchar_timeout_us

`int 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

4.4.15.3.2. stdio_filter_driver

`void stdio_filter_driver (stdio_driver_t *driver)`

Control limiting of output to a single 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

4.4.15.3.3. stdio_flush

`void stdio_flush ()`

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 and semihosting based on the presence of the respective librariess in the binary.

See also

- stdio_uart, stdio_usb, stdio_semihosting

4.4.15.3.4. stdio_init_all

`void stdio_init_all ()`

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 and semihosting based on the presence of the respective librariess in the binary.

See also

- stdio_uart, stdio_usb, stdio_semihosting

4.4.15.3.5. 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.

Parameters

- `driver` the driver
- `enabled` true to add, false to remove

4.4.15.3.6. 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

Parameters

- `driver` the driver
- `translate` If true, convert line feeds to carriage return on transmissions

4.4.16. pico_stdio_semihosting

Experimental support for stdout using RAM semihosting.

Linking this library or calling pico_enable_stdio_semihosting(TARGET) in the CMake (which achieves the same thing) will add semihosting to the drivers used for standard output

4.4.16.1. Function List

- `void stdio_semihosting_init ()`

4.4.16.2. Function Documentation

4.4.16.2.1. stdio_semihosting_init

```
void stdio_semihosting_init ()
```

Explicitly initialize stdout over semihosting and add it to the current set of stdout targets.

4.4.17. pico_stdio_uart

Support for stdin/stdout using UART.

Linking this library or calling pico_enable_stdio_uart(TARGET) in the CMake (which achieves the same thing) will add UART to the drivers used for standard output

4.4.17.1. Function List

- `void stdio_uart_init ()`

- `void stdout_uart_init ()`
- `void stdin_uart_init ()`
- `void stdio_uart_init_full (uart_inst_t *uart, uint baud_rate, int tx_pin, int rx_pin)`
- `bool stdio_usb_init ()`

4.4.17.2. Function Documentation

4.4.17.2.1. stdin_uart_init

```
void stdin_uart_init ()
```

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

4.4.17.2.2. stdio_uart_init

```
void stdio_uart_init ()
```

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.

4.4.17.2.3. 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)

4.4.17.2.4. stdio_usb_init

```
bool stdio_usb_init ()
```

Explicitly initialize USB stdio and add it to the current set of stdin drivers.

4.4.17.2.5. stdout_uart_init

```
void stdout_uart_init ()
```

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

4.4.18. pico_stdio_usb

Support for stdin/stdout over USB serial (CDC)

Linking this library or calling pico_enable_stdio_usb(TARGET) in the CMake (which achieves the same thing) will add USB CDC to the drivers used for standard 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.

4.4.19. pico_standard_link

Standard link step providing the basics for creating a runnable binary.

This includes

- C runtime initialization
- Linker scripts for 'default', 'no_flash', 'blocked_ram' and 'copy_to_ram' binaries
- 'Binary Information' support
- Linker option control

4.5. External API Headers

Headers for interfaces that are shared with code outside of the SDK.

boot_picoboot	Header file for the PICOBOOT USB interface exposed by an RP2040 in BOOTSEL mode.
boot_uf2	Header file for the UF2 format supported by an RP2040 in BOOTSEL mode.

4.5.1. boot_picoboot

Header file for the PICOBOOT USB interface exposed by an RP2040 in BOOTSEL mode.

4.5.2. boot_uf2

Header file for the UF2 format supported by an RP2040 in BOOTSEL mode.

Appendix A: App Notes

Attaching a 7 segment LED via GPIO

This example code shows how to interface the Raspberry Pi Pico to a generic 7 segment LED device. It uses the LED to count from 0 to 9 and then repeat. If the button is pressed, then the numbers will count down instead of up.

Wiring information

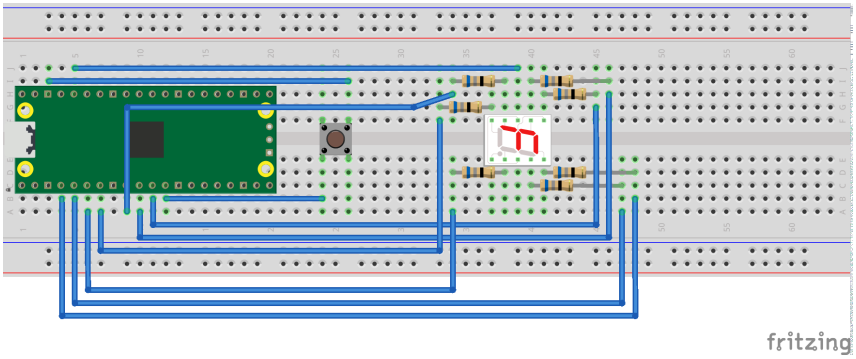
Our 7 Segment display has pins as follows.

```
--A--
F   B
--G--
E   C
--D--
```

By default we are allocating GPIO 2 to A, 3 to B etc. So, connect GPIO 2 to pin A on the 7 segment LED display and so on. You will need the appropriate resistors (68 ohm should be fine) for each segment. The LED device used here is common anode, so the anode pin is connected to the 3.3v supply, and the GPIO's need to pull low (to ground) to complete the circuit. The pull direction of the GPIO's is specified in the code itself.

Connect the switch to connect on pressing. One side should be connected to ground, the other to GPIO 9.

Figure 8. Wiring Diagram for 7 segment LED.



List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/master/gpio/hello_7segment/CMakeLists.txt Lines 1 - 12

```
1 add_executable(hello_7segment
2         hello_7segment.c
3         )
4
5 # Pull in our pico_stdlib which pulls in commonly used features
6 target_link_libraries(hello_7segment pico_stdlib)
7
8 # 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/tree/master/gpio/hello_7segment/hello_7segment.c Lines 1 - 95

```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 A, 3 to B etc.
  So, connect GOIP 2 to pin A on the 7 segment LED display etc. Don't forget
  the appropriate resistors, best to use one for each segment!

  Connect button so that pressing the switch connects the GPIO 9 (default) to
  ground (pull down)
*/

#define FIRST_GPIO 2
#define BUTTON_GPIO (FIRST_GPIO+7)

// This array converts a number 0-9 to a bit pattern to send to the GPIO's
int bits[10] = {
        0x3f,  // 0
        0x06,  // 1
        0x5b,  // 2
        0x4f,  // 3
        0x66,  // 4
        0x6d,  // 5
        0x7d,  // 6
        0x07,  // 7
        0x7f,  // 8
        0x67   // 9
};

// tag::hello_gpio[]
int main() {
    stdio_init_all();
    printf("Hello, 7segment - press button to count down!\n");

    // We could use gpio_set_dir_out_masked() here
    for (int gpio = FIRST_GPIO; gpio < FIRST_GPIO + 7; gpio++) {
```

```
52          gpio_init(gpio);
53          gpio_set_dir(gpio, GPIO_OUT);
54          // Our bitmap above has a bit set where we need an LED on, BUT, we are pulling low to
    light
55          // so invert our output
56          gpio_set_outover(gpio, GPIO_OVERRIDE_INVERT);
57      }
58
59      gpio_init(BUTTON_GPIO);
60      gpio_set_dir(BUTTON_GPIO, GPIO_IN);
61      // We are using the button to pull down to 0v when pressed, so ensure that when
62      // unpressed, it uses internal pull ups. Otherwise when unpressed, the input will
63      // be floating.
64      gpio_pull_up(BUTTON_GPIO);
65
66      int val = 0;
67      while (true) {
68          // Count upwards or downwards depending on button input
69          // We are pulling down on switch active, so invert the get to make
70          // a press count downwards
71          if (!gpio_get(BUTTON_GPIO)) {
72              if (val == 9) {
73                  val = 0;
74              } else {
75                  val++;
76              }
77          } else if (val == 0) {
78              val = 9;
79          } else {
80              val--;
81          }
82
83          // We are starting with GPIO 2, our bitmap starts at bit 0 so shift to start at 2.
84          int32_t mask = bits[val] << FIRST_GPIO;
85
86          // Set all our GPIO's in one go!
87          // If something else is using GPIO, we might want to use gpio_put_masked()
88          gpio_set_mask(mask);
89          sleep_ms(250);
90          gpio_clr_mask(mask);
91      }
92
93      return 0;
94  }
95  // end::hello_gpio[]
```

Bill of Materials

Table 11. A list of materials required for the example

Item	**Quantity**	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	http://raspberrypi.org/
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 9 for wiring instructions.

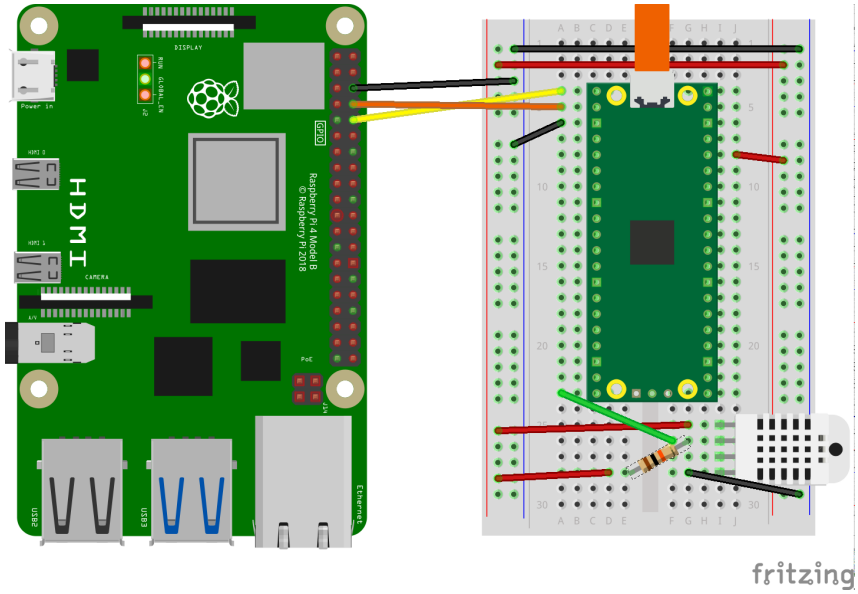


Figure 9. Wiring the DHT-22 temperature sensor to Raspberry Pi Pico, and connecting Pico's UART0 to the Raspberry Pi 4.

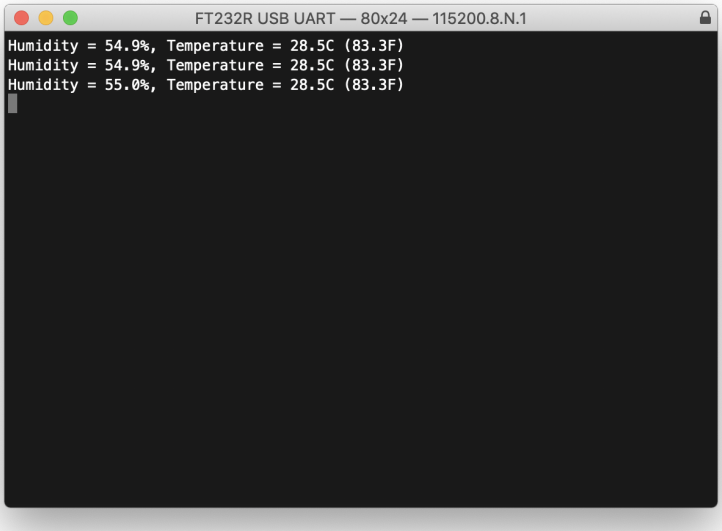
NOTE

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 9 and you should see something similar to Figure 10 in `minicom` when connected to `/dev/serial0` on the Raspberry Pi.

Figure 10. 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/tree/master/gpio/dht_sensor/CMakeLists.txt Lines 1 - 11

```
1 add_executable(dht
2         dht.c
3         )
4
5 target_link_libraries(dht pico_stdlib)
6
7 pico_add_extra_outputs(dht)
8
9 # add url via pico_set_program_url
10 example_auto_set_url(dht)
```

dht.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/master/gpio/dht_sensor/dht.c Lines 1 - 83

```
1 /**
2  * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
3  *
```

```
 * SPDX-License-Identifier: BSD-3-Clause
 **/

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

const uint LED_PIN = PICO_DEFAULT_LED_PIN;
const uint DHT_PIN = 15;
const uint MAX_TIMINGS = 85;

typedef struct {
    float humidity;
    float temp_celsius;
} dht_reading;

void read_from_dht(dht_reading *result);

int main() {
    stdio_init_all();
    gpio_init(LED_PIN);
    gpio_init(DHT_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);
    while (1) {
        dht_reading reading;
        read_from_dht(&reading);
        float fahrenheit = (reading.temp_celsius * 9 / 5) + 32;
        printf("Humidity = %.1f%%, Temperature = %.1fC (%.1fF)\n",
               reading.humidity, reading.temp_celsius, fahrenheit);

        sleep_ms(2000);
    }
}

void read_from_dht(dht_reading *result) {
    int data[5] = {0, 0, 0, 0, 0};
    uint last = 1;
    uint j = 0;

    gpio_set_dir(DHT_PIN, GPIO_OUT);
    gpio_put(DHT_PIN, 0);
    sleep_ms(20);
    gpio_set_dir(DHT_PIN, GPIO_IN);

    gpio_put(LED_PIN, 1);
    for (uint i = 0; i < MAX_TIMINGS; i++) {
        uint count = 0;
        while (gpio_get(DHT_PIN) == last) {
            count++;
            sleep_us(1);
            if (count == 255) break;
        }
        last = gpio_get(DHT_PIN);
        if (count == 255) break;

        if ((i >= 4) && (i % 2 == 0)) {
            data[j / 8] <<= 1;
            if (count > 16) data[j / 8] |= 1;
            j++;
        }
    }
    gpio_put(LED_PIN, 0);
```

```
67
68      if ((j >= 40) && (data[4] == ((data[0] + data[1] + data[2] + data[3]) & 0xFF))) {
69          result->humidity = (float) ((data[0] << 8) + data[1]) / 10;
70          if (result->humidity > 100) {
71              result->humidity = data[0];
72          }
73          result->temp_celsius = (float) (((data[2] & 0x7F) << 8) + data[3]) / 10;
74          if (result->temp_celsius > 125) {
75              result->temp_celsius = data[2];
76          }
77          if (data[2] & 0x80) {
78              result->temp_celsius = -result->temp_celsius;
79          }
80      } else {
81          printf("Bad data\n");
82      }
83  }
```

Bill of Materials

Table 12. A list of materials required for the example

Item	**Quantity**	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	http://raspberrypi.org/
10 kΩ resistor	1	generic part
M/M Jumper wires	4	generic part
DHT-22 sensor	1	generic part

Attaching a BME280 temperature/humidity/pressure sensor via SPI

This example code shows how to interface the Raspberry Pi Pico to a BME280 temperature/humidity/pressure. The particular device used can be interfaced via I2C or SPI, we are using SPI, and interfacing at 3.3v.

This examples reads the data from the sensor, and runs it through the 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.

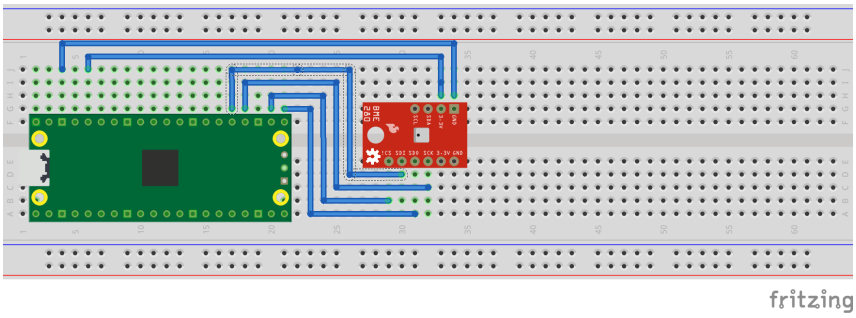


Figure 11. Wiring Diagram for bme280.

List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/master/spi/bme280_spi/CMakeLists.txt Lines 1 - 12

```
1 add_executable(bme280_spi
2         bme280_spi.c
3         )
4
5 # Pull in our (to be renamed) simple get you started dependencies
6 target_link_libraries(bme280_spi pico_stdlib hardware_spi)
7
8 # create map/bin/hex file etc.
9 pico_add_extra_outputs(bme280_spi)
10
11 # add url via pico_set_program_url
12 example_auto_set_url(bme280_spi)
```

bme280_spi.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/master/spi/bme280_spi/bme280_spi.c Lines 1 - 233

```
1 /**
2  * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
3  *
4  * SPDX-License-Identifier: BSD-3-Clause
5  */
6
7 #include <stdio.h>
8 #include <string.h>
9 #include "pico/stdlib.h"
10 #include "hardware/spi.h"
11
12 /* Example code to talk to a bme280 humidity/temperature/pressure sensor.
13
14    NOTE: Ensure the device is capable of being driven at 3.3v NOT 5v. The Pico
15    GPIO (and therefor SPI) cannot be used at 5v.
```

```
16
17    You will need to use a level shifter on the SPI lines if you want to run the
18    board at 5v.
19
20    Connections on Raspberry Pi Pico board and a generic bme280 board, other
21    boards may vary.
22
23    GPIO 16 (pin 21) MISO/spi0_rx-> SDO/SDO on bme280 board
24    GPIO 17 (pin 22) Chip select -> CSB/!CS on bme280 board
25    GPIO 18 (pin 24) SCK/spi0_sclk -> SCL/SCK on bme280 board
26    GPIO 19 (pin 25) MOSI/spi0_tx -> SDA/SDI on bme280 board
27    3.3v (pin 3;6) -> VCC on bme280 board
28    GND (pin 38)  -> GND on bme280 board
29
30    Note: SPI devices can have a number of different naming schemes for pins. See
31    the Wikipedia page at https://en.wikipedia.org/wiki/Serial_Peripheral_Interface
32    for variations.
33
34    This code uses a bunch of register definitions, and some compensation code derived
35    from the Bosch datasheet which can be found here.
36    https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme280-
   ds002.pdf
37 */
38
39 #define PIN_MISO 16
40 #define PIN_CS   17
41 #define PIN_SCK  18
42 #define PIN_MOSI 19
43
44 #define SPI_PORT spi0
45 #define READ_BIT 0x80
46
47 int32_t t_fine;
48
49 uint16_t dig_T1;
50 int16_t dig_T2, dig_T3;
51 uint16_t dig_P1;
52 int16_t dig_P2, dig_P3, dig_P4, dig_P5, dig_P6, dig_P7, dig_P8, dig_P9;
53 uint8_t dig_H1, dig_H3;
54 int8_t dig_H6;
55 int16_t dig_H2, dig_H4, dig_H5;
56
57 /* The following compensation functions are required to convert from the raw ADC
58 data from the chip to something usable. Each chip has a different set of
59 compensation parameters stored on the chip at point of manufacture, which are
60 read from the chip at startup and used inthese routines.
61 */
62 int32_t compensate_temp(int32_t adc_T) {
63     int32_t var1, var2, T;
64     var1 = ((((adc_T >> 3) - ((int32_t) dig_T1 << 1))) * ((int32_t) dig_T2)) >> 11;
65     var2 = (((((adc_T >> 4) - ((int32_t) dig_T1)) * ((adc_T >> 4) - ((int32_t) dig_T1))) >>
   12) * ((int32_t) dig_T3))
66             >> 14;
67
68     t_fine = var1 + var2;
69     T = (t_fine * 5 + 128) >> 8;
70     return T;
71 }
72
73 uint32_t compensate_pressure(int32_t adc_P) {
74     int32_t var1, var2;
75     uint32_t p;
76     var1 = (((int32_t) t_fine) >> 1) - (int32_t) 64000;
```

```
77      var2 = (((var1 >> 2) * (var1 >> 2)) >> 11) * ((int32_t) dig_P6);
78      var2 = var2 + ((var1 * ((int32_t) dig_P5)) << 1);
79      var2 = (var2 >> 2) + (((int32_t) dig_P4) << 16);
80      var1 = (((dig_P3 * (((var1 >> 2) * (var1 >> 2)) >> 13)) >> 3) + ((((int32_t) dig_P2) *
    var1) >> 1)) >> 18;
81      var1 = ((((32768 + var1)) * ((int32_t) dig_P1)) >> 15);
82      if (var1 == 0)
83          return 0;
84
85      p = (((uint32_t) (((int32_t) 1048576) - adc_P) - (var2 >> 12))) * 3125;
86      if (p < 0x80000000)
87          p = (p << 1) / ((uint32_t) var1);
88      else
89          p = (p / (uint32_t) var1) * 2;
90
91      var1 = (((int32_t) dig_P9) * ((int32_t) (((p >> 3) * (p >> 3)) >> 13))) >> 12;
92      var2 = (((int32_t) (p >> 2)) * ((int32_t) dig_P8)) >> 13;
93      p = (uint32_t) ((int32_t) p + ((var1 + var2 + dig_P7) >> 4));
94
95      return p;
96  }
97
98  uint32_t compensate_humidity(int32_t adc_H) {
99      int32_t v_x1_u32r;
100     v_x1_u32r = (t_fine - ((int32_t) 76800));
101     v_x1_u32r = (((((adc_H << 14) - (((int32_t) dig_H4) << 20) - (((int32_t) dig_H5) *
    v_x1_u32r)) +
102                     ((int32_t) 16384)) >> 15) * (((((((v_x1_u32r * ((int32_t) dig_H6)) >>
    10) * (((v_x1_u32r *
103
    ((int32_t) dig_H3))
104             >> 11) + ((int32_t) 32768))) >> 10) + ((int32_t) 2097152)) *
105                                                 ((int32_t) dig_H2) + 8192) >> 14));
106     v_x1_u32r = (v_x1_u32r - (((((v_x1_u32r >> 15) * (v_x1_u32r >> 15)) >> 7) * ((int32_t)
    dig_H1)) >> 4));
107     v_x1_u32r = (v_x1_u32r < 0 ? 0 : v_x1_u32r);
108     v_x1_u32r = (v_x1_u32r > 419430400 ? 419430400 : v_x1_u32r);
109
110     return (uint32_t) (v_x1_u32r >> 12);
111 }
112
113 static inline void cs_select() {
114     asm volatile("nop \n nop \n nop");
115     gpio_put(PIN_CS, 0);  // Active low
116     asm volatile("nop \n nop \n nop");
117 }
118
119 static inline void cs_deselect() {
120     asm volatile("nop \n nop \n nop");
121     gpio_put(PIN_CS, 1);
122     asm volatile("nop \n nop \n nop");
123 }
124
125 static void write_register(uint8_t reg, uint8_t data) {
126     uint8_t buf[2];
127     buf[0] = reg & 0x7f;  // remove read bit as this is a write
128     buf[1] = data;
129     cs_select();
130     spi_write_blocking(SPI_PORT, buf, 2);
131     cs_deselect();
132     sleep_ms(10);
133 }
134
```

```
135 static void read_registers(uint8_t reg, uint8_t *buf, uint16_t len) {
136     // For this particular device, we send the device the register we want to read
137     // first, then subsequently read from the device. The register is auto incrementing
138     // so we don't need to keep sending the register we want, just the first.
139     reg |= READ_BIT;
140     cs_select();
141     spi_write_blocking(SPI_PORT, &reg, 1);
142     sleep_ms(10);
143     spi_read_blocking(SPI_PORT, 0, buf, len);
144     cs_deselect();
145     sleep_ms(10);
146 }
147
148 /* This function reads the manufacturing assigned compensation parameters from the device */
149 void read_compensation_parameters() {
150     uint8_t buffer[26];
151
152     read_registers(0x88, buffer, 24);
153
154     dig_T1 = buffer[0] | (buffer[1] << 8);
155     dig_T2 = buffer[2] | (buffer[3] << 8);
156     dig_T3 = buffer[4] | (buffer[5] << 8);
157
158     dig_P1 = buffer[6] | (buffer[7] << 8);
159     dig_P2 = buffer[8] | (buffer[9] << 8);
160     dig_P3 = buffer[10] | (buffer[11] << 8);
161     dig_P4 = buffer[12] | (buffer[13] << 8);
162     dig_P5 = buffer[14] | (buffer[15] << 8);
163     dig_P6 = buffer[16] | (buffer[17] << 8);
164     dig_P7 = buffer[18] | (buffer[19] << 8);
165     dig_P8 = buffer[20] | (buffer[21] << 8);
166     dig_P9 = buffer[22] | (buffer[23] << 8);
167
168     dig_H1 = buffer[25];
169
170     read_registers(0xE1, buffer, 8);
171
172     dig_H2 = buffer[0] | (buffer[1] << 8);
173     dig_H3 = (int8_t) buffer[2];
174     dig_H4 = buffer[3] << 4 | (buffer[4] & 0xf);
175     dig_H5 = (buffer[5] >> 4) | (buffer[6] << 4);
176     dig_H6 = (int8_t) buffer[7];
177 }
178
179 static void bme280_read_raw(int32_t *humidity, int32_t *pressure, int32_t *temperature) {
180     uint8_t buffer[8];
181
182     read_registers(0xF7, buffer, 8);
183     *pressure = ((uint32_t) buffer[0] << 12) | ((uint32_t) buffer[1] << 4) | (buffer[2] >>
    4);
184     *temperature = ((uint32_t) buffer[3] << 12) | ((uint32_t) buffer[4] << 4) | (buffer[5]
    >> 4);
185     *humidity = (uint32_t) buffer[6] << 8 | buffer[7];
186 }
187
188 int main() {
189     stdio_init_all();
190
191     printf("Hello, bme280! Reading raw data from registers via SPI...\n");
192
193     // This example will use SPI0 at 0.5MHz.
194     spi_init(SPI_PORT, 500 * 1000);
195     gpio_set_function(PIN_MISO, GPIO_FUNC_SPI);
```

```
196     gpio_set_function(PIN_SCK, GPIO_FUNC_SPI);
197     gpio_set_function(PIN_MOSI, GPIO_FUNC_SPI);
198
199     // Chip select is active-low, so we'll initialise it to a driven-high state
200     gpio_init(PIN_CS);
201     gpio_set_dir(PIN_CS, GPIO_OUT);
202     gpio_put(PIN_CS, 1);
203
204     // See if SPI is working - interrograte the device for its I2C ID number, should be 0x60
205     uint8_t id;
206     read_registers(0xD0, &id, 1);
207     printf("Chip ID is 0x%x\n", id);
208
209     read_compensation_parameters();
210
211     write_register(0xF2, 0x1); // Humidity oversampling register - going for x1
212     write_register(0xF4, 0x27);// Set rest of oversampling modes and run mode to normal
213
214     int32_t humidity, pressure, temperature;
215
216     while (1) {
217         bme280_read_raw(&humidity, &pressure, &temperature);
218
219         // These are the raw numbers from the chip, so we need to run through the
220         // compensations to get human understandable numbers
221         pressure = compensate_pressure(pressure);
222         temperature = compensate_temp(temperature);
223         humidity = compensate_humidity(humidity);
224
225         printf("Humidity = %.2f%%\n", humidity / 1024.0);
226         printf("Pressure = %dPa\n", pressure);
227         printf("Temp. = %.2fC\n", temperature / 100.0);
228
229         sleep_ms(1000);
230     }
231
232     return 0;
233 }
```

Bill of Materials

Table 13. A list of materials required for the example

Item	Quantity	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	http://raspberrypi.org/
BME280 board	1	generic part
M/M Jumper wires	6	generic part

Attaching a MPU9250 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:

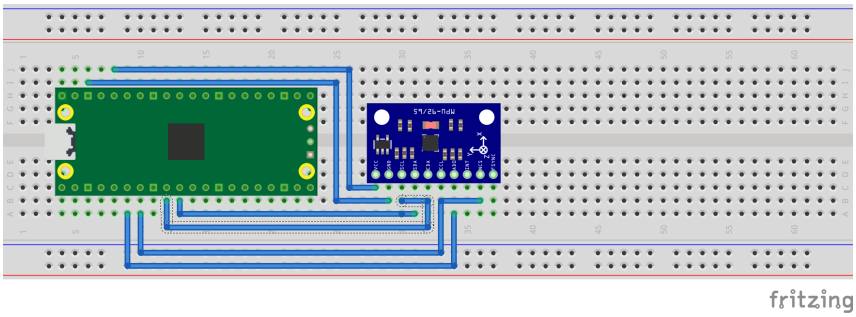
- GPIO 4 (pin 6) MISO/spi0_rx→ ADO on MPU9250 board
- GPIO 5 (pin 7) Chip select → NCS on MPU9250 board
- GPIO 6 (pin 9) SCK/spi0_sclk → SCL on MPU9250 board
- GPIO 7 (pin 10) MOSI/spi0_tx → SDA on MPU9250 board
- 3.3v (pin 36) → VCC on MPU9250 board
- GND (pin 38) → GND on MPU9250 board

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 12. Wiring Diagram for MPU9250.



List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/master/spi/mpu9250_spi/CMakeLists.txt Lines 1 - 12

```
1 add_executable(mpu9250_spi
2         mpu9250_spi.c
3         )
4
5 # Pull in our (to be renamed) simple get you started dependencies
6 target_link_libraries(mpu9250_spi pico_stdlib hardware_spi)
7
8 # create map/bin/hex file etc.
9 pico_add_extra_outputs(mpu9250_spi)
10
```

```
# 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/tree/master/spi/mpu9250_spi/mpu9250_spi.c Lines 1 - 150

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

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

/* Example code to talk to a MPU9250 MEMS accelerometer and gyroscope.
   Ignores the magnetometer, that is left as a exercise for the reader.

   This is taking to simple approach of simply reading registers. It's perfectly
   possible to link up an interrupt line and set things up to read from the
   inbuilt FIFO to make it more useful.

   NOTE: Ensure the device is capable of being driven at 3.3v NOT 5v. The Pico
   GPIO (and therefor SPI) cannot be used at 5v.

   You will need to use a level shifter on the I2C lines if you want to run the
   board at 5v.

   Connections on Raspberry Pi Pico board and a generic MPU9250 board, other
   boards may vary.

   GPIO 4 (pin 6) MISO/spi0_rx-> ADO on MPU9250 board
   GPIO 5 (pin 7) Chip select -> NCS on MPU9250 board
   GPIO 6 (pin 9) SCK/spi0_sclk -> SCL on MPU9250 board
   GPIO 7 (pin 10) MOSI/spi0_tx -> SDA on MPU9250 board
   3.3v (pin 36) -> VCC on MPU9250 board
   GND (pin 38)  -> GND on MPU9250 board

   Note: SPI devices can have a number of different naming schemes for pins. See
   the Wikipedia page at https://en.wikipedia.org/wiki/Serial_Peripheral_Interface
   for variations.
   The particular device used here uses the same pins for I2C and SPI, hence the
   using of I2C names
*/

#define PIN_MISO 4
#define PIN_CS   5
#define PIN_SCK  6
#define PIN_MOSI 7

#define SPI_PORT spi0
#define READ_BIT 0x80

static inline void cs_select() {
    asm volatile("nop \n nop \n nop");
    gpio_put(PIN_CS, 0);  // Active low
    asm volatile("nop \n nop \n nop");
```

```
54 }
55
56 static inline void cs_deselect() {
57     asm volatile("nop \n nop \n nop");
58     gpio_put(PIN_CS, 1);
59     asm volatile("nop \n nop \n nop");
60 }
61
62 static void mpu9250_reset() {
63     // Two byte reset. First byte register, second byte data
64     // There are a load more options to set up the device in different ways that could be
   added here
65     uint8_t buf[] = {0x6B, 0x00};
66     cs_select();
67     spi_write_blocking(SPI_PORT, buf, 2);
68     cs_deselect();
69 }
70
71
72 static void read_registers(uint8_t reg, uint8_t *buf, uint16_t len) {
73     // For this particular device, we send the device the register we want to read
74     // first, then subsequently read from the device. The register is auto incrementing
75     // so we don't need to keep sending the register we want, just the first.
76
77     reg |= READ_BIT;
78     cs_select();
79     spi_write_blocking(SPI_PORT, &reg, 1);
80     sleep_ms(10);
81     spi_read_blocking(SPI_PORT, 0, buf, len);
82     cs_deselect();
83     sleep_ms(10);
84 }
85
86
87 static void mpu9250_read_raw(int16_t accel[3], int16_t gyro[3], int16_t *temp) {
88     uint8_t buffer[6];
89
90     // Start reading acceleration registers from register 0x3B for 6 bytes
91     read_registers(0x3B, buffer, 6);
92
93     for (int i = 0; i < 3; i++) {
94         accel[i] = (buffer[i * 2] << 8 | buffer[(i * 2) + 1]);
95     }
96
97     // Now gyro data from reg 0x43 for 6 bytes
98     read_registers(0x43, buffer, 6);
99
100    for (int i = 0; i < 3; i++) {
101        gyro[i] = (buffer[i * 2] << 8 | buffer[(i * 2) + 1]);;
102    }
103
104    // Now temperature from reg 0x41 for 2 bytes
105    read_registers(0x41, buffer, 2);
106
107    *temp = buffer[0] << 8 | buffer[1];
108 }
109
110 int main() {
111     stdio_init_all();
112
113     printf("Hello, MPU9250! Reading raw data from registers via SPI...\n");
114
115     // This example will use SPI0 at 0.5MHz.
```

```
116     spi_init(SPI_PORT, 500 * 1000);
117     gpio_set_function(PIN_MISO, GPIO_FUNC_SPI);
118     gpio_set_function(PIN_SCK, GPIO_FUNC_SPI);
119     gpio_set_function(PIN_MOSI, GPIO_FUNC_SPI);
120
121     // Chip select is active-low, so we'll initialise it to a driven-high state
122     gpio_init(PIN_CS);
123     gpio_set_dir(PIN_CS, GPIO_OUT);
124     gpio_put(PIN_CS, 1);
125
126     mpu9250_reset();
127
128     // See if SPI is working - interrograte the device for its I2C ID number, should be 0x71
129     uint8_t id;
130     read_registers(0x75, &id, 1);
131     printf("I2C address is 0x%x\n", id);
132
133     int16_t acceleration[3], gyro[3], temp;
134
135     while (1) {
136         mpu9250_read_raw(acceleration, gyro, &temp);
137
138         // These are the raw numbers from the chip, so will need tweaking to be really
    useful.
139         // See the datasheet for more information
140         printf("Acc. X = %d, Y = %d, Z = %d\n", acceleration[0], acceleration[1],
    acceleration[2]);
141         printf("Gyro. X = %d, Y = %d, Z = %d\n", gyro[0], gyro[1], gyro[2]);
142         // Temperature is simple so use the datasheet calculation to get deg C.
143         // Note this is chip temperature.
144         printf("Temp. = %f\n", (temp / 340.0) + 36.53);
145
146         sleep_ms(100);
147     }
148
149     return 0;
150 }
```

Bill of Materials

Table 14. A list of materials required for the example

Item	Quantity	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	http://raspberrypi.org/
MPU9250 board	1	generic part
M/M Jumper wires	6	generic part

Attaching a MPU6050 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.

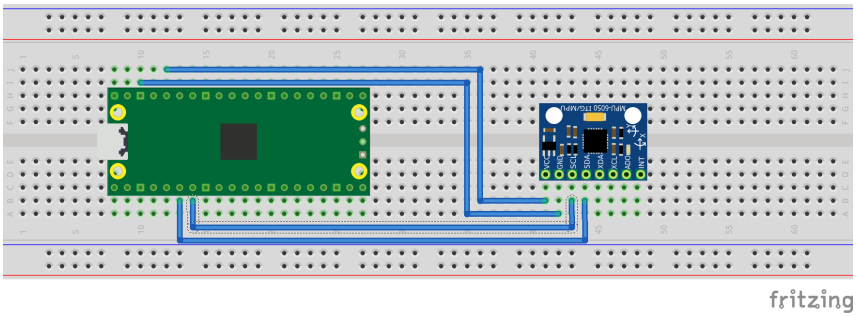
Wiring information

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 13. Wiring Diagram for MPU6050.



List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/master/i2c/mpu6050_i2c/CMakeLists.txt Lines 1 - 12

```
 1 add_executable(mpu6050_i2c
 2         mpu6050_i2c.c
 3         )
 4
 5 # Pull in our (to be renamed) simple get you started dependencies
 6 target_link_libraries(mpu6050_i2c pico_stdlib hardware_i2c)
 7
 8 # create map/bin/hex file etc.
 9 pico_add_extra_outputs(mpu6050_i2c)
10
11 # add url via pico_set_program_url
12 example_auto_set_url(mpu6050_i2c)
```

mpu6050_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/master/i2c/mpu6050_i2c/mpu6050_i2c.c Lines 1 - 110

```
 1 /**
```

```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"

/* 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 therefor I2C) cannot be used at 5v.

   You will need to use a level shifter on the I2C lines if you want to run the
   board at 5v.

   Connections on Raspberry Pi Pico board, other boards may vary.

   GPIO 4 (pin 6)-> SDA on MPU6050 board
   GPIO 5 (pin 7)-> SCL on MPU6050 board
   3.3v (pin 36) -> VCC on MPU6050 board
   GND (pin 38)  -> GND on MPU6050 board
*/

// By default these devices  are on bus address 0x68
static int addr = 0x68;

#define I2C_PORT i2c0

static void mpu6050_reset() {
    // Two byte reset. First byte register, second byte data
    // There are a load more options to set up the device in different ways that could be
added here
    uint8_t buf[] = {0x6B, 0x00};
    i2c_write_blocking(I2C_PORT, addr, buf, 2, false);
}

static void mpu6050_read_raw(int16_t accel[3], int16_t gyro[3], int16_t *temp) {
    // For this particular device, we send the device the register we want to read
    // first, then subsequently read from the device. The register is auto incrementing
    // so we 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_PORT, addr, &val, 1, true); // true to keep master control of bus
    i2c_read_blocking(I2C_PORT, addr, buffer, 6, false);

    for (int i = 0; i < 3; i++) {
        accel[i] = (buffer[i * 2] << 8 | buffer[(i * 2) + 1]);
    }

    // Now gyro data from reg 0x43 for 6 bytes
    // The register is auto incrementing on each read
    val = 0x43;
    i2c_write_blocking(I2C_PORT, addr, &val, 1, true);
```

```
64      i2c_read_blocking(I2C_PORT, addr, buffer, 6, false);  // False - finished with bus
65
66      for (int i = 0; i < 3; i++) {
67          gyro[i] = (buffer[i * 2] << 8 | buffer[(i * 2) + 1]);;
68      }
69
70      // Now temperature from reg 0x41 for 2 bytes
71      // The register is auto incrementing on each read
72      val = 0x41;
73      i2c_write_blocking(I2C_PORT, addr, &val, 1, true);
74      i2c_read_blocking(I2C_PORT, addr, buffer, 2, false);  // False - finished with bus
75
76      *temp = buffer[0] << 8 | buffer[1];
77  }
78
79  int main() {
80      stdio_init_all();
81
82      printf("Hello, MPU6050! Reading raw data from registers...\n");
83
84      // This example will use I2C0 on GPIO4 (SDA) and GPIO5 (SCL) running at 400kHz.
85      i2c_init(I2C_PORT, 400 * 1000);
86      gpio_set_function(4, GPIO_FUNC_I2C);
87      gpio_set_function(5, GPIO_FUNC_I2C);
88      gpio_pull_up(4);
89      gpio_pull_up(5);
90
91      mpu6050_reset();
92
93      int16_t acceleration[3], gyro[3], temp;
94
95      while (1) {
96          mpu6050_read_raw(acceleration, gyro, &temp);
97
98          // These are the raw numbers from the chip, so will need tweaking to be really
    useful.
99          // See the datasheet for more information
100         printf("Acc. X = %d, Y = %d, Z = %d\n", acceleration[0], acceleration[1],
    acceleration[2]);
101         printf("Gyro. X = %d, Y = %d, Z = %d\n", gyro[0], gyro[1], gyro[2]);
102         // Temperature is simple so use the datasheet calculation to get deg C.
103         // Note this is chip temperature.
104         printf("Temp. = %f\n", (temp / 340.0) + 36.53);
105
106         sleep_ms(100);
107     }
108
109     return 0;
110 }
```

Bill of Materials

Table 15. A list of materials required for the example

Item	Quantity	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	http://raspberrypi.org/
MPU6050 board	1	generic part
M/M Jumper wires	4	generic part

Attaching a 16x2 LCD via I2C

This example code shows how to interface the Raspberry Pi Pico to one of the very common 16x2 LCD character displays. The display will need a 3.3V I2C adapter board as this example uses I2C for communications.

NOTE

These LCD displays can also be driven directly using GPIO without the use of an adapter board. That is beyond the scope of this example.

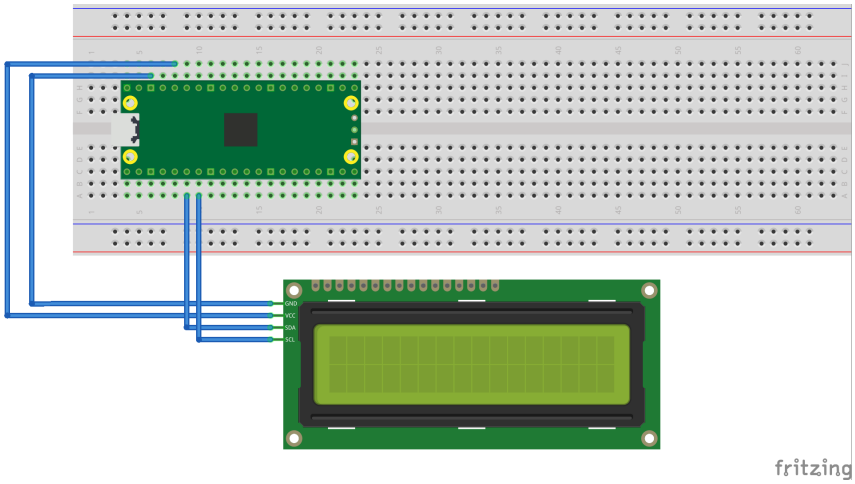
Wiring information

Wiring up the device requires 4 jumpers, to connect VCC (3.3v), GND, SDA and SCL. The example here uses I2C port 0, which is assigned to GPIO 4 (SDA) and 5 (SCL) in software. Power is supplied from the 3.3V pin.

WARNING

Many displays of this type are 5v. If you wish to use a 5v display you will need to use level shifters on the SDA and SCL lines to convert from the 3.3V used by the RP2040. Whilst a 5v display will just about work at 3.3v, the display will be dim.

Figure 14. Wiring Diagram for LCD1602A LCD with I2C bridge.



List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/master/i2c/lcd_1602_i2c/CMakeLists.txt Lines 1 - 12

```
 1 add_executable(lcd_1602_i2c
 2         lcd_1602_i2c.c
 3         )
 4
 5 # Pull in our (to be renamed) simple get you started dependencies
 6 target_link_libraries(lcd_1602_i2c pico_stdlib hardware_i2c)
 7
 8 # create map/bin/hex file etc.
 9 pico_add_extra_outputs(lcd_1602_i2c)
10
11 # add url via pico_set_program_url
```

```
12 example_auto_set_url(lcd_1602_i2c)
```

lcd_1602_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/master/i2c/lcd_1602_i2c/lcd_1602_i2c.c Lines 1 - 164

```
 1 /**
 2  * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
 3  *
 4  * SPDX-License-Identifier: BSD-3-Clause
 5  */
 6
 7 #include <stdio.h>
 8 #include <string.h>
 9 #include "pico/stdlib.h"
10 #include "hardware/i2c.h"
11 #include "pico/binary_info.h"
12
13 /* Example code to drive a 16x2 LCD panel via a I2C bridge chip (e.g. PCF8574)
14
15    NOTE: The panel must be capable of being driven at 3.3v NOT 5v. The Pico
16    GPIO (and therefor I2C) cannot be used at 5v.
17
18    You will need to use a level shifter on the I2C lines if you want to run the
19    board at 5v.
20
21    Connections on Raspberry Pi Pico board, other boards may vary.
22
23    GPIO 4 (pin 6)-> SDA on LCD bridge board
24    GPIO 5 (pin 7)-> SCL on LCD bridge board
25    3.3v (pin 36) -> VCC on LCD bridge board
26    GND (pin 38)  -> GND on LCD bridge board
27 */
28 // commands
29 const int LCD_CLEARDISPLAY = 0x01;
30 const int LCD_RETURNHOME = 0x02;
31 const int LCD_ENTRYMODESET = 0x04;
32 const int LCD_DISPLAYCONTROL = 0x08;
33 const int LCD_CURSORSHIFT = 0x10;
34 const int LCD_FUNCTIONSET = 0x20;
35 const int LCD_SETCGRAMADDR = 0x40;
36 const int LCD_SETDDRAMADDR = 0x80;
37
38 // flags for display entry mode
39 const int LCD_ENTRYSHIFTINCREMENT = 0x01;
40 const int LCD_ENTRYLEFT = 0x02;
41
42 // flags for display and cursor control
43 const int LCD_BLINKON = 0x01;
44 const int LCD_CURSORON = 0x02;
45 const int LCD_DISPLAYON = 0x04;
46
47 // flags for display and cursor shift
48 const int LCD_MOVERIGHT = 0x04;
49 const int LCD_DISPLAYMOVE = 0x08;
50
51 // flags for function set
52 const int LCD_5x10DOTS = 0x04;
53 const int LCD_2LINE = 0x08;
54 const int LCD_8BITMODE = 0x10;
```

```
55
56 // flag for backlight control
57 const int LCD_BACKLIGHT = 0x08;
58
59 const int LCD_ENABLE_BIT = 0x04;
60
61 #define I2C_PORT i2c0
62 // By default these LCD display drivers are on bus address 0x27
63 static int addr = 0x27;
64
65 // Modes for lcd_send_byte
66 #define LCD_CHARACTER  1
67 #define LCD_COMMAND    0
68
69 #define MAX_LINES      2
70 #define MAX_CHARS      16
71
72 /* Quick helper function for single byte transfers */
73 void i2c_write_byte(uint8_t val) {
74     i2c_write_blocking(I2C_PORT, addr, &val, 1, false);
75 }
76
77 void lcd_toggle_enable(uint8_t val) {
78     // Toggle enable pin on LCD display
79     // We cannot do this too quickly or things don't work
80 #define DELAY_US 600
81     sleep_us(DELAY_US);
82     i2c_write_byte(val | LCD_ENABLE_BIT);
83     sleep_us(DELAY_US);
84     i2c_write_byte(val & ~LCD_ENABLE_BIT);
85     sleep_us(DELAY_US);
86 }
87
88 // The display is sent a byte as two separate nibble transfers
89 void lcd_send_byte(uint8_t val, int mode) {
90     uint8_t high = mode | (val & 0xF0) | LCD_BACKLIGHT;
91     uint8_t low = mode | ((val << 4) & 0xF0) | LCD_BACKLIGHT;
92
93     i2c_write_byte(high);
94     lcd_toggle_enable(high);
95     i2c_write_byte(low);
96     lcd_toggle_enable(low);
97 }
98
99 void lcd_clear(void) {
100     lcd_send_byte(LCD_CLEARDISPLAY, LCD_COMMAND);
101 }
102
103 // go to location on LCD
104 void lcd_set_cursor(int line, int position) {
105     int val = (line == 0) ? 0x80 + position : 0xC0 + position;
106     lcd_send_byte(val, LCD_COMMAND);
107 }
108
109 static void inline lcd_char(char val) {
110     lcd_send_byte(val, LCD_CHARACTER);
111 }
112
113 void lcd_string(const char *s) {
114     while (*s) {
115         lcd_char(*s++);
116     }
117 }
```

```
118
119 void lcd_init() {
120     lcd_send_byte(0x03, LCD_COMMAND);
121     lcd_send_byte(0x03, LCD_COMMAND);
122     lcd_send_byte(0x03, LCD_COMMAND);
123     lcd_send_byte(0x02, LCD_COMMAND);
124
125     lcd_send_byte(LCD_ENTRYMODESET | LCD_ENTRYLEFT, LCD_COMMAND);
126     lcd_send_byte(LCD_FUNCTIONSET | LCD_2LINE, LCD_COMMAND);
127     lcd_send_byte(LCD_DISPLAYCONTROL | LCD_DISPLAYON, LCD_COMMAND);
128     lcd_clear();
129 }
130
131 int main() {
132     // This example will use I2C0 on GPIO4 (SDA) and GPIO5 (SCL)
133     i2c_init(I2C_PORT, 100 * 1000);
134     gpio_set_function(4, GPIO_FUNC_I2C);
135     gpio_set_function(5, GPIO_FUNC_I2C);
136     gpio_pull_up(4);
137     gpio_pull_up(5);
138     // Make the I2C pins available to picotool
139     bi_decl( bi_2pins_with_func(4, 5, GPIO_FUNC_I2C));
140
141     lcd_init();
142
143     static uint8_t *message[] =
144             {
145                     "RP2040 by", "Raspberry Pi",
146                     "A brand new", "microcontroller",
147                     "Twin core M0", "Full C SDK",
148                     "More power in", "your product",
149                     "More beans", "than Heinz!"
150             };
151
152     while (1) {
153         for (int m = 0; m < sizeof(message) / sizeof(message[0]); m += MAX_LINES) {
154             for (int line = 0; line < MAX_LINES; line++) {
155                 lcd_set_cursor(line, (MAX_CHARS / 2) - strlen(message[m + line]) / 2);
156                 lcd_string(message[m + line]);
157             }
158             sleep_ms(2000);
159             lcd_clear();
160         }
161     }
162
163     return 0;
164 }
```

Bill of Materials

Table 16. A list of materials required for the example

Item	**Quantity**	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	http://raspberrypi.org/
1602A based LCD panel 3.3v	1	generic part
1602A to I2C bridge device 3.3v	1	generic part
M/M Jumper wires	4	generic part

Appendix B: SDK Configuration

SDK configuration is the process of customising the SDK differently to the defaults. In cases where you do need to make changes for specific circumstances, this chapter will show how that can be done, and what parameters can be changed.

Configuration is done by setting various predefined values in header files in your code. These will override the default values from the SDK itself.

So for example, if you wanted to change the default pins used by the UART, you would add the following to your project header files, before any SDK includes.

```
#define PICO_DEFAULT_UART_TX_PIN 16
#define PICO_DEFAULT_UART_RX_PIN 17
```

Configuration Parameters

Table 17. SDK and Board Configuration Parameters

Parameter name	Defined in	Default	Description
PARAM_ASSERTIONS_DISABLE_ALL	assert.h	0	Global assert disable
PARAM_ASSERTIONS_ENABLED_ADC	adc.h	0	Enable/disable assertions in the ADC module
PARAM_ASSERTIONS_ENABLED_CLOCKS	clocks.h	0	Enable/disable assertions in the clocks module
PARAM_ASSERTIONS_ENABLED_DMA	dma.h	0	Enable/disable DMA assertions
PARAM_ASSERTIONS_ENABLED_FLASH	flash.h	0	Enable/disable assertions in the flash module
PARAM_ASSERTIONS_ENABLED_GPIO	gpio.h	0	Enable/disable assertions in the GPIO module
PARAM_ASSERTIONS_ENABLED_I2C	i2c.h	0	Enable/disable assertions in the I2C module
PARAM_ASSERTIONS_ENABLED_INTERP	interp.h	0	Enable/disable assertions in the interpolation module
PARAM_ASSERTIONS_ENABLED_IRQ	irq.h	0	Enable/disable assertions in the IRQ module
PARAM_ASSERTIONS_ENABLED_PHEAP	pheap.h	0	Enable/disable assertions in the pheap module
PARAM_ASSERTIONS_ENABLED_PIO	pio.h	0	Enable/disable assertions in the PIO module
PARAM_ASSERTIONS_ENABLED_PIO_INSTRUCTIONS	pio_instructions.h	0	Enable/disable assertions in the PIO instructions
PARAM_ASSERTIONS_ENABLED_PWM	pwm.h	0	Enable/disable assertions in the PWM module
PARAM_ASSERTIONS_ENABLED_SPI	spi.h	0	Enable/disable assertions in the SPI module

Parameter name	Defined in	Default	Description
PARAM_ASSERTIONS_ENABLED_SYNC	sync.h	0	Enable/disable assertions in the HW sync module
PARAM_ASSERTIONS_ENABLED_TIME	time.h	0	Enable/disable assertions in the time module
PARAM_ASSERTIONS_ENABLED_TIMER	timer.h	0	Enable/disable assertions in the timer module
PARAM_ASSERTIONS_ENABLED_UART	uart.h	0	Enable/disable assertions in the UART module
PARAM_ASSERTIONS_ENABLE_ALL	assert.h	0	Global assert enable
PICO_CORE1_STACK_SIZE	multicore.h	PICO_STACK_SIZE /0x800	Stack size for core 1
PICO_DEBUG_MALLOC	malloc.h	0	Enable/disable debug printf from malloc
PICO_DEBUG_MALLOC_LOW_WATER	malloc.h	0	Define the lower bound for allocation addresses to be printed by PICO_DEBUG_MALLOC
PICO_DEBUG_PIN_BASE	gpio.h	19	First pin to use for debug output (if enabled)
PICO_DEBUG_PIN_COUNT	gpio.h	3	Number of pins to use for debug output (if enabled)
PICO_DEFAULT_IRQ_PRIORITY	irq.h	0x80	Define the default IRQ priority
PICO_DEFAULT_UART	uart.h	0	Define the default UART used for printf etc
PICO_DEFAULT_UART_BAUD_RATE	uart.h	115200	Define the default UART baudrate
PICO_DEFAULT_UART_RX_PIN	uart.h	1	Define the default UART RX pin
PICO_DEFAULT_UART_TX_PIN	uart.h	0	Define the default UART TX pin
PICO_DISABLE_SHARED_IRQ_HANDLERS	irq.h	0	Disable shared IRQ handers
PICO_HEAP_SIZE	platform_defs.h	0x800	Heap size to reserve
PICO_MALLOC_PANIC	malloc.h	1	Enable/disable panic when an allocation failure occurs
PICO_MAX_SHARED_IRQ_HANDLERS	irq.h	4	Maximum Number of shared IRQ handers
PICO_NO_RAM_VECTOR_TABLE	platform_defs.h	0	Enable/disable the RAM vector table
PICO_PHEAP_MAX_ENTRIES	pheap.h	255	Maximum number of entries in the pheap
PICO_PRINTF_ALWAYS_INCLUDED	printf.h	1 in debug build 0 otherwise	Whether to always include printf code even if only called weakly (by panic)
PICO_PRINTF_DEFAULT_FLOAT_PRECISION	printf.c	6	Define default floating point precision
PICO_PRINTF_FTOA_BUFFER_SIZE	printf.c	32	Define printf ftoa buffer size

Parameter name	Defined in	Default	Description
PICO_PRINTF_MAX_FLOAT	printf.c	1e9	Define the largest float suitable to print with %f
PICO_PRINTF_NTOA_BUFFER_SIZE	printf.c	32	Define printf ntoa buffer size
PICO_PRINTF_SUPPORT_EXPONENTIAL	printf.c	1	Enable exponential floating point printing
PICO_PRINTF_SUPPORT_FLOAT	printf.c	1	Enable floating point printing
PICO_PRINTF_SUPPORT_LONG_LONG	printf.c	1	Enable support for long long types (%llu or %p)
PICO_PRINTF_SUPPORT_PTRDIFF_T	printf.c	1	Enable support for the ptrdiff_t type (%t)
PICO_SHARED_IRQ_HANDLER_DEFAULT_ORDER_PRIORITY	irq.h	0x80	Set default shared IRQ order priority
PICO_SPINLOCK_ID_CLAIM_FREE_END	sync.h	31	Spinlock ID for claim free end
PICO_SPINLOCK_ID_CLAIM_FREE_FIRST	sync.h	24	Spinlock ID for claim free first
PICO_SPINLOCK_ID_HARDWARE_CLAIM	sync.h	11	Spinlock ID for Hardware claim protection
PICO_SPINLOCK_ID_IRQ	sync.h	9	Spinlock ID for IRQ protection
PICO_SPINLOCK_ID_STRIPED_FIRST	sync.h	16	Spinlock ID for striped first
PICO_SPINLOCK_ID_STRIPED_LAST	sync.h	23	Spinlock ID for striped last
PICO_SPINLOCK_ID_TIMER	sync.h	10	Spinlock ID for Timer protection
PICO_STACK_SIZE	platform_defs.h	0x800	Stack Size
PICO_STDIO_DEFAULT_CRLF	stdio.h	1	Default for CR/LF conversion enabled on all stdio outputs
PICO_STDIO_ENABLE_CRLF_SUPPORT	stdio.h	1	Enable/disable CR/LF output conversion support
PICO_STDIO_SEMIHOSTING_DEFAULT_CRLF	stdio_semihosting.h	PICO_STDIO_DEFAULT_CRLF	Default state of CR/LF translation for semihosting output
PICO_STDIO_STACK_BUFFER_SIZE	stdio.h	128	Define printf buffer size (on stack)... this is just a working buffer not a max output size
PICO_STDIO_UART_DEFAULT_CRLF	stdio_uart.h	PICO_STDIO_DEFAULT_CRLF	Default state of CR/LF translation for UART output
PICO_STDIO_USB_DEFAULT_CRLF	stdio_usb.h	PICO_STDIO_DEFAULT_CRLF	Default state of CR/LF translation for USB output
PICO_STDIO_USB_LOW_PRIORITY_IRQ	stdio_usb.h	31	low priority (non hardware) IRQ number to claim for tud_task() background execution
PICO_STDIO_USB_STDOUT_TIMEOUT_US	stdio_usb.h	500000	Number of microseconds to be blocked trying to write USB output before assuming the host has disappeared and discarding data

Parameter name	Defined in	Default	Description
PICO_STDIO_USB_TASK_INTERVAL_US	stdio_usb.h	1000	Period of microseconds between calling tud_task in the background
PICO_STDOUT_MUTEX	stdio.h	1	Enable/disable mutex around stdout
PICO_TIME_DEFAULT_ALARM_POOL_DISABLED	time.h	0	Disable the default alarm pool
PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM	time.h	3	Select which HW alarm is used for the default alarm pool
PICO_TIME_DEFAULT_ALARM_POOL_MAX_TIMERS	time.h	16	Selects the maximum number of concurrent timers in the default alarm pool
PICO_TIME_SLEEP_OVERHEAD_ADJUST_US	time.h	6	How many microseconds to wake up early (and then busy_wait) to account for timer overhead when sleeping in low power mode
PICO_UART_DEFAULT_CRLF	uart.h	0	Enable/disable CR/LF translation on UART
PICO_UART_ENABLE_CRLF_SUPPORT	uart.h	1	Enable/disable CR/LF translation support
PICO_USE_MALLOC_MUTEX	malloc.h	1 with pico_multicore, 0 otherwise	Whether to protect malloc etc with a mutex
USB_DPRAM_MAX	usb.h	4096	Set amount of USB RAM used by USB system

Appendix C: CMake Build Configuration

CMake configuration variables can be used to customize the way the SDK performs builds

Configuration Parameters

Table 18. CMake Configuration Variables

Parameter name	Defined in	Default	Description
PICO_BARE_METAL	CMakeLists.txt	0	Flag to exclude anything except base headers from the build
PICO_BOARD	board_setup.cmake	rp2040	The board name being built for. This is overridable from the user environment
PICO_BOARD_CMAKE_DIRS	board_setup.cmake	""	Directors to look for <PICO_BOARD>.cmake in. This is overridable from the user environment
PICO_BOARD_HEADER_DIRS	generic_board.cmake	""	Directors to look for <PICO_BOARD>.h in. This is overridable from the user environment
PICO_CMAKE_RELOAD_PLATFORM_FILE	pico_pre_load_platform.cmake	none	custom CMake file to use to set up the platform environment
PICO_COMPILER	pico_pre_load_toolchain.cmake	none	Optionally specifies a different compiler (other than pico_arm_gcc.cmake) - this is not yet fully supported
PICO_CONFIG_HEADER_FILES	CMakeLists.txt	""	List of extra header files to include from pico/config.h for all platforms
PICO_CONFIG_HOST_HEADER_FILES	CMakeLists.txt	""	List of extra header files to include from pico/config.h for host platform
PICO_CONFIG_RP2040_HEADER_FILES	CMakeLists.txt	""	List of extra header files to include from pico/config.h for rp2040 platform
PICO_CXX_ENABLE_CXA_ATEXIT	CMakeLists.txt	0	Enabled cxa-atexit
PICO_CXX_ENABLE_EXCEPTIONS	CMakeLists.txt	0	Enabled CXX exception handling
PICO_CXX_ENABLE_RTTI	CMakeLists.txt	0	Enabled CXX rtti
PICO_DEFAULT_BOOT_STAGE2_FILE	CMakeLists.txt	.../boot2_w25q080.S	Default stage2 file to use unless overridden by pico_set_boot_stage2 on the TARGET
PICO_NO_GC_SECTIONS	CMakeLists.txt	0	Disable -ffunction-sections -fdata-sections and --gc-sections
PICO_NO_HARDWARE	rp2_common.cmake	1 for PICO_PLATFORM host 0 otherwise	OPTION: Whether the build is not targeting an RP2040 device

Parameter name	Defined in	Default	Description
PICO_NO_TARGET_NAME	rp2_common.cmake	0	Don't defined PICO_TARGET_NAME
PICO_NO_UF2	rp2_common.cmake	0	Disable UF2 output
PICO_ON_DEVICE	rp2_common.cmake	0 for PICO_PLATFORM host 1 otherwise	OPTION: Whether the build is targeting an RP2040 device
PICO_PLATFORM	pico_pre_load_platform.cmake	rp2040 or environment value	platform to build for e.g. rp2040/host
PICO_STDIO_SEMIHOSTING	CMakeLists.txt	0	OPTION: Globally enable stdio semihosting
PICO_STDIO_UART	CMakeLists.txt	1	OPTION: Globally enable stdio UART
PICO_STDIO_USB	CMakeLists.txt	0	OPTION: Globally enable stdio USB
PICO_TOOLCHAIN_PATH	pico_pre_load_toolchain.cmake	none (i.e. search system paths)	Path to search for compiler

Control of binary type produced (advanced)

These variables control how executables for RP2040 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 copied into RAM at startup.

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

TIP

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

Appendix D: Board Configuration

Board Configuration

Board configuration is the process of customising the SDK to run on a specific board design. The SDK comes some predefined configurations for boards produced by Raspberry Pi, the main (and default) example being the Raspberry Pi Pico.

Configurations specify a number of parameters that could vary between hardware designs. For example, default UART ports, on board LED locations and flash capacities etc.

This chapter will go through where these configurations files are, how to make changes and set parameters, and how to build your SDK using CMake with your customisations.

The Configuration files

Board 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/tree/master/src/boards/include/boards/pico.h Lines 1 - 52

```
1  /*
2   * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
3   *
4   * SPDX-License-Identifier: BSD-3-Clause
5   */
6
7  // ------------------------------------------------------
8  // NOTE: THIS HEADER IS ALSO INCLUDED BY ASSEMBLER SO
9  //       SHOULD ONLY CONSIST OF PREPROCESSOR DIRECTIVES
10 // ------------------------------------------------------
11
12 // This header may be included by other board headers as "boards/pico.h"
13
14 #ifndef _BOARDS_PICO_H
15 #define _BOARDS_PICO_H
16
17 #ifndef PICO_DEFAULT_UART
18 #define PICO_DEFAULT_UART 0
19 #endif
20
21 #ifndef PICO_DEFAULT_UART_TX_PIN
22 #define PICO_DEFAULT_UART_TX_PIN 0
23 #endif
24
25 #ifndef PICO_DEFAULT_UART_RX_PIN
26 #define PICO_DEFAULT_UART_RX_PIN 1
27 #endif
28
29 #ifndef PICO_DEFAULT_LED_PIN
30 #define PICO_DEFAULT_LED_PIN 25
31 #endif
```

```
32
33 #ifndef PICO_FLASH_SPI_CLKDIV
34 #define PICO_FLASH_SPI_CLKDIV 2
35 #endif
36
37 #ifndef PICO_FLASH_SIZE_BYTES
38 #define PICO_FLASH_SIZE_BYTES (2 * 1024 * 1024)
39 #endif
40
41 // Drive high to force power supply into PWM mode (lower ripple on 3V3 at light loads)
42 #define PICO_SMPS_MODE_PIN 23
43
44 #ifndef PICO_FLOAT_SUPPORT_ROM_V1
45 #define PICO_FLOAT_SUPPORT_ROM_V1 1
46 #endif
47
48 #ifndef PICO_DOUBLE_SUPPORT_ROM_V1
49 #define PICO_DOUBLE_SUPPORT_ROM_V1 1
50 #endif
51
52 #endif
```

As can be seen, it sets up the default UART to `uart0`, the GPIO pins to be used for that UART, the GPIO pin used for the onboard LED, and the flash size.

To create your own configuration file, create a file in the board `../source/folder` with the name of your board, for example, `myboard.h`. Enter your board specific parameters in this file.

Building applications with a custom board configuration

The CMake system is what specifies which board configuration is going to be used.

To create a new build based on a new board configuration (we will use the `myboard` example from the previous section) first create a new build folder under your project folder. For our example we will use the pico-examples folder.

```
$ cd pico-examples
$ mkdir myboard_build
$ cd myboard_build
```

then run cmake as follows:

```
cmake -D"PICO_BOARD=myboard" ..
```

This will set up the system ready to build so you can simply type `make` in the `myboard_build` folder and the examples will be built for your new board configuration.

Available configuration parameters

Table 17 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.

Appendix E: Building the SDK API documentation

The SDK documentation can also be found as part of 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,

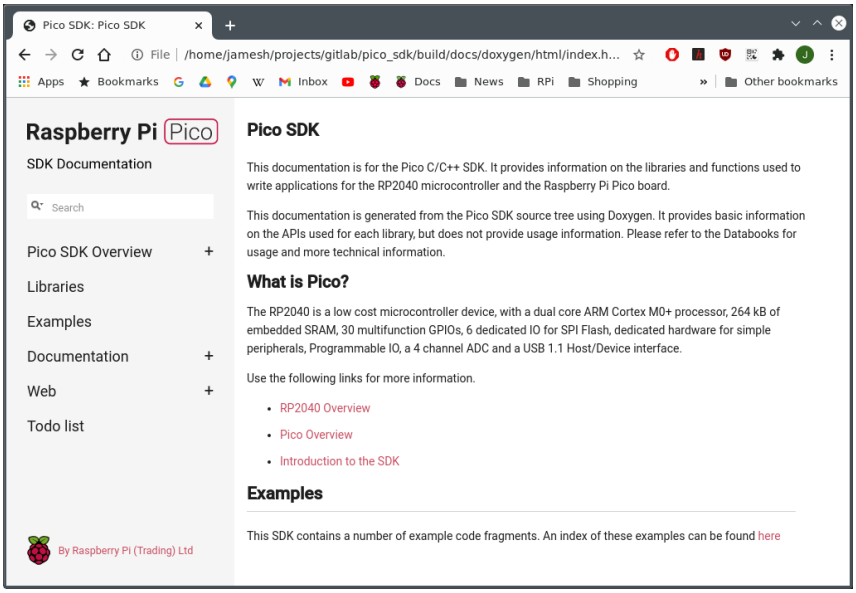
```
$ cd ~/
$ mkdir pico
$ cd pico
$ git clone -b master https://github.com/raspberrypi/pico-sdk.git
$ cd pico-sdk
$ git submodule update --init --recursive
$ cd ..
$ git clone -b master https://github.com/raspberrypi/pico-examples.git
```

and then afterwards you can go ahead and build the documentation.

```
$ cd pico-sdk
$ mkdir build
$ cd build
$ cmake ..
$ make docs
```

The API documentation will be build and can be found in the `pico-sdk/build/docs/doxygen/html` directory, see Figure 15.

Figure 15. The SDK API documentation



When building the documentation `cmake` will go ahead and clone the `pico-examples` repo into the `build` directory directly from Github. This can be over ridden if you have a local copy on the command line,

```
$ cmake -DPICO_EXAMPLES_PATH=../../pico-examples ..
```

or by using an environment variable.

```
$ export PICO_EXAMPLES_PATH=../../pico-examples
$ cmake ..
```

Raspberry Pi is a trademark of the Raspberry Pi Foundation

Raspberry Pi Trading Ltd